Cahors (2.20.0.3)

© 2015- 2024 XSharp BV

Table of Contents

X#

Part

I

1 X# Documentation

Welcome to X# (XSharp) Cahors (2.20.0.3)

[Click here for the version history](#)
[Click here for the list of known issues](#)
X# is an open source development language for .NET, based on the xBase language.

It comes in different flavors, such as Core, Visual Objects, Vulcan.NET, xBase++, Harbour, Foxpro and more.
The current version of X# supports the **Core** dialect as well as the **VO, Vulcan and Harbour** dialect

At this moment X# supports the following Xbase dialects:

Dialect	Syntax	Classes	Functions
Core	Complete	Complete	Complete
Visual Objects	Complete	Complete	Complete
Vulcan	Complete	Complete	Complete

Xbase++	Complete	Partially.	Partially
(Visual) FoxPro	Partially.	Partially.	Partially.
(X)Harbour	Partially.	Partially.	Partially.

For more info about the dialects see the [dialects topic](#) in this documentation

This documentation is still under development. Some sections in the documentation are incomplete.

1.1 Getting Started with X#

At this moment X# supports the following Xbase dialects:

Dialect	Syntax	Classes	Functions
Core	Complete	Complete	Complete
Visual Objects	Complete	Complete	Complete
Vulcan	Complete	Complete	Complete
Xbase++	Complete	Partially.	Partially
(Visual) FoxPro	Partially.	Partially.	Partially.
(X)Harbour	Partially.	Partially.	Partially.

For more info about the dialects see the dialects topic in this documentation

Click here for the version history
Click here for the list of known issues

1.1.1 Dialects

The X# compiler can run in different dialects. The table below shows the differences between these dialects.
You can use the /dialect compiler option to select the dialect that you want to use.
The Core dialect is default. All dialects apart from the Core dialect require a reference to the XSharp runtime DLLs

Feature	Core	VO	Vulcan	Harbour	Xbase++ + 14	FoxPro
4 Letter abbreviations of keywords (FUNC, WHIL etc).	-	Yes [11]	-	Yes [11]	Follow the conventions of Xbase++	Yes [11]
&& as single line comment character (alternative syntax for //)	-	Yes	-	Yes	Yes	Yes
***** at beginning of line as comment character	Yes	Yes	Yes	Yes	Yes	Yes
ALIAS (->) operator	-	Yes	Yes	Yes	Yes	Yes
AND, NOT, OR, XOR as alternatives for .AND., .NOT. etc	-	-	-	-	-	Yes
ARRAY type, Including Array literals & NULL_ARRAY, including missing elements	-	Yes	Yes	Yes	Yes	Yes
BREAK statement	-	Yes	Yes	Yes	Yes	Yes
BEGIN SEQUENCE .. RECOVER .. END SEQUENCE	-	Yes	Yes	Yes	Yes	Yes
CLIPPER calling convention (requires USUAL support)	-	Yes	Yes	Yes	Yes	Yes
CODEBLOCK type	-	Yes	Yes	Yes	Yes	Yes
DATE type, including NULL_DATE	-	Yes	Yes	Yes	Yes	Yes
DATE and DateTime literals	Yes	Yes	Yes	Yes	Yes	Yes
DEFINE statement with optional type clause	Yes	Yes	Yes	Yes	Yes	Yes

Feature	Core	VO	Vulcan	Harbour	Xbase+ + 14	FoxPro
FIELD statement, and recognition of Field names in stead of locals or instance variables	-	Yes	Yes	Yes	Yes	Yes
FLOAT type, including Float literals & the **/vo14** compiler option	-	Yes	Yes	Yes	Yes	Yes
FOR .. ENDFOR as alternative for **FOR .. NEXT**	-	-	-	-	Yes	Yes
FOR EACH as alternative for **FOREACH**	-	-	-	-	-	Yes
GLOBAL statement	Yes	Yes	Yes	Yes	Yes	Yes
IIF() expression and the **/vo10** compiler option	Yes [7]	Yes [7]	Yes [7]	Yes	Yes	Yes
Late binding	-	Yes (with /lb +)	Yes (with /lb +)	Yes	Yes	Yes
LPARAMETERS statement	-	-	-	-	-	Yes
MACRO compiler	- [13]	Yes	Yes	Yes	Yes	Yes
MEMVAR, PUBLIC, PRIVATE, PARAMETERS statement , including undeclared variables [14]	-	Yes	-	Yes	Yes	Yes
PSZ Type, including NULL_PSZ and (pseudo) PSZ conversion functions (String2Psz(), Cast2Psz()	-	Yes	Yes	Yes	Yes	Yes
PSZ Indexer	n/a	1 based	**0 based**	1 based	1 based	1 based
NULL_STRING	Yes [9]	Yes [9]	Yes	Yes	Yes	Yes
SYMBOL type, including Symbol literals & NULL_SYMBOL	-	Yes	Yes	Yes	Yes	Yes
Statements before the first entity in a source file	-	-	-	-	-	Yes

Feature	Core	VO	Vulcan	Harbour	Xbase++ + 14	FoxPro
USUAL type, including NIL literal (1,2)	-	Yes	Yes	Yes	Yes	Yes 15
TEXT Command	-	-	-	-	-	Yes
Special features				Yes	Yes	Yes
VOSTRUCT and **AS/IS** syntax (10)	-	Yes	Yes	-	-	-
UNION and **AS/IS** syntax (10)	-	Yes	Yes	-	-	-
Untyped Literal Arrays	-	Yes 8	Yes 8	Yes	Yes	Yes
Missing types allowed	-	Yes 3	Yes 3	Yes	Yes	Yes
Missing arguments in function/method calls MyFunc(1,,2)	-	Yes	Yes	Yes	Yes	Yes
Pseudo functions				Yes	Yes	Yes
PCount(), _GetMParam(), _GetFParam(), _Args()	-	Yes 4	Yes 4	Yes 4	Yes 4	Yes 4
SLen()	-	Yes 5	Yes 5	Yes 5	Yes 5	Yes 5
_Chr()	-	Yes	Yes	Yes	Yes	Yes
AltD()	-	Yes 6	Yes 6	Yes 6	Yes 6	Yes 6
PCall, PCallNative	-	Yes	Yes	Yes	Yes	Yes
CCall, CCallNative	-	Yes	Yes	Yes	Yes	Yes
Other information						
Generated Function class name	Functions X$<ModuleName>$Functions for static functions and globals	<AssemblyName>.Functions (DLLs) <AssemblyName>.Exe.Functions (EXEs) <AssemblyName>.Exe.$<ModuleName>$.Functions for static functions and globals				

1. Requires VulcanRT.DLL or XSharp.VO.DLL
2. Requires VulcanRTFuncs.DLL or XSharp.VO.DLL
3. Missing types get translated to USUAL. With the new /vo15 compiler option you can tell the compiler that you do not want to allow missing types
4. Only supported inside functions and methods with Clipper Calling convention

X#

5. The Vulcan runtime does not have a SLen() function. The compiler translates this to accessing the Length property of the string

6. This gets translated to

    IF System.Diagnostics.Debugger.Attached
        System.Diagnostics.Debugger.Break()
    ENDIF

7. In the core dialect each of the expressions is cast to OBJECT. In the other dialects the expression is cast to USUAL.

8. Untyped literal arrays in the VO/Vulcan dialect are translated to the VO Array type:
{"aa", "bb", "cc"}
Typed literal arrays are prefixed with <Type> like this:
<STRING>{"aa", "bb", "cc"}
This becomes a literal string array

9. With the /vo2 compiler option NULL_STRING is compiled to an empty string. Otherwise to a NULL.

10. The VOSTRUCT and UNION can be replaced with a standard .NET structure. For Union you need to use the [StructLayout(LayoutKind.Explicit)] attribute on the structure and the [FieldOffSet()] attribute on each field to explicitly define the location of the fields.

11. Only for VO compatible keywords, not for keywords introduced in Vulcan or XSharp.

12. The true and false expressions for an IIF() are optional in Harbour and Xbase++

13. The macro compiler is not supported. However when you use the RDD system in the Core dialect then the index expression will still use the macro compiler and a reference to the XSharp.RT.DLL is needed for the support of the macro compiler.

14. This requires the XSharp runtime.

15. In FoxPro NIL is not allowed and redefined to FALSE.

1.1.1.1   Core

The compiler and runtime have the following "special" behavior when compiling for the "Core" dialect.

Compiler

- Does NOT allows 4 letter abbreviations of keywords
- Allows the DOT ('.') operator to call Instance methods
- Single quotes are used for Character literals
- String Comparisons are mapped to the String.Compare() method in the .Net runtime
- The String "=" operator is not supported
- The String "-" operator is not supported
- Anything that requires runtime support, such as the X# specific types (DATE, ARRAY, SYMBOL, FLOAT and USUAL) and dynamic memory variables is not supported
- Supports the use of '@' to retrieve the address of a variable. This may also be used for REF variables if the compiler option -vo7 is used.
- The '=' operator may be used for assigns but will generate a warning when used.
- The compiler generated functions class is called
    Functions   for normal functions and globals
    X$<ModuleName>$Functions for static functions and globals
- Procedures cannot return values
- Does not allow skipping arguments in method calls.

Runtime

- The Core dialect does not require a runtime. However you can still link to XSharp.Core and XSharp.RDD and call methods and functions in these assemblies.

1.1.1.2   All Non Core dialects

The compiler and runtime have the following "special" behavior when compiling for any or the non-core dialects

Compiler

- Allows "garbage" after keywords such as NEXT, ENDDO etc.
- Does not allows the DOT ('.') operator to call Instance methods
- Requires a reference to the XSharp.Core and XSharp.RT DLLs
- String Comparisons are mapped to a function in the XSharp runtime
- NULL_STRING is compiled into either "" or NULL depending on the compiler option -vo2
- Supports literal symbols (#SomeName)
- The String "=" operator is mapped to a function in the XSharp runtime
- The X# specific types such as DATE, ARRAY, SYMBOL, FLOAT and USUAL are not supported, but required a reference to the runtime
- Adds support for Dynamic Memory Variables and Undeclared variables
- Single quotes are used for String literals (except in Vulcan). Character literals must be prefixed with a c, like this: cChar := c'A'
- Allow the use of ASend() to call methods for each element inside a X# array.
- The '=' operator may be used for assigns but will generate a warning when used.
- Adds support for BEGIN SEQUENCE .. END SEQUENCE
- Adds support for the ALIAS (->) operator
- Adds support for the FIELD statement
- Adds support for the Macro compiler and &(variable) syntax
- Adds support for the ARRAY OF <type> syntax
- The compiler generated functions class is called
  <AssemblyName>.Functions (DLLs) for functions and globals
  <AssemblyName>.Exe.Functions (EXEs) for functions and globals
  <AssemblyName>.Exe.$<ModuleName>$.Functions for static functions and
globals
- The compiler adds several attributes (defined in XSharp.Core) to describe the default namespace and compiler version
- The compiler generates code for EXE files that set several properties in the Runtimestate to match compiler options and the dialect of the main app.
- Procedures cannot return values
- Adds support for untyped variables and return values
- Adds support for untyped function and method parameters (the so-called Clipper calling convention)
- Adds support for late bound code (requires the -lb compiler option)
- Adds support for INIT and EXIT procedures
- Adds support for Codeblocks (untyped Lambda expressions with an array of USUAL parameters and a USUAL return value)
- Allows skipping arguments in method calls. Skipped arguments are assumed to be NIL

Runtime

- The default RDD all except the FoxPro dialect is DBFNTX

1.1.1.3 **Visual Objects**

This dialect shares the features of "All Non Core Dialects"
The compiler and runtime have the following "special" behavior when compiling for the "Visual Objects" dialect.

Compiler

- Allows 4 letter abbreviations of some older keywords
- Allows "&&" as same line comment characters, just like "//"
- When a reference to XSharp.VO is added then certain functions that are VO specific are enabled, such as RtRegString()
- Supports the use of '@' to retrieve the address of a variable. This may also be used for REF variables if the compiler option -vo7 is used.
- The preprocessor adds a define __VO__ with a value of TRUE
- Adds the VOSTRUCT and UNION entity types
- Uses the _WINBOOL type for logical values inside VOSTRUCT and UNION entities
- The indexer on PSZ types start with element 1

Runtime

- When running in Ansi more (SetAnsi(TRUE), which is the default) then the DBF header for DBFNTX gets the Ansi bit set

1.1.1.4 **Vulcan**

This dialect shares the features of "All Non Core Dialects"
The compiler and runtime have the following "special" behavior when compiling for the "Vulcan" dialect.

Compiler

- Does NOT allows 4 letter abbreviations of keywords
- Does NOT support Memory variables
- Does NOT allow && as same line comment characters (&& means .AND. in Vulcan)
- Single quotes are used for Character literals
- Supports the use of '@' to retrieve the address of a variable. This may also be used for REF variables if the compiler option -vo7 is used.
- NULL_STRING is compiled into either "" or NULL depending on the compiler option -vo2
- The preprocessor adds a define __VULCAN__ with a value of TRUE
- Adds the VOSTRUCT and UNION entity types
- Uses the _WINBOOL type for logical values inside VOSTRUCT and UNION entities
- The indexer on PSZ types start with element 0

Runtime

- When running in Ansi more (SetAnsi(TRUE), which is the default) then the DBF header for DBFNTX gets the Ansi bit set
- The NoMethod() method gets an extra 1st parameter that contains the name of the method that was called.

1.1.1.5 Xbase++

This dialect shares the features of "All Non Core Dialects"
The compiler and runtime have the following "special" behavior when compiling for the "Xbase++" dialect.

Compiler

- Allows 4 letter abbreviations of those keywords where this is also supported in Xbase++
- Allows "&&" as same line comment characters, just like "//"
- The '@' operator is only used to pass variables by reference.
- The preprocessor adds a define __XPP__ with a value of TRUE
- Allows ENDFOR instead of NEXT
- Adds the Xbase++ specific CLASS syntax to define classes
- The entry point to the code is the main() function or main() procedure.

Runtime

- FieldGet() and FieldPut() will not throw an error when writing to non existing field Positions
- If you use the index operator on a usual which contains a string then you will be receive the 1 based substring of the value.

1.1.1.6 FoxPro

This dialect shares the features of "All Non Core Dialects"
The compiler and runtime have the following "special" behavior when compiling for the "FoxPro" dialect.

Compiler

- Allows 4 letter abbreviations of some older keywords
- Allows "&&" as same line comment characters, just like "//"
- Allows the DOT ('.') operator to call Instance methods
- The '@' operator is only used to pass variables by reference.
- Allows ENDFOR instead of NEXT and FOR EACH instead of FOREACH
- The '=' operator will NOT generate a warning when used as assignment operator
- Adds several keywords such as THIS (as alias for SELF)
- Adds support for CursorName.FieldName syntax
- Adds support for M.VariableName syntax
- Adds the DIMENSION statement syntax
- Adds the LPARAMETERS statement

- Adds the TEXT .. ENDTEXT statement
- Adds the \\ and \\\ statement
- Adds the "= <Expression>" command
- Adds the FoxPro specific DEFINE CLASS syntax to define classes, including the use of FUNCTION and PROCEDURE to define methods inside a class and the use of the _ACCESS and _ASSIGN suffixes on the names of these functions and procedures to declare access/assign methods
- Procedures may return values and are therefore just like Functions
- Allows code before the first entity in a source file. This will be compiled into a function with the same name as the PRG file
- Adds support for the DoDefault() pseudo function
- When compiled with /fox1 then the compiler assumes that all classes inherit from the Custom class and will generate special code when declaring classes with the DEFINE CLASS syntax.
- The NIL keyword in FoxPro has the property 'unitialized' but a value of FALSE.

Runtime

- The default RDD in the FoxPro dialect is DBFVFP
- The MemoWrit() function adds an extra ^Z character to the end of file. MemoRead() removes this character when it finds it.
- The DBF() function returns the full name of the file
- The _MRelease() function does not clear the memory variables but completely releases them
- New memory variables are always filled with a value of FALSE
- When comparing an initialized USUAL value with an unitialized value then in the FoxPro dialect an error will be generated.
  The other dialects will simply return FALSE.

1.1.1.7   Harbour

This dialect shares the features of "All Non Core Dialects"
The compiler and runtime have the following "special" behavior when compiling for the "Harbour" dialect.

Compiler

- Allows 4 letter abbreviations of some older keywords
- The '@' operator is only used to pass variables by reference.
- Allows IIF() expressions with a missing element, for example IIF(someCondition,DoSomething(),). The compiler will insert a NIL for a missing entry.

Runtime

X#

1.1.2 Bring Your Own Runtime (BYOR)

> **The X# Runtime is now available. There is no need anymore to compile against the Vulcan Runtime !**
> **For now we still support the Vulcan runtime, but that support may be dropped in a future build of X#.**

VO and Vulcan support is available in this build of XSharp through what we call the Bring Your Own Runtime principle.

If you own a license of Vulcan, you can copy the DLLs that you find in the <Vulcan.NET BaseFolder>\Redist\4.0 folder to a folder that is inside your solution.
Then add references to the DLLs that you need in your project.

DLLs you MUST add if you compile for the VO/Vulcan dialect with the Vulcan runtime:

- VulcanRT.DLL
- VulcanRTFuncs.DLL

These 2 files are NEVER added to your Vulcan projects, Vulcan adds a reference to these DLLs automatically. XSharp does not do that, so you should add them yourself.

DLLs that you MAY want to add, depending on what you are using in your application:

- VulcanVOSystemClasses.dll
- VulcanVORDDClasses.dll
- VulcanVOGUIClasses.dll
- VulcanVOInternetClasses.dll
- VulcanVOSQLClasses.dll
- VulcanVOConsoleClasses.dll
- VulcanVOWin32APILibrary.dll

DLLs that you normally do NOT add to your project (these are handled automatically by the Vulcan Runtime)

- VulcanRDD.DLL
- VulcanMacroCompiler.DLL
- VulcanDBFCDX.dll
- VulcanDBFFPT.dll

1.1.3 Known Issues

Below is the complete (but short) list of things that are not supported yet or are known problems in the current build of X#.

Compiler

- Some parser errors need improvements, compiler errors may be cryptic every now and then
- Sometimes you may see a "Include file not found" error, where the file DOES exist. This seems to be a side effect of the codepage that was used to save the include files.
  When you save the file as UTF8 (File Advanced Save Options in Visual Studio) then this problem almost always disappears.

Visual Studio

- New features in the Debugger, such as a window for Globals and RDD Workareas

RDD System

- Query Optimization (Rushmore) is not supported yet

Runtime

- Some runtime functions are not supported yet.

1.1.4   Installation

When you install XSharp you will find the following folders on your machine.

Folder	Setup component	Contents
C:\Program Files (x86)\XSharp	Main	Main installation folder
Assemblies	Main	The runtime assemblies that are selectable in the Add Reference dialog inside Visual Studio. Please note that this is a subset of the folders in the Redist Folder. For deploying your apps please use the files in the Redist folder. In a future version this folder may contain so called reference assemblies (assemblies with only meta data and no code).
Bin	main	The command line compiler, the Script interpreter and Vulcan XPorter
Debug	main	Debug versions of the runtime DLLs and SDK DLLs (**FOX only**)
Extension	main	The components that must be installed inside Visual Studio for the X# project system and Language support. If you want to reinstall the project systems you should run the deployvs<num>.cmd files in the Uninst folder. These files were created at install time and match your specific configuration of Visual Studio versions.
Help	main	The PDF, CHM help files and the Visual Studio help
Images	main	Some images
Include	main	XSharpDefs.xh and other header files needed by X#
MsBuild	main	The MsBuild integration for XsProj files
NetCore31	main	A version of the X# compiler and script engine for .Net Core 3.1 (**FOX only**)
ProjectSystem	main	VSIX files to install the project system and debugger inside Visual Studio. Only use these when the X# support team tells you to do so.
Redist	main	Files that you may want to include with your application compiled with X#.

Templates	main	Contains the templates for the VO Compatible editors for Windows, Servers and Menus
Tools	main	Tools that are used during installation,
Uninst	main	The uninstaller and some cmd files generated during installation to register your X# Extension inside Visual Studio. It also contains cmd files created at install time to help generating native images for the X# compiler. The cmd files with the number 1-6 are used to install into different versions of Visual Studio.
VOXPorter	main	The VOXPorter
XIDE	xide	The XIDE installer
<Global Assembly Cache>	main\gac	X# runtime files will be installed in the GAC when this component is selected
<Global Assembly Cache>\NativeImages	main\ngen	The X# compiler files will be precompiled when the option "Optimize performance by generating native images" is selected
<Common Documents\XSharp\Examples	main\examples	The X# examples will be installed in the common documents folder when this option is selected
Visual Studio folders	main\vs..	X# will be integrated into Visual Studio when this is selected
Visual Studio 2017 and/or 2019 Assuming you have installed (one of) these in the default location: c:\Program Files (x86)\Microsoft Visual Studio**<number>****<Version>** then X# will be in the subfolder • Common7\IDE\Extensions\XSharp	main\vs1 .. main\vs6	• **<number>** is 2017 or 2019. • **<Version>** can be one of Professinal, Community, Enterprise or Buildtools. There can be more that one edition of VS2017/VS2019 on your machine. The installer will show all instances that are detected.
Visual Studio 2022 Assuming you have installed VS 2022 in the default location: c:\Program Files\Microsoft Visual Studio\2022**<Version>** then X# will be in the subfolder • Common7\IDE\Extensions\XSharp	main\vs1 .. main\vs6	• **<Version>** can be one of Professinal, Community, Enterprise or Buildtools. There can be more that one edition of VS2017/VS2019 on your machine. The installer will show all instances that are detected.
Registry	main main\scr	

The installer will also write some   ipt
changes to the registry:

- HKLM\Software\XSharp
- HKLM\Software\Microsoft\.NETF
  ramework\v4.0.30319\Assembly
  FoldersEx\XSharp
- HKCR\.ppo
- HKCR\.prg
- HKCR\.prgx
- HKCR\.vh
- HKCR\.xh
- HKCR\.xs
- HKCR\.xsproj
- HKCR\XSharpScript
- HKCR\XSharp.headerfile
- HKCR\XSharp.ppofile
- HKCR\XSharp.sourcefile

Environment variables
The installer creates / modifies the
following environment variables
XSHARPPATH
XSHARPMSBUILDDIR

See the topic Building XSharp apps with Visual Studio and/or MsBuild for a description
how the various registry settings and folders are involved with the build process with
MsBuild and Visual Studio.

1.1.5 Redistributing X#

You are allowed to distribute the following XSharp DLLs with your projects:
You are also allowed to distribute the PDB files that come with these DLLs.
These PDB files may help in locating the line numbers where an error occurs.
These files can be found in the <Program Files>\XSharp\Redist folder on your machine.

XSharp.Runtime

Name	Needed for
XSharp.CodeAnalysis.dll	Full macro compiler & Scripting
XSharp.Core.dll	All dialects
XSharp.Data.dll	All dialects
XSharp.MacroCompiler.dll	Macro compiler
XSharp.MacroCompiler.Full.dll	Full Macro compiler
XSharp.RT.dll	All dialects
XSharp.RT.Debugger.dll	All dialects
XSharp.RDD.dll	All dialects
XSharp.Scripting.dll	Full Macro compiler & Scripting
XSharp.VFP.dll	FoxPro dialect
XSharp.VFP.UI.dll	FoxPro dialect
XSharp.VO.dll	VO and Vulcan dialect
XSharp.XPP.dll	Xbase++ Dialect

VO Compatible SDK

Name
VOConsoleClasses.dll
VORDDClasses.dll
VOSQLClasses.dll
VOGUIClasses.dll
VOReportClasses.dll
VOSystemClasses.dll
VOWin32APILibrary.dll

Strongly Typed version of the VO SDK

Name
XSharp.VOConsoleClasses.dll
XSharp.VOGUIClasses.dll
XSharp.VORDDClasses.dll
XSharp.VOSQLClasses.dll
XSharp.VOSystemClasses.dll

The following files are needed by VOGUIClasses.dll when you use the DataBrowser or Splitwindow

Name
CATO3CNT.DLL
CATO3DAT.DLL
CATO3MSK.DLL
CATO3NBR.DLL
CATO3SBR.DLL
CATO3TBR.DLL
CATO3TIM.DLL
CATO3SPL.DLL
MSVCRT.DLL

The following file is needed for image support in VOGUIClasses.DLL

CAPAINT.DLL

XSharp Scripting

Name

Xsi.exe

XSharp.Scripting.dll

XSharp.CodeAnalysis.dll

These scripting files can all be found in the <Program Files>\XSharp\Bin folder

Support files

Name

Microsoft.DiaSymReader.Native.amd64.dll

Microsoft.DiaSymReader.Native.x86.dll

System.*.dll

These support files can all be found in the <Program Files>\XSharp\Bin folder

1.1.6 New language features

Below is a list of some of the most visible new language features in the Core language of X#, compared to Visual Objects and Vulcan.

As you can see many new keywords were introduced, but these are positional: they will also be recognized as Identifiers on other places, so there is very little chance that you will have to make changes to avoid naming conflicts.

FEATURE	Description
DEFINE <id> := <Expression>	The VO Define is back again in X#. It will be compiled into a constant of the Globals class, the same class in which all Functions and Methods are implemented. The biggest advantage of a DEFINE over the preprocessor DEFINEs in Vulcan.NET is that there is no longer a chance that a DEFINE with the same name as a Method, Property or Variable will lead to incomprehensible compiler errors.
USING STATIC <Name>	The STATIC modifier for USING (note that the # sign is no longer needed) allows you to name a static class. When you do so you can then use the methods of this class as functions. For example: ``` USING STATIC System.Console FUNCTION Start as VOID WriteLine("X# is cool!") RETURN ```
BEGIN USING <Var> <Statements> END [USING]	The USING block allows you to control the lifetime of a variable. If <Var> has a destructor then it will be automatically destructed once the block has finished
SWITCH <Expression> CASE <Const> <Statements> CASE <Const2> CASE <Const3> <Statements> OTHERWISE <Statements> END [SWITCH]	The SWITCH statement generates a more efficient jump structure than the DO CASE command. Also the expression is only evaluated once.
BEGIN UNSAFE <Statements> END [UNSAFE]	Allows unsafe code in the context of this block , regardless of the compiler setting for the project as a whole.
BEGIN CHECKED <Statements> END [CHECKED] Also allowed as expression x := CHECKED(y)	The statements inside the block will have checked conversions, regardless of the compiler setting for the project as a whole.
BEGIN UNCHECKED <Statements> END [UNCHECKED]	The statements inside the block will have unchecked conversions, regardless of the compiler setting for the project as a whole.

FEATURE	Description
Also allowed as expression x := UNCHECKED(y)	
VAR <Identifier> := <Expression>	This is a synonym for LOCAL IMPLIED

CLASS <Id> < <ParamName> > WHERE <TypeConstraints> <Classmembers> END [CLASS]	Creating Generic classes is now supported in X#, with all the features that C# also has For example
	<pre>CLASS MyList<T> WHERE T IS CLASS
	..
	END CLASS</pre>or
	<pre>CLASS MyList<T> WHERE T IS ICustomer, NEW()
	..
	END CLASS</pre>

ASYNC - AWAIT	The ASYNC AWAIT infratructure is fully available inside X#
<Expression> IS <Type>	Allows to check an expression for a type. Can be used in stead of IsInstanceOf() and will perform better

Conditional Access Operator ?:
 <Expression> ? <Expression>
	Conditional access for properties, methods etc. For example
	<pre>nCount := MyList?:Count</pre>This translates to something like:
	<pre>VAR temp := MyList
	IF temp != NULL
	   nCount := temp:Count
	ENDIF</pre>The expression on the Left hand side of the Questionmark will be evaluated only once !

<Expression> DEFAULT <Expression>	The default operator allows you to inline a check for NULL:
	<pre>lResult := Foo() DEFAULT Bar()</pre>This translates to the same as
	<pre>lResult := Foo()
	IF lResult == NULL
	   lResult := Bar()
	ENDIF</pre>Foo() will be evaluated only once. And only when the result is NULL then Bar() will be evaluated.

y := CHECKED(x)	Tells the compiler to generate code that checks for overflow
y := UNCHECKED(x)	Tells the compiler to generate code that does NOT check for overflow
LINQ Query expressions are now supported: VAR CustQuery = FROM Cust in Customers ; WHERE Cust.City = "Athens" ; ORDER BY	The full LINQ feature set will be supported by X#: FROM LET WHERE JOIN ORDER BY EQUALS INTO

FEATURE	Description
Cust.Zipcode Select Cust	
YIELD RETURN <Value>	Can be used in a method declared as Enumerator of a type. This will instruct the compiler to automatically generate a class that implemented an enumerator and return to the calling code directly on the YIELD RETURN line. The next time the Iterator is called the code will remember where the code was the previous time it was executed and will continue on the next statement after the YIELD RETURN line.

1.1.6.1 Anonymous Methods

An example of an Anonymous Method Expression (AME) (note the DELEGATE keyword):
Note that the body of the that you can have :
1. A single Expression
2. An Expression List
3. A statement List
The first 2 require the expression(s) to be on the same line as the opening Curly **{** . Of course you can use the statement continuation character **;** to tell the compiler that you have spread the statement over more than one line.
The last one requires the statements in the list to be on separate lines and the closing Curly **}** must also be on a separate line. This is shown in the example below.

```
USING System.Windows.Forms
FUNCTION Start() AS VOID
    TestAnonymous()
    RETURN

FUNCTION TestAnonymous() AS VOID
    LOCAL oForm AS Form
    oForm := Form{}
    oForm:Text := "Click me to activate the anonymous method"
    oForm:Click += DELEGATE(o AS System.Object, e AS
System.EventArgs ) {
                System.Windows.Forms.MessageBox.Show("Click 1!")

                System.Windows.Forms.MessageBox.Show("Click 2!")

            }
    oForm:ShowDialog()
    RETURN
```

1.1.6.2 Anonymous Types

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first.
The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

The syntax for an anonymous class is :

```
VAR o := CLASS { Name := "test", Value := "something" }
```

In LINQ this may lead to:

```
VAR    result := from c in db.Customers where c.Orders.Count > 0 ;
   select CLASS{ ID := C.CustomerID, Name := C.CustomerName,
OrderCount := C.Orders.Count}
```

In this case the object will have the properties **ID**, **Name** and **OrderCount** (all explicitly given)

If you select named properties from another object you can omit the <Name> := part. In that case the compiler will simply use the same name:

```
VAR    result := from c in db.Customers where c.Orders.Count > 0 ;
   select CLASS{ C.CustomerID, C.CustomerName, OrderCount :=
C.Orders.Count}
```

In this case the anonymous class will have the properties **CustomerID**, **CustomerName** (inherited from C) and **OrderCount** (explicitly given)

Anonymous types contain one or more public read-only properties. No other kinds of class members, such as methods or events, are valid.
The expression that is used to initialize a property cannot be null, an anonymous function, or a pointer type.

The most common scenario is to initialize an anonymous type with properties from another type. In the following example, assume that a class exists that is named Product. Class Product includes Color and Price properties, together with other properties that you are not interested in. Variable products is a collection of Product objects. The anonymous type declaration starts with the new keyword. The declaration initializes a new type that uses only two properties from Product. This causes a smaller amount of data to be returned in the query.

If you do not specify member names in the anonymous type, the compiler gives the anonymous type members the same name as the property being used to initialize them. You must provide a name for a property that is being initialized with an expression, as

shown in the previous example. In the following example, the names of the properties of the anonymous type are Color and Price.

```
var productQuery := ;
    from prod in products ;
    select CLASS { prod:Color, prod:Price }

foreach var v in productQuery
    Console.WriteLine("Color={0}, Price={1}", v:Color, v:Price)
next
```

Typically, when you use an anonymous type to initialize a variable, you declare the variable as an implicitly typed local variable by using var. The type name cannot be specified in the variable declaration because only the compiler has access to the underlying name of the anonymous type.

Remarks

Anonymous types are class types that derive directly from object, and that cannot be cast to any type except object. The compiler provides a name for each anonymous type, although your application cannot access it. From the perspective of the common language runtime (CLR), an anonymous type is no different from any other reference type.

If two or more anonymous object initializers in an assembly specify a sequence of properties that are in the same order and that have the same names and types, the compiler treats the objects as instances of the same type. They share the same compiler-generated type information.

You cannot declare a field, a property, an event, or the return type of a method as having an anonymous type. Similarly, you cannot declare a formal parameter of a method, property, constructor, or indexer as having an anonymous type. To pass an anonymous type, or a collection that contains anonymous types, as an argument to a method, you can declare the parameter as type object. However, doing this defeats the purpose of strong typing. If you must store query results or pass them outside the method boundary, consider using an ordinary named struct or class instead of an anonymous type.

Because the Equals and GetHashCode methods on anonymous types are defined in terms of the Equals and GetHashCode methods of the properties, two instances of the same anonymous type are equal only if all their properties are equal.

1.1.6.3 ASTYPE

We have added a new keyword that allows you to cast a value to a type with type, just like the C# "as" keyword:

```
FUNCTION Test (oVar as ParentClass)
LOCAL oChild := oVar ASTYPE ChildClass
IF (oChild != NULL_OBJECT)
    DoSomething()
ENDIF
```

1.1.6.4 ASYNC - AWAIT

```
//
// This example shows that you can call an async task and wait for
it to finish
// The result of the async task (in this case the size of the file
that has been downloaded)
// will be come available when the task has finished
// The calling code (The Start()) function will not have to wait
until the async task has
// finished. That is why the line "2....." will be printed before
the results from TestClass.DoTest()
// The sample also shows an event and displays the thread id's.
You can see that the DownloadFileTaskAsync() method
// starts multiple threads to download the web document in
multiple pieces.

USING System
USING System.Threading.Tasks

FUNCTION Start() AS VOID
    ? "1. calling long process"
    TestClass.DoTest()
    ? "2. this should be printed while processing"
    Console.ReadKey()

CLASS TestClass
    STATIC PROTECT oLock AS OBJECT      // To make sure we
synchronize the writing to the screen
    STATIC CONSTRUCTOR
       oLock := OBJECT{}

    ASYNC STATIC METHOD DoTest() AS VOID
        LOCAL Size AS INT64
         Size := AWAIT LoooongProcess()
```

```
        ? "3. returned from long process"
        ? Size, " Bytes downloaded"

    ASYNC STATIC METHOD LoooongProcess() AS Task<INT64>
        VAR WebClient := System.Net.WebClient{}
        VAR FileName := System.IO.Path.GetTempPath()+"temp.txt"
        webClient:DownloadProgressChanged += OnDownloadProgress
        webClient:Credentials :=
System.Net.CredentialCache.DefaultNetworkCredentials
        AWAIT
webClient:DownloadFileTaskAsync("http://www.xsharp.info/index.php"
, FileName)
        VAR dirInfo      :=
System.IO.DirectoryInfo{System.IO.Path.GetTempPath()}
        VAR Files        := dirInfo:GetFiles("temp.txt")
        IF Files:Length > 0
            System.IO.File.Delete(FileName)
            RETURN Files[1]:Length
        ENDIF
        RETURN 0


    STATIC METHOD OnDownloadProgress (sender AS OBJECT, e AS
System.Net.DownloadProgressChangedEventArgs) AS VOID
        BEGIN LOCK oLock
          ? String.Format("{0,3} % Size: {1,8:N0} Thread {2}",
100*e:BytesReceived / e:TotalBytesToReceive , e:BytesReceived, ;
            System.Threading.Thread.CurrentThread:ManagedThreadId)
        END LOCK
        RETURN

    END CLASS
```

1.1.6.5 BEGIN CHECKED

```
FUNCTION Start() AS VOID
    LOCAL d AS DWORD
    LOCAL n AS INT

    d := UInt32.MaxValue
    ? "Initial value of d:", d

    BEGIN UNCHECKED
        // arithmetic operations inside an UNCHECKED block will not
produce
        // overflow exceptions on arithmetic conversions and
```

```
operations,
    // no matter if overflow checking is enabled application-
wide or not
    n := (INT)d
    ? "Value of n after conversion:", n
    d ++
    ? "Value of d after increasing it:", d
END UNCHECKED

d := UInt32.MaxValue
BEGIN CHECKED
    // arithmetic operations inside a CHECKED block always do
    // overflow checking and throw exceptions if overflow is
detected
    TRY
        n := (INT)d
        d ++
    CATCH e AS Exception
        ? "Exception thrown in CHECKED operation:", e:Message
    END TRY
END CHECKED
Console.ReadLine()
RETURN
```

1.1.6.6 BEGIN FIXED

The new FIXED modifier and BEGIN FIXED .. END FIXED keywords allow you to tell the .Net runtime that you do not want a variable to be moved by the Garbage collector.

```
UNSAFE FUNCTION Start AS VOID
    VAR s := "SDRS"
    BEGIN FIXED LOCAL p := s AS CHAR PTR
        VAR i := 0
        WHILE p[i] != 0
            p[i++]++
        END
    END FIXED
    Console.WriteLine(s)
    Console.Read()
    RETURN
```

As you can see the BEGIN FIXED statement requires a local variable declaration. The contents of this local (in the example above a CHAR PTR) will be excluded from garbage collection inside the block.

Please note:
The FIXED keyword and the example above should be used with extreme care. Strings in .Net are immutable. You normally should not manipulate strings this way !

X#

1.1.6.7   BEGIN UNCHECKED

```
FUNCTION Start() AS VOID
    LOCAL d AS DWORD
    LOCAL n AS INT

    d := UInt32.MaxValue
    ? "Initial value of d:", d

    BEGIN UNCHECKED
        // arithmetic operations inside an UNCHECKED block will not
produce
        // overflow exceptions on arithmetic conversions and
operations,
        // no matter if overflow checking is enabled application-
wide or not
        n := (INT)d
        ? "Value of n after conversion:", n
        d ++
        ? "Value of d after increasing it:", d
    END UNCHECKED

    d := UInt32.MaxValue
    BEGIN CHECKED
        // arithmetic operations inside a CHECKED block always do
        // overflow checking and throw exceptions if overflow is
detected
        TRY
            n := (INT)d
            d ++
        CATCH e AS Exception
            ? "Exception thrown in CHECKED operation:", e:Message
        END TRY
    END CHECKED
    Console.ReadLine()
RETURN
```

1.1.6.8   BEGIN UNSAFE

Enter topic text here.

1.1.6.9   BEGIN USING

```
//
// XSharp allows you to not only use the using statement to link
to namespaces
// You can also link to a static class and call the methods in
```

```
this class as if they are functions.
// The functions WriteLine and ReadKey() in the following code are
actually resolved as System.Console.WriteLine()
// and System.Console.ReadKey()
// Finally there is also the BEGIN USING .. END USING construct
which controls the lifetime of a variable
// At the end of the block the Variable will be automatically
disposed.
USING System
USING STATIC System.Console

FUNCTION Start() AS VOID
      WriteLine("Before Using Block")
   WriteLine("------------------")
   BEGIN USING VAR oTest := Test{}
     oTest:DoSomething()
   END USING
   WriteLine("------------------")
   WriteLine("After Using Block")
   ReadKey()


CLASS Test IMPLEMENTS IDisposable
   CONSTRUCTOR()
      Console.WriteLine("Test:Constructor()")

   METHOD DoSomething() AS VOID
      Console.WriteLine("Test:DoSomething()")

   METHOD Dispose() AS VOID
      Console.WriteLine("Test:Dispose()")

END CLASS
```

1.1.6.10 Collection Initializers

Collection initializers allow you to fill a collection with a list of values.
In the background the compiler will generate code that calls the Add method of the collection.
The example below creates a list of Integers, Strings and Persons.

```
USING System
USING System.Collections.Generic
USING System.Linq
USING System.Text
```

X#

```
    FUNCTION Start() AS VOID
        LOCAL oList AS List<Int>
        // The next line creates the collection and adds 5 elements
        // Note the double curly braces:
        // The first pair calls the default constructor of the
List<> Class
        // The second pair of curly braces surrounds the list of
values
        Console.WriteLine("Collection Initializers")
        oList := List<Int>{} {1,2,3,4,5}
        FOREACH VAR i IN oLIst
            Console.WriteLine(i)
        NEXT
        VAR oCompass := List<String>{}{"North", "East", "South",
"West"}
        FOREACH VAR sDirection in oCompass
            Console.WriteLine(sDirection)
        NEXT
        Console.ReadLine()
        // Now an example of an Object Initializer
        // Note that the object has no constructor
        // We are assigning the values directly to the properties
        // This will only work if there are public properties
        // Again there are double curly braces:
        // The first pair calls the default constructor of the
Person class
        // The second pair of curly braces surrounds the list of
name-value pairs

        Console.WriteLine("Object Initializer")
        VAR oPerson := Person{}{FirstName := "John", LastName :=
"Smith"}
        ? oPerson:Name
        Console.ReadLine()
        // Combine the two
        Var oPeople := List<Person> {} {;
                                Person{}{FirstName := "John",
LastName := "Smith"}, ;
                                Person{}{FirstName := "Jane",
LastName := "Doe"} ;
                            }
        Console.WriteLine("Collection and Object Initializers")

        FOREACH var oP in oPeople
            Console.WriteLine(oP:Name)
        NEXT
        Console.ReadLine()
        RETURN
```

```
PUBLIC CLASS Person
    PROPERTY FirstName  AS STRING AUTO
    PROPERTY LastName   AS STRING AUTO
    PROPERTY Name       AS STRING GET FirstName+" "+LastName
END CLASS
```

1.1.6.11 Conditional Access Expression

```
//
// This example shows various new expression formats
//
using System.Collections.Generic

Function Start() as void
    VAR oNone    := Person{"No", "Parent"}

    FOREACH VAR oValue in GetList()
        if oValue IS STRING                 // Value IS Type
            ? (String) oValue
        ELSEIF oValue IS INT
            ? (Int) oValue
        ELSEIF oValue IS DateTime
            ? (DateTime) oValue
        ELSEIF oValue IS Person
            LOCAL oPerson as Person
            oPerson := (Person) oValue
            ? oPerson:FirstName, oPerson:LastName
        // Value DEFAULT Value2 . When Value IS NULL then Value2
will be used
            oPerson := oPerson:Parent DEFAULT oNone
            ? "Parent: ", oPerson:FirstName, oPerson:LastName
        ENDIF
    NEXT
    LOCAL oEmptyPerson as Person
    LOCAL sName as STRING
    oEmptyPerson := GetAPerson()
    sName    := oEmptyPerson?:FirstName        // Conditional
Access: This will not crash, even when Person is a NULL_OBJECT
    ? sName DEFAULT "None"
    Console.ReadLine()
    RETURN

FUNCTION GetList() AS List<OBJECT>
    VAR aList := List<OBJECT>{}
```

```
        aList:Add(DateTime.Now)
        aList:Add("abcdefg")
        aList:Add(123456)
        VAR oPerson := Person{"John", "Doe"}
        aList:Add(oPerson)
        VAR oChild := Person{"Jane", "Doe"}
        oChild:Parent := oPerson
        aList:Add(oChild)
        RETURN aList


    CLASS Person
        EXPORT FirstName    AS STRING
        EXPORT LastName      as STRING
        EXPORT Parent        as Person
        CONSTRUCTOR(First as STRING, Last as STRING)
            FirstName   := First
            LastName    := Last


    END CLASS


    FUNCTION GetAPerson() as Person
        RETURN NULL_OBJECT
```

1.1.6.12 Creating Generic Classes

Stack Example
This example shows that we can now create generic classes with X# !
In the Stack class the T parameter will be replaced with a type at compile time.

```
    /*
    Stack Example - Written by Robert van der Hulst
    This example shows that we can now create generic classes with X#
    !
    Note: Compile with the /AZ option
    */


    USING System.Collections.Generic
    USING STATIC System.Console

    FUNCTION Start AS VOID
        LOCAL oStack AS Stack<INT>
        LOCAL i AS LONG
        TRY
        oStack := Stack<INT>{25}
            WriteLine("Created a stack with {0} items",oStack:Capacity)
            WriteLine("Pushing 10 items")
```

```
        FOR I := 1 TO 10
           oStack:Push(i)
        NEXT
        WriteLine("Popping the stack until it is empty")
    i := 0
    WHILE oStack:Size > 0
        i += 1
        WriteLine(oStack:Pop())
    END
    WriteLine("{0} Items popped from the stack",i)
    WriteLine("Press Enter")
    ReadLine()
    WriteLine("The next line pops from an empty stack and throws an
exception")
    ReadLine()
    WriteLine(oStack:Pop())
    CATCH e AS Exception
        WriteLine("An exception was catched: {0}", e:Message)
    END TRY
    WriteLine("Press Enter to Exit")
    ReadLine()
    RETURN

CLASS Stack<T>  WHERE T IS STRUCT, NEW()
    PROTECT _Items        AS T[]
    PROTECT _Size         AS INT
    PROTECT _Capacity     AS INT
    PROPERTY Size         AS INT GET _Size
    PROPERTY Capacity     AS INT GET _Capacity

    CONSTRUCTOR()
        SELF(100)

    CONSTRUCTOR(nCapacity AS INT)
        _Capacity := nCapacity
        _Items := T[]{nCapacity}
        _Size  := 0
        RETURN

    PUBLIC METHOD Push( item AS T) AS VOID
        IF _Size >= _Capacity
            THROW StackOverFlowException{}
        ENDIF
        _Items[_Size] := item
        _Size++
        RETURN

    PUBLIC METHOD Pop( ) AS T
```

```
            _Size--
            IF _Size >= 0
                RETURN _Items[_Size]
            ELSE
                _Size := 0
                THROW Exception{"Cannot pop from an empty stack"}
            ENDIF
    END CLASS
```

1.1.6.13 DEFAULT Expressions

```
    //
    // This example shows various new expression formats
    //
    using System.Collections.Generic

    Function Start() as void
        VAR oNone   := Person{"No", "Parent"}

        FOREACH VAR oValue in GetList()
            if oValue IS STRING                  // Value IS Type
                ? (String) oValue
            ELSEIF oValue IS INT
                ? (Int) oValue
            ELSEIF oValue IS DateTime
                ? (DateTime) oValue
            ELSEIF oValue IS Person
                LOCAL oPerson as Person
                oPerson := (Person) oValue
                ? oPerson:FirstName, oPerson:LastName
            // Value DEFAULT Value2 . When Value IS NULL then Value2
    will be used
                oPerson := oPerson:Parent DEFAULT oNone
                ? "Parent: ", oPerson:FirstName, oPerson:LastName
            ENDIF
        NEXT
        LOCAL oEmptyPerson as Person
        LOCAL sName as STRING
        oEmptyPerson := GetAPerson()
        sName   := oEmptyPerson?:FirstName        // Conditional
    Access: This will not crash, even when Person is a NULL_OBJECT
        ? sName DEFAULT "None"
        Console.ReadLine()
        RETURN
```

```
FUNCTION GetList() AS List<OBJECT>
   VAR aList := List<OBJECT>{}
   aList:Add(DateTime.Now)
   aList:Add("abcdefg")
   aList:Add(123456)
   VAR oPerson := Person{"John", "Doe"}
   aList:Add(oPerson)
   VAR oChild := Person{"Jane", "Doe"}
   oChild:Parent := oPerson
   aList:Add(oChild)
   RETURN aList

CLASS Person
   EXPORT FirstName   AS STRING
   EXPORT LastName     as STRING
   EXPORT Parent       as Person
   CONSTRUCTOR(First as STRING, Last as STRING)
      FirstName   := First
      LastName    := Last

END CLASS

FUNCTION GetAPerson() as Person
   RETURN NULL_OBJECT
```

1.1.6.14 EVENT (Add and Remove)

XSharp now supports an extended Event syntax. Both a single line syntax is supported as a multi line:

Old style syntax

```
[Attributes] [Modifiers] EVENT [<ExplicitInterface>.] <Id> AS
<Type>    // Old Style
```

Single Line syntax with explicit expressions

```
[Attributes] [Modifiers] EVENT [<ExplicitInterface>.] <Id> AS
<Type>   [ADD <ExpressionList] [REMOVE <ExpressionList]        //
Single Line
```

Multi line syntax with explicit expressions

```
[Attributes] [Modifiers] EVENT [<ExplicitInterface>.] <Id> AS
<Type>      // Multi line

ADD
    <Statements>
END [ADD]
REMOVE
    <Statements>
END [REMOVE]
END [EVENT]
```

Example

```
USING System.Collections.Generic
FUNCTION Start AS VOID
    LOCAL e AS EventsExample
    e := EventsExample{}
    e:Event1 += TestClass.DelegateMethod
    e:Event1 += TestClass.DelegateMethod
    e:Event1 -= TestClass.DelegateMethod    // added 2, removed 1,
should be called once
    e:Event2 += TestClass.DelegateMethod
    e:Event2 += TestClass.DelegateMethod
    e:Event2 -= TestClass.DelegateMethod // added 2, removed 1,
should be called once
    e:Event3 += TestClass.DelegateMethod
    e:RaiseEvent1("This is a test through a multi line event
definition")
    e:RaiseEvent2("This is a test through a single line event
definition")
    e:RaiseEvent3("This is a test through an old style event
definition")
    Console.WriteLine("Press a Key")
    Console.ReadLine()


DELEGATE EventHandler (s AS STRING) AS VOID

CLASS TestClass
    STATIC METHOD DelegateMethod(s AS STRING ) AS VOID
        Console.WriteLine( s)
```

```
END CLASS


CLASS EventsExample
    PRIVATE eventsTable AS Dictionary<STRING, System.Delegate>
    PRIVATE CONST sEvent1 := "Event1" AS STRING
    PRIVATE CONST sEvent2 := "Event2" AS STRING
    CONSTRUCTOR()
        eventsTable := Dictionary<STRING, System.Delegate>{}
        eventsTable:Add(sEvent1,NULL_OBJECT)
        eventsTable:Add(sEvent2,NULL_OBJECT)

    // Multiline definition
    EVENT Event1 AS EventHandler
        ADD
            BEGIN LOCK eventsTable
                eventsTable[sEvent1] := ((EventHandler)
eventsTable[sEvent1]) + value
            END LOCK
            Console.WriteLine(__ENTITY__ + " "+value:ToString())
        END
        REMOVE
            BEGIN LOCK eventsTable
                eventsTable[sEvent1] := ((EventHandler)
eventsTable[sEvent1]) - value
            END LOCK
            Console.WriteLine(__ENTITY__+ " "+value:ToString())
        END
    END EVENT


    // Single Line defintion on multilpe lines with semi colons,
for better reading !
    EVENT Event2 AS EventHandler  ;
        ADD     eventsTable[sEvent2] := ((EventHandler)
eventsTable[sEvent2]) + value ;
        REMOVE  eventsTable[sEvent2] := ((EventHandler)
eventsTable[sEvent2]) - value

    // Old style definition
    EVENT Event3 AS EventHandler

    METHOD RaiseEvent1(s AS STRING) AS VOID
        VAR handler := (EventHandler) eventsTable[sEvent1]
        IF handler != NULL
            handler(s)
        ENDIF
```

```
        METHOD RaiseEvent2(s AS STRING) AS VOID
            VAR handler := (EventHandler) eventsTable[sEvent2]
            IF handler != NULL
                handler(s)
            ENDIF

        METHOD RaiseEvent3(s AS STRING) AS VOID
            IF SELF:Event3 != NULL
                Event3(s)
            ENDIF
END CLASS
```

1.1.6.15 Expression IS Type

```
//
// This example shows various new expression formats
//
using System.Collections.Generic

Function Start() as void
    VAR oNone    := Person{"No", "Parent"}

    FOREACH VAR oValue in GetList()
        if oValue IS STRING                     // Value IS Type
            ? (String) oValue
        ELSEIF oValue IS INT
            ? (Int) oValue
        ELSEIF oValue IS DateTime
            ? (DateTime) oValue
        ELSEIF oValue IS Person
            LOCAL oPerson as Person
            oPerson := (Person) oValue
            ? oPerson:FirstName, oPerson:LastName
        // Value DEFAULT Value2 . When Value IS NULL then Value2
will be used
            oPerson := oPerson:Parent DEFAULT oNone
            ? "Parent: ", oPerson:FirstName, oPerson:LastName
        ENDIF
    NEXT
    LOCAL oEmptyPerson as Person
    LOCAL sName as STRING
    oEmptyPerson := GetAPerson()
    sName    := oEmptyPerson?:FirstName          // Conditional
Access: This will not crash, even when Person is a NULL_OBJECT
    ? sName DEFAULT "None"
```

```
        Console.ReadLine()
        RETURN

    FUNCTION GetList() AS List<OBJECT>
        VAR aList := List<OBJECT>{}
        aList:Add(DateTime.Now)
        aList:Add("abcdefg")
        aList:Add(123456)
        VAR oPerson := Person{"John", "Doe"}
        aList:Add(oPerson)
        VAR oChild := Person{"Jane", "Doe"}
        oChild:Parent := oPerson
        aList:Add(oChild)
        RETURN aList

    CLASS Person
        EXPORT FirstName   AS STRING
        EXPORT LastName      as STRING
        EXPORT Parent        as Person
        CONSTRUCTOR(First as STRING, Last as STRING)
            FirstName   := First
            LastName    := Last

    END CLASS

    FUNCTION GetAPerson() as Person
        RETURN NULL_OBJECT
```

1.1.6.16 Initializers

```
USING System
USING System.Collections.Generic
USING System.Linq
USING System.Text

    FUNCTION Start() AS VOID
        LOCAL oList AS List<Int>
        // The next line creates the collection and adds 5 elements
        // Note the double curly braces:
        // The first pair calls the default constructor of the
List<> Class
        // The second pair of curly braces surrounds the list of
values
        Console.WriteLine("Collection Initializers")
        oList := List<Int>{} {1,2,3,4,5}
```

```
        FOREACH VAR i IN oLIst
            Console.WriteLine(i)
        NEXT
        VAR oCompass := List<String>{}{"North", "East", "South",
"West"}
        FOREACH VAR sDirection in oCompass
            Console.WriteLine(sDirection)
        NEXT
        Console.ReadLine()
        // Now an example of an Object Initializer
        // Note that the object has no constructor
        // We are assigning the values directly to the properties
        // This will only work if there are public properties
        // Again there are double curly braces:
        // The first pair calls the default constructor of the
Person class
        // The second pair of curly braces surrounds the list of
name-value pairs

        Console.WriteLine("Object Initializer")
        VAR oPerson := Person{}{FirstName := "John", LastName :=
"Smith"}
        ? oPerson:Name
        Console.ReadLine()
        // Combine the two
        Var oPeople := List<Person> {} {;
                                Person{}{FirstName := "John",
LastName := "Smith"}, ;
                                Person{}{FirstName := "Jane",
LastName := "Doe"} ;
                            }
        Console.WriteLine("Collection and Object Initializers")

        FOREACH var oP in oPeople
            Console.WriteLine(oP:Name)
        NEXT
        Console.ReadLine()
        RETURN

PUBLIC CLASS Person
    PROPERTY FirstName  AS STRING AUTO
    PROPERTY LastName   AS STRING AUTO
    PROPERTY Name       AS STRING GET FirstName+" "+LastName
END CLASS
```

1.1.6.17 Interpolated Strings

Interpolated strings is a feature that allows you to embed local variables, instance variables or other expressions inside literal strings.
X# supports two kinds of interpolated strings:

1. Normal Interpolated strings: i"...."

This works like a normal X# string but with an embedded expression:

```
FUNCTION Start AS VOID
    LOCAL Who AS STRING
    Who := "World"
    Console.Writeline( i"Hello {Who}")
    Console.Read()
    RETURN
```

2. Extended Interpolated strings: ie"..." and ei"...."

This is a combination of an interpolated string and an extended string. In the example below the \t will be replaced with a tab character.

```
FUNCTION Start AS VOID
    LOCAL Who AS STRING
    Who := "World"
    Console.Writeline( ie"Hello\t{Who}")
    Console.Read()
    RETURN
```

Notes

The expression parsing inside the interpolated strings recognizes:
- SELF:
- Local variables, Member variables and Properties with SELF: prefix and without this prefix
- Other expressions must be in C# syntax for now, using the dot (**.**) operator as send operator.

The expression elements inside the string can use formatting notation just like the String.Format() notation. For example:

```
    FUNCTION Start AS VOID
       LOCAL i AS INT
       i := 42
       Console.Writeline( i"Hello {i:x}")    // i is printed in hex
notation, so Hello 2a
       Console.Read()
       RETURN
```

1.1.6.18 LINQ Query Expressions

The following example shows a couple of LINQ Queries in X#

```
//references:
//System.dll
//System.Core.dll
//System.Linq.dll
USING System.Collections.Generic
USING System.Linq
USING STATIC System.Console

FUNCTION Start AS VOID
   VAR oDev   := GetDevelopers()
   VAR oC     := GetCountries()
   VAR oAll   := FROM D        IN oDev  ;
                 JOIN C        IN oC ON D:Country EQUALS C:Name
;
                 ORDERBY D:LastName  ;
                 SELECT CLASS {D:Name, D:Country,
C:Region}       // Anonymous class !
   // The type of oAll is
IOrderedEnumerable<<>f__AnonymousType0<Developer,Country>>
   // We prefer the VAR keyword!

   VAR oGreek := FROM Developer IN oDev ;
           WHERE Developer:Country == "Greece" ;
           ORDERBY Developer:LastName DESCENDING ;
           SELECT Developer
   // The type of oGreek is IOrderedEnumerable<Developer>
   // We prefer the VAR keyword!

   VAR oCount :=    FROM Developer IN oDev ;
           GROUP Developer BY Developer:Country INTO NewGroup ;
           ORDERBY NewGroup:Key SELECT NewGroup
   // The type of oCount is
IOrderedEnumerable<<IGrouping<string,Developer>>
   // We prefer the VAR keyword!
```

```
      WriteLine(e"X# does LINQ!\n")
      WriteLine(e"All X# developers (country+lastname order)\n")
      FOREACH VAR oDeveloper IN oAll
         WriteLine("{0} in {1}, {2}",oDeveloper:Name,
   oDeveloper:Country, oDeveloper:Region)
      NEXT

      WriteLine(e"\nGreek X# Developers (descending lastname)\n")
      FOREACH oDeveloper AS Developer IN oGreek
         WriteLine(oDeveloper:Name)
      NEXT

      WriteLine(e"\nDevelopers grouped per country\n")

      FOREACH VAR country IN oCount
         WriteLine(i"{country.Key}, {country.Count()} developer(s)")
         FOREACH VAR oDeveloper IN country
            WriteLine("  " + oDeveloper:Name)
         NEXT
      NEXT
      WriteLine("Enter to continue")
      ReadLine()
      RETURN


   FUNCTION GetDevelopers AS IList<Developer>
      // This function uses a collection initializer for the List of
   Developers
      // and Object initializers for the Developer Objects
      VAR oList := List<Developer>{}    { ;
                             Developer{}{ FirstName  := "Chris",
   LastName := "Pyrgas", Country := "Greece"},;
                             Developer{}{ FirstName  := "Robert",
   LastName := "van der Hulst", Country := "The Netherlands"},;
                             Developer{}{ FirstName  := "Fabrice",
   LastName := "Foray", Country := "France"},;
                             Developer{}{ FirstName  := "Nikos",
   LastName := "Kokkalis", Country := "Greece"} ;
                          }
      RETURN oList

   FUNCTION GetCountries AS IList<Country>
      // This function uses a collection initializer for the List of
   Counties
      // and Object initializers for the Country Objects
      VAR oList := List<Country>{}{ ;
                          Country{} {Name := "Greece",       Region
```

```
                    := "South East Europe"},;
                                    Country{} {Name := "France",         Region
    := "West Europe"},;
                                    Country{} {Name := "The Netherlands",
    Region := "North West Europe"} ;
                        }
       RETURN oList

CLASS Developer
   PROPERTY Name        AS STRING GET FirstName + " " + LastName
   PROPERTY FirstName   AS STRING AUTO
   PROPERTY LastName    AS STRING AUTO
   PROPERTY Country     AS STRING AUTO
END CLASS

CLASS Country
   PROPERTY Name        AS STRING AUTO
   PROPERTY Region      AS STRING AUTO
END CLASS
```

1.1.6.19 NOP

The NOP keyword allows you to define a line of code that does nothing but satisfies the compiler, so it will not complain about missing code.

```
// The NOP keyword is an empty statement.
// This tells the compiler that there is no code missing !
FUNCTION Start() AS VOID
LOCAL i as LONG
FOR i := 1 to 10
   IF I % 2 == 0
      Console.WriteLine(i)
   ELSE
      NOP        // Nothing happens here. This tells the compiler
that there is no code missing !
   ENDIF
NEXT
RETURN
```

1.1.6.20 Object Initializers

Object Initializers allow you to instantiate an object and assign values to its properties in one line of code.

The example below uses object initializers to set the FirstName and LastName property of the Person object

```
USING System
USING System.Collections.Generic
USING System.Linq
USING System.Text

   FUNCTION Start() AS VOID
      LOCAL oList AS List<Int>
      // The next line creates the collection and adds 5 elements
      // Note the double curly braces:
      // The first pair calls the default constructor of the
List<> Class
      // The second pair of curly braces surrounds the list of
values
      Console.WriteLine("Collection Initializers")
      oList := List<Int>{} {1,2,3,4,5}
      FOREACH VAR i IN oLIst
         Console.WriteLine(i)
      NEXT
      VAR oCompass := List<String>{}{"North", "East", "South",
"West"}
      FOREACH VAR sDirection in oCompass
         Console.WriteLine(sDirection)
      NEXT
      Console.ReadLine()
      // Now an example of an Object Initializer
      // Note that the object has no constructor
      // We are assigning the values directly to the properties
      // This will only work if there are public properties
      // Again there are double curly braces:
      // The first pair calls the default constructor of the
Person class
      // The second pair of curly braces surrounds the list of
name-value pairs

      Console.WriteLine("Object Initializer")
      VAR oPerson := Person{}{FirstName := "John", LastName :=
"Smith"}
      ? oPerson:Name
      Console.ReadLine()
      // Combine the two
      Var oPeople := List<Person> {} {;
```

X#

```
                                          Person{}{FirstName := "John",
        LastName := "Smith"}, ;
                                          Person{}{FirstName := "Jane",
        LastName := "Doe"} ;
                          }
          Console.WriteLine("Collection and Object Initializers")

          FOREACH var oP in oPeople
              Console.WriteLine(oP:Name)
          NEXT
          Console.ReadLine()
          RETURN

PUBLIC CLASS Person
    PROPERTY FirstName  AS STRING AUTO
    PROPERTY LastName   AS STRING AUTO
    PROPERTY Name       AS STRING GET FirstName+" "+LastName
END CLASS
```

1.1.6.21 SWITCH

```
//
// The SWITCH statement is a replacement for the DO CASE statement
// The biggest difference is that the expression (in this case
sDeveloper) is only evaluated once.
// which will have a performance benefit over the DO CASE
statement
// Empty statement lists for a CASE are allowed. In that case the
labels share the code (see CHRIS and NIKOS below)
//
// Please note that EXIT statements inside a switch are not
allowed, however RETURN, LOOP and THROW are allowed.
using System.Collections.Generic

Function Start() as void
   FOREACH VAR sDeveloper in GetDevelopers()
     SWITCH sDeveloper:ToUpper()
     CASE "FABRICE"
        ? sDeveloper, "France"
     CASE "CHRIS"
     CASE "NIKOS"
        ? sDeveloper, "Greece"
     CASE "ROBERT"
        ? sDeveloper, "The Netherlands"
     OTHERWISE
```

```
            ? sDeveloper, "Earth"
        END SWITCH
    NEXT
      Console.ReadKey()
    RETURN


FUNCTION GetDevelopers as List<String>
VAR aList := List<String>{}
aList:AddRange(<string>{ "Chris", "Fabrice", "Nikos", "Robert",
"YourName" } )
RETURN aList
```

1.1.6.22 USING

```
//
// XSharp allows you to not only use the using statement to link
to namespaces
// You can also link to a static class and call the methods in
this class as if they are functions.
// The functions WriteLine and ReadKey() in the following code are
actually resolved as System.Console.WriteLine()
// and System.Console.ReadKey()
// Finally there is also the BEGIN USING .. END USING construct
which controls the lifetime of a variable
// At the end of the block the Variable will be automatically
disposed.
USING System
USING STATIC System.Console

FUNCTION Start() AS VOID
    WriteLine("Before Using Block")
    WriteLine("------------------")
    BEGIN USING VAR oTest := Test{}
       oTest:DoSomething()
    END USING
    WriteLine("------------------")
    WriteLine("After Using Block")
    ReadKey()


CLASS Test IMPLEMENTS IDisposable
    CONSTRUCTOR()
        Console.WriteLine("Test:Constructor()")
```

```
    METHOD DoSomething() AS VOID
        Console.WriteLine("Test:DoSomething()")

    METHOD Dispose() AS VOID
        Console.WriteLine("Test:Dispose()")

END CLASS
```

1.1.6.23 VAR

```
//
// The VAR keyword has been added to the language because in many
situations
// the result of an expression will be directly assigned to a
local, and the expression
// will already describe the type of the variable
// VAR is a synonym for LOCAL IMPLIED
using System.Collections.Generic

FUNCTION Start AS VOID
// In the next line the compiler "knows" that today is a DateTime
VAR today := System.DateTime.Now
? today

// In the next line the compiler "knows" that text is a String
VAR text  := Convert.ToString(123)
? text

// In the next line the compiler "knows" that s is a string
FOREACH VAR s in GetList()
    ? s
NEXT

Console.ReadLine()

RETURN

FUNCTION GetList AS List<String>
VAR aList := List<String>{}
aList:Add("abc")
aList:Add("def")
aList:Add("ghi")
return aList
```

1.1.6.24 Xbase++ class declarations

As of build 2.0.0.8 X# also support Xbase++ style class declarations. Of course we have added strong typing to the language definition to make the code run faster.
Full documentation will be included in one of the next builds.
The code below shows the Xbase++ support in action

```
CLASS Developer
    class var list    as array      // class vars are like static
variables.
    class VAR nextid  as int
    class var random  as Random
PROTECTED:
    VAR id as int NOSAVE            // the Nosave clause will mark
the field with the [NonSerialized] attribute
EXPORTED:
    VAR FirstName     as string
    VAR LastName      as string
    VAR Country       as string
    // The inline prefix in the next line is needed when declaring
a method inside the class .. endclass block.
    INLINE METHOD initClass()      // The Xbase++ equivalent of the
Static constructor
        list := {}
        nextid := 1
        random := Random{}
        RETURN

    INLINE METHOD Init(cFirst as string , cLast as string, cCountry
as string)   // Init is the constructor
        // you can use :: instead of SELF:
        ::FirstName := cFirst
        ::LastName := cLast
        ::Country  := cCountry
        ::id := nextid
        nextid += 1
        aadd(list, SELF)
        RETURN

    INLINE Method SayHello() as STRING
        if ::Age < 40
            RETURN "Hello, I am " + ::FirstName+ " from "+::Country
        else
            RETURN "Hello, I am mr " + ::LastName+ " from
"+::Country+" but you can call me "+::FirstName
        endif

    INLINE METHOD FullName() as string
```

```
        RETURN ::FirstName + " " + ::LastName

   INLINE METHOD Fire() as logic
     local nPos as dword
     nPos := Ascan(list, SELF)
     if nPos > 0
         aDel(list, nPos)
         ASize(list, aLen(list)-1)
         return true
     endif
     return false
     // the next block contains forward declarations. The methods
has to be declared below the class .. endclass block

   // SYNC method makes sure that only one thread can run this
code at the same time
   SYNC METHOD LivesIn
   // ACCESS indicates a PROPERTY GET
   // ASSIGN indicates a PROPERTY SET
   ACCESS CLASS METHOD Length as dword
   ACCESS METHOD Age as int
ENDCLASS

// This is the implementation of LivesIn. SYNC does not have to be
repeated here
METHOD LivesIn(cCountry as string) as logic
   return lower(::Country) == lower(cCountry)

// This is the implementation of the AGE Property. ACCESS does not
have to be repeated here
METHOD Age() as int
   local nAge as int
   nAge  := random:@@Next(25,60)
   return nAge

// This is the implementation of the Length Property. ACCESS does
not have to be repeated. CLASS however must be specified.
CLASS METHOD Length as dwor
   return ALen(list)

// the entry point is Main() like in Xbase++
function Main(a) as int
  local oDeveloper  as Developer
  local aDevs := {} as array
  local i as int
  if PCount() > 0
     ? "Parameters"
     for i := 1 to PCount()
```

```
         ? _GetFParam(i)
       next
     endif
   // create a new object with Xbase++ Syntax. Developer{} would
   have worked as well.
     aadd(aDevs, Developer():New("Chris", "Pyrgas", "Greece"))
     aadd(aDevs, Developer():New("Nikos", "Kokkalis","Greece"))
     aadd(aDevs, Developer():New("Fabrice", "Foray","France"))
     aadd(aDevs, Developer():New("Robert", "van der Hulst","The
   Netherlands"))
     ? "# of devs before", Developer.Length
     for i := 1 to alen(aDevs)
       oDeveloper := aDevs[i]
       ? "Fields", oDeveloper:FirstName, oDeveloper:LastName,
   oDeveloper:Country
       ? "FullName",oDeveloper:FullName()
       ? oDeveloper:SayHello()
       ? "Greece ?", oDeveloper:LivesIn("Greece")
       ? "Fired", oDeveloper:Fire()
       ? "# of devs after firing", oDeveloper:FirstName,
   Developer.Length
     next
     _wait()
   RETURN  PCount()
```

1.1.6.25 YIELD

```
   using System.Collections.Generic

   // The Yield return statement allows you to create code that
   returns a
   // collection of values without having to create the collection in
   memory first.
   // The compiler will create code that "remembers" where you were
   inside the
   // loop and returns to that spot.
   FUNCTION Start AS VOID
     FOREACH nYear AS INT IN GetAllLeapYears(1896, 2040)
       ? "Year", nYear, "is a leap year."
     NEXT
     Console.ReadLine()
   RETURN

   FUNCTION GetAllLeapYears(nMin AS INT, nMax AS INT) AS
```

```
IEnumerable<INT>
   FOR LOCAL nYear := nMin AS INT UPTO nMax
      IF nYear % 4 == 0 .and. (nYear % 100 != 0 .or. nYear % 400
== 0)
         YIELD RETURN nYear
      END IF
      IF nYear == 2012
         YIELD EXIT      // Exit the loop
      ENDIF
   NEXT
```

1.1.7   Licensing

XSharp is FREE Software. Of course that does not mean that you can do with it whatever you want. We have set a couple of rules that determine what you can and cannot do with our software, so our software is released under a license.

In an ideal world we would have published all of our source code and binaries under a very open license agreement, such as the Apache License, just like how the Roslyn code is published.

However the reality is that there are people on this world that would like to take our source code and binaries and release this with minimal changes under their own name.

To prevent that from happening we have been forced to set some limitations to how you can use our product.

For any "normal" developer that uses XSharp to develop applications for end users this should present no problems.

The simplified version of the limitations means that :

- You can only redistribute the 'runtime' components that are listed in the documentation. You can not redistribute our compiler. However when needed, you can ask your customer to download our compiler from this website
- You (when you are a FOX subscriber) have access to the compiler source code and may change and rebuild the compiler for internal use inside your company. However you are not allowed to publish or redistribute the (changed) compiler to your customers

We are convinced that you will find our license agreements fair and easy to work with. If you have any questions about the licenses, please contact us at info@xsharp.eu

Component & Name	Short summary of the license
The compiler binaries and runtime binaries XSharp Open Software License Agreement 1.0	This can be downloaded for free from our website. All you have to do is to create an account on this website. - You can download the components from our website - You can use the components for free both for commercial and personal use - You can only deploy the runtime components to your customers - OEM Licensing of the compiler binaries is only allowed when you have a separate agreement with XSharp BV
The source code to the runtime, visual studio integration and tools Apache License version 2.0	Everybody can access this public source code in our public repository on GitHub - You can download the source from our GitHub project - You can make adjustments to the source code both for internal use and also to redistribute the changed runtime binaries to your customers - If you have a contribution that you want to see included in a future version of XSharp you can send us a pull request on GitHub - XSharp BV decides which changes will be included in the main branch and which not.

The Roslyn part of the Compiler Source code [Apache License version 2.0](#)	This license applies to the Roslyn code that we have used and that you can also find on [GitHub](#)
The Antlr part of the Compiler Source code [Antlr BSD License](#)	This license applies to the Antlr 4 source code that we have used and that you can find on [GitHub](#)

1.1.7.1 XSharp Open Software License

The X# Compiler binaries and runtime binaries are licensed under the XSharp Open Software License Agreement. The full text of this license is:

```
XSHARP OPEN SOFTWARE LICENSE AGREEMENT

Version effective date: Sept 15, 2015

Preamble:

The use of the Software is unsupported and is for personal or
commercial use. Support is available from XSharp under a separate
agreement, see Part 3.c.
For redistribution of the Software, you will require an OEM license,
see part 4.b. For more information on support options or
redistribution (e.g. OEM Licensing) please contact XSharp.
This license establishes the terms under which the Software may be
used, copied, modified, distributed and/or redistributed. The intent
of this license is that XSharp maintains control over the
development and distribution of the Software, while allowing its use
it in a variety of ways. If the terms of this license do not permit
your proposed usage or if you require clarification regarding your
intended use of the Software, please contact info@xsharp.eu

XSharp BV. ("XSharp") is willing to license the software only upon
the condition that you accept all of the terms contained in this
software license agreement. Please read the terms carefully. By
clicking on "yes, accept" or by installing the software, you will
indicate your agreement with them. If you are entering into this
agreement on behalf of a company or other legal entity, your
acceptance represents that you have the authority to bind such
entity to these terms, in which case "you" or "your" shall refer to
your entity. If you do not agree with these terms, or if you do not
have the authority to bind your entity, then XSharp is unwilling to
license the software, and you should not install the software.

1. Parties.

The parties to this Agreement are you, the licensee ("You") and
XSharp. If You are not acting on behalf of yourself as an
individual, then "You" means your company or organization. A company
or organization shall in either case mean a single business entity,
and shall not include its affiliates or wholly owned subsidiaries.
```

2. The software.

The accompanying materials including, but not limited to, binary
executables, documentation, images, and scripts, which are
distributed by XSharp, and derivatives of that collection and/or
those files are referred to herein as the "Software".

3. License Grant for the Software.

a.   You are granted worldwide, perpetual, paid up, royalty free,
non-exclusive rights to install and use the Software subject to the
terms and conditions contained herein.

b.   You may: (i) copy the Software for archival purposes, (ii) copy
the Software for personal use purposes, (iii) use, copy, and
distribute the Software solely for your organization's internal use
and or business operation purposes including copying the Software to
other workstations inside Your organization. Any copy must contain
the original Software's proprietary notices in unaltered form.

c.   No Other Software and Services. XSharp will not provide you
with any other software or services (including any support or
maintenance services) relating to the Software, except to the extent
that such software and services, if any, are required and provided
pursuant to an applicable maintenance and support agreement.

4. Restrictions.

a.   XSharp encourages you to promote use of the Software. However,
this agreement does not grant permission to use the trade names,
trademarks, service marks, or product names of XSharp, except as
required for reasonable and customary use in describing the origin
of the Software. In particular, you cannot use any of these marks in
any way that might state or imply that XSharp endorses Your work, or
might state or imply that You created the Software covered by this
Agreement. Except as expressly provided herein, you may not:

i.   modify or translate the Software;

ii.   reverse engineer, decompile, or disassemble the Software,
except to the extent this restriction is expressly prohibited by
applicable law;

iii.   create derivative works based on the Software;

iv.   merge the Software with another product;

v.   copy the Software; or

vi.   remove or obscure any proprietary rights notices or labels on
the Software.

b.   You may not distribute the Software via OEM Distribution (as

defined below) without entering into a separate OEM Distribution Agreement with XSharp. "OEM Distribution" means permitting others outside Your organization to use the Software, distribution and/or use of the Software as either a bundled add-on to, or embedded component of, another application, with such application being made available to its users as, but not limited to, an on-premises application, a hosted application, a Software-as-a-Service offering or a subscription service for which the distributor of the application receives a license fee or any form of direct or indirect compensation. Except as expressly provided herein, you may not:

i.   permit others outside Your organization to use the Software,

ii.   redistribute:
        1. the Software as a whole whether as a wrapped application or on a stand-alone basis, or
        2. parts of the Software to create a language distribution, or
        3. the XSharp components with Your Wrapped Application.

The exception to this rule are the components of the software that are explicitly listed in the documentation as "redistributable files". These files are also copied by the installation process of the Software into to a separate folder with the name "Redist" under the Software's installation folder.

c.   You are excluded from the foregoing restrictions in paragraph 4b if You are using the Software for non-commercial purposes as determined by XSharp at its sole discretion, or if You are using the Software solely for Your organization's internal use and or internal business operation purposes on non-production servers (e.g. development and or testing).

5. Ownership.

XSharp and its suppliers own the Software and all intellectual property rights embodied therein, including copyrights and valuable trade secrets embodied in the Software's design and coding methodology. The Software is protected by the copyright laws from The Netherlands and international treaty provisions. This Agreement provides You only a limited use license, and no ownership of any intellectual property.

6. Infringement Indemnification.

You shall defend or settle, at Your expense, any action brought against XSharp based upon the claim that any modifications to the Software or combination of the Software with products infringes or violates any third party right; provided, however, that: (i) XSharp shall notify Licensee promptly in writing of any such claim; (ii) XSharp shall not enter into any settlement or compromise any such claim without Your prior written consent; (iii) You shall have sole control of any such action and settlement negotiations; and (iv) XSharp shall provide You with commercially reasonable information

and assistance, at Your request and expense, necessary to settle or defend such claim. You agree to pay all damages and costs finally awarded against XSharp attributable to such claim.

7. Limited Warranty.

Neither XSharp nor any of its suppliers or resellers makes any warranty of any kind, express or implied, and XSharp and its suppliers specifically disclaim the implied warranties of title, non-infringement, merchantability, fitness for a particular purpose, system integration, and data accuracy. There is no warranty or guarantee that the operation of the software will be uninterrupted, error-free, or virus-free, or that the software will meet any particular criteria of performance, quality, accuracy, purpose, or need. You assume the entire risk of selection, installation, and use of the software. This disclaimer of warranty constitutes an essential part of this agreement. No use of the software is authorized hereunder except under this disclaimer.

8. Local Law.

If implied warranties may not be disclaimed under applicable law, then any implied warranties are limited in duration to the period required by applicable law. Some jurisdictions do not allow limitations on how long an implied warranty may last, so the above limitations may not apply to You. This warranty gives you specific rights, and You may have other rights which vary from jurisdiction to jurisdiction.

9. Limitation of Liability.

Independent of the forgoing provisions, in no event and under no legal theory, including without limitation, tort, contract, or strict products liability, shall XSharp or any of its suppliers be liable to you or any other person for any indirect, special, incidental, or consequential damages of any kind, including without limitation, damages for loss of goodwill, work stoppage, computer malfunction, or any other kind of commercial damage, even if XSharp has been advised of the possibility of such damages. This limitation shall not apply to liability for death or personal injury to the extent prohibited by applicable law. In no event shall XSharp's liability for damages for any cause whatsoever, and regardless of the form of action, exceed in the aggregate the amount of the purchase price paid for the software license.

10. Export Controls.

You agree to comply with all export laws and restrictions and regulations of The Netherlands, The European Union or foreign agencies or authorities, and not to export or re-export the Software or any direct product thereof in violation of any such restrictions, laws or regulations, or without all necessary approvals. As applicable, each party shall obtain and bear all expenses relating

to any necessary licenses and/or exemptions with respect to its own export of the Software from The Netherlands or the European Union.

11. Severability.

If any provision of this Agreement is declared invalid or unenforceable, such provision shall be deemed modified to the extent necessary and possible to render it valid and enforceable. In any event, the unenforceability or invalidity of any provision shall not affect any other provision of this Agreement, and this Agreement shall continue in full force and effect, and be construed and enforced, as if such provision had not been included, or had been modified as above provided, as the case may be.

12. Jurisdiction and Venue.

This Agreement is governed by the applicable laws of The Netherlands.

13. Assignment.

Except as expressly provided herein neither this Agreement nor any rights granted hereunder, nor the use of any of the Software may be assigned, or otherwise transferred, in whole or in part, by Licensee, without the prior written consent of XSharp. XSharp may assign this Agreement in the event of a merger or sale of all or substantially all of the stock of assets of XSharp without the consent of Licensee. Any attempted assignment will be void and of no effect unless permitted by the foregoing. This Agreement shall inure to the benefit of the parties permitted successors and assigns.

1.1.7.2 Apache 2

Both the source code to the public elements of X# and to Roslyn are published under the Apache 2.0 license.
You can find the public source code on Github in the XSharpPublic repository :
https://github.com/X-Sharp/XSharpPublic
The full text of this license is:


```
Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction,
and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by
```

the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all
other entities that control, are controlled by, or are under common
control with that entity. For the purposes of this definition,
"control" means (i) the power, direct or indirect, to cause the
direction or management of such entity, whether by contract or
otherwise, or (ii) ownership of fifty percent (50%) or more of the
outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity
exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications,
including but not limited to software source code, documentation
source, and configuration files.

"Object" form shall mean any form resulting from mechanical
transformation or translation of a Source form, including but
not limited to compiled object code, generated documentation,
and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or
Object form, made available under the License, as indicated by a
copyright notice that is included in or attached to the work
(an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object
form, that is based on (or derived from) the Work and for which the
editorial revisions, annotations, elaborations, or other modifications
represent, as a whole, an original work of authorship. For the purposes
of this License, Derivative Works shall not include works that remain
separable from, or merely link (or bind by name) to the interfaces of,
the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside

or as an addendum to the NOTICE text from the Work, provided
that such additional attribution notices cannot be construed
as modifying the License.

You may add Your own copyright statement to Your modifications and
may provide additional or different license terms and conditions
for use, reproduction, or distribution of Your modifications, or
for any such Derivative Works as a whole, provided Your use,
reproduction, and distribution of the Work otherwise complies with
the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
any Contribution intentionally submitted for inclusion in the Work
by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify
the terms of any separate license agreement you may have executed
with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
names, trademarks, service marks, or product names of the Licensor,
except as required for reasonable and customary use in describing the
origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
agreed to in writing, Licensor provides the Work (and each
Contributor provides its Contributions) on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied, including, without limitation, any warranties or conditions
of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
PARTICULAR PURPOSE. You are solely responsible for determining the
appropriateness of using or redistributing the Work and assume any
risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
whether in tort (including negligence), contract, or otherwise,
unless required by applicable law (such as deliberate and grossly
negligent acts) or agreed to in writing, shall any Contributor be
liable to You for damages, including any direct, indirect, special,
incidental, or consequential damages of any character arising as a
result of this License or out of the use or inability to use the
Work (including but not limited to damages for loss of goodwill,
work stoppage, computer failure or malfunction, or any and all
other commercial damages or losses), even if such Contributor
has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
the Work or Derivative Works thereof, You may choose to offer,
and charge a fee for, acceptance of support, warranty, indemnity,
or other liability obligations and/or rights consistent with this
License. However, in accepting such obligations, You may act only
on Your own behalf and on Your sole responsibility, not on behalf
of any other Contributor, and only if You agree to indemnify,

```
defend, and hold each Contributor harmless for any liability
incurred by, or claims asserted against, such Contributor by reason
of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following
boilerplate notice, with the fields enclosed by brackets "[]"
replaced with your own identifying information. (Don't include
the brackets!) The text should be enclosed in the appropriate
comment syntax for the file format. We also recommend that a
file or class name and description of purpose be included on the
same "printed page" as the copyright notice for easier
identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
See the License for the specific language governing permissions and
limitations under the License.
```

1.1.7.3   BSD

The Antlr source that we have used in our product is published under the BSD license. The full text of this license is:

```
[The "BSD license"]
Copyright (c) 2015 Terence Parr, Sam Harwell
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
 notice, this list of conditions and the following disclaimer.
 2. Redistributions in binary form must reproduce the above copyright
 notice, this list of conditions and the following disclaimer in the
 documentation and/or other materials provided with the distribution.
 3. The name of the author may not be used to endorse or promote
products
 derived from this software without specific prior written permission.
```

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.1.8   Acknowledgements

We could not have done this project without (source code) contributions from others. Below is a list of the contributions that we used for our compiler and Visual Studio integration

Name	Description
Roslyn	The X# compiler is based on the open source project Roslyn, which contains the source to the Microsoft VB and C# compilers. Our project would have been impossible with this source. We are very grateful that Microsoft has moved this project to the open source. (github.com/dotnet/roslyn) The source code to Roslyn is published under the Apache License, version 2
Antlr	The X# compiler uses Antlr 4 as parser generator to generate the front end of the compiler. We would like to thank Terence Parr and Sam Harwell for their excellent work. The source code to Antlr is published under the BSD License. (www.antlr.org)
Wix (votive)	The project system for Wix (Votive). We have been inspired by some of the code in this system. We would like to thank Rob Mensching and the other authors of this project system. (wixtoolset.org/)
Nemerle	The project system for Nemerle. We have been inspired by some of the solutions in their project system. We would like to thank the authors of Nemerle for showing us how to solve certain problems. (www.nemerle.org)
F# project system	The visual F# project system (www.fsharp.org)
Help and Manual	Our documentation is written with the excellent tool "Help and Manual" from Alexander Halser (www.helpandmanual.com)
Sandcastle Help File Builder	The documentation for the runtime is generated with a special version of the Sandcastle Help File Builder product. (github.com/EWSoftware/SHFB)
Paul Piko	For his Hybrid classes that allow to mix VO GUI with Windows Forms
JB Evain	We are using his Mono.Cecil library inside our project system to collect type information for external types
The SQLite team	We are using their SQLite database inside our project system
Karl-Heinz Rauscher (Germany) António Lopes (Portugal)	Have contributed functions for the FoxPro library.

1.2 Version History

Note: When an item has a matching GitHub ticket then the ticket number is behind the item in parentheses prefixed with #. You can find these tickets by going to: https://github.com/X-Sharp/XSharpPublic/issues/**nnn** where **nnn** is the ticket number.
If you find an issue in X# we recommend that you report it on GitHub. You will be notified of the progress on the work on your issue.

Changes in 2.20.0.3

Compiler

Bug fixes

- Fixed problem with the USE command with an AGAIN clause in the FoxPro dialect (#235)
- Fixed problem with calling typed array constructors with named parameters when compiling with the /namedargs compiler option enabled (#1430)
- Fixed inconsistency with the INSTANCE keyword and the use inside the class (#1432)
- Fixed problem with the REPLACE UDC that could prevent the use of a variable named "replace" (#1443)
- Fixed problem with the /vo9 (handle missing RETURN statements) compiler option with ACCESSes in PARTIAL classes (#1450)
- Fixed problem with the Lexer recognizing line continuation characters inside a string in the FoxPro dialect (#1453)
- Fixed problem with the memvar pragma option (#1454)
- Fixed a problem with the /xpp compiler option. (#1243, #1458)
- Fixed a problem with accessing Hidden class members in a method from the class where the member was defined, when the object involved was untyped.(#1335, #1457)
- Fixed an internal compiler error with a line of code containing a single comma (#1462)
- Fixed a problem with the USE command when the filename was specified as a bracketed string (#1468)

New Features

- You can now use the NULL() and DEFAULT() expression to initialize any variable with a default value. This is the equivalent of the default keyword in C#.
- We have added a new compiler option /modernsyntax (#1394). This **disables** certain legacy features:
  - && for line comments
  - * at the start of a line for a comment line
  - Bracketed strings
  - Parenthesized expression lists (thus makes it easier to recognize tuples)
- Added support for IS NULL and IS NOT NULL pattern (#1422)
- Added support for file wide FIELD statements in the Harbour dialect (#1436)

Runtime

Bug fixes

- Fixed runtime error in Transform() with PTR argument (#1428)
- Fixed problem with several String runtime functions throwing a runtime error when passed a PSZ argument (#1429)
- Fixed problem with OrdKeyVal() and ADS/ADT files in the ADS RDD (#1434)
- Fixed incompatibilities with various xBase dialects with creating and using orders with long names (#1438)
- Fixed VO incompatibility in OrderKeyNo() with the ADS RDD when the setting Ax_SetExactKeyPos() is TRUE (#1444)
- Fixed a problem in the macro compiler with passing more than 2 arguments by reference (#1445)
- Fixed problem with DBSetIndex() seting the record pointer at eof (#1448)
- Fixed problem reading fields from OEM dbfs (#1449)

New Features

- Implemented the DBFMEMO driver (#604)
- Implemented the DBFBLOB driver (#605)
- Added missing SetColor() function overload with no parameters (#1440)
- This version includes the new XSharp.VFP.UI.DLL that is used by forms exported from Visual FoxPro with the VFP Exporter.

Visual Studio integration

Bug fixes

- Fixed a problem with "Jump to File" command in VS 2019 (#1146)
- Fixed problem with "Go to definition" not working for local function (#1415)
- Fixed problem with the Class navigation box showing the wrong current entry in some cases (#1426)
- Fixed problem with setting the "enable named arguments" project option (#1431)
- Fixed problem with the code generator for types in external assemblies not generating parameters for Indexed properties (#1442)
- Fixed problem with the VODBServer editor not saving access/assigns and other entities of the [DBSERVER] section in CAVOFED.TPL (#1452)
- Fixed problem with loading supplemental files provided in the cavowed.inf file for the VO Window Editor with absolute or relative paths (#1470)
- Fixed a problem in the VS2022 Debugger when different DLLs contained the same namespace with different case.
- Fixed a problem where the entity parser inside the editor did not correctly determine the end of an entity that contains a local function or procedure
- Fixed a problem where the entity parser inside the editor would choke on a param token at the start of the line when the /memvars compiler option was NOT enabled.

New features

- We have added a menu entry to the Help menu for the Chinese version of the documentation.

VOXporter

Bug fixes

- Fixed problem with incorrectly converting attributes to string literals (#1404)

New Features

- It is now possible to define special TEXTBLOCK entities in the VO code in any module with name "VXP-TOP" or "{VOXP:TOP}" and VOXporter will automatically insert the contents of the textblock in the beginning of the exported X# .prg file for the module. This is particularly helpful for specifying top level commands like #using statements (#1425)

VFPXporter

- This version of X# includes the [VFP Exporter](). This tool takes a Visual FoxPro project file and converts that into aVisual Studio solution

XIDE

- Added option when trying to debug a 32/64bit app in the wrong XIDE version, to automatically open the alternative version
- Fixed coloring of several positional keywords in the editor
- Improved editor support for TEXT...END TEXT
- Added editor support for the NOT NULL code pattern
- Added project support for the compiler options /namedargs, /initlocals, /modernsyntax and /allowoldstyleassignments
- Now pressing the SHIFT key on startup, resets the layout of the IDE to default positions (and does not save it on exit)
- Added menu command View->Save Current Layout
- Fixed a problem with toggling case (CTR+U) of text selected in a column selection
- Fixed several issues with incorrectly identifying a line with identifiers like PROC or FUNC as entity definitions

Documentation

Bug fixes

- Fixed typo in the /namedargs compiler option topic

New Features

- We have added several chapters about [modifiers]().
- We have added a (partially) translated help file in (Simplified) Chinese

Changes in 2.19.0.2

Compiler

Bug fixes

- Now the compiler properly reports an error when duplicate field names are defined in a type (#1385)
- Fixed problem with defining multiple type constraints in a generic type (#1389)
- Fixed problem with global MEMVARs hiding local variables or parameters with the same name (#1294)
- Bogus compiler error messages with not found type (#1396)
- Fixed compiler crash with missing reference to XSharp.VFP in the FoxPro dialect (#1405)
- Fixed problem with /initlocals compiler option incorrectly also initializing class fields (#1408)
- Fixed a problem in the preprocessor where an extended match symbol would not properly match an expression that started with a string literal

New features

- We added support for dimensioning (FoxPro) class properties, such as in

  ```
  DIMENSION this.Field(10)
  ```

- We have added support for FOREACH AWAIT, like in the following example. (Works in .Net Core, .Net 5 and later)

  ```
  FOREACH AWAIT VAR data IN GenerateNumbersAsync(number)
       SELF:oListView1:Items:Add(data)
  NEXT
  ```

- We have added support for "Coalescing Member Access, such as in the following example where FirstName and LastName are both properties of the oPerson object:

  ```
  ? oPerson:(FirstName+" "+LastName)
  ```

- The WITH command now also recognizes the AS DataType clause
- XBase++ Class declarations now also allow "END CLASS" as closing token.
- Now the compiler reports an error when attempting to convert from Lambda Expression to usual (#1343)
- We have added support for [TUPLE datatypes.](#) This includes declaring local variables, parameters, return value etc.
  We also support decomposition of a tuple return value into multiple locals. See the TUPLE help topic for more information.

Runtime

Bug fixes

- Fixed problem calling DoEvents() from the macro compiler (#872)
- Fixed problem with __Mem2StringRaw() (undocumented) function (#1302)
- Fixed problem opening DBFCDX index file with incorrect collation information in the header #1360
- Fixed problem with OrdSetFocus() resetting the current order when called without arguments (#1362)
- Fixed problems with some index files after a DBPack() (#1367)
- Fixed problem with Deleted() returning TRUE on a table with all records deleted (#1370)
- Fixed problem with opening and writing to a file with FWrite() etc functions when opened in exclusive and write only mode (#1382)
- Fixed several problems (VO incompatibilities) with the SplitPath() function (#1384)
- Now when there is no codepage found in a dbf header, then the DOS codepage from the RuntimeState is used and no longer the hardcoded codepage 437 (#1386)
- Replaced the Dictionary<,> class used in some areas of the runtime with ConcurrentDictionary<,> to avoid issues in multi threaded apps (#1391)
- Fixed problem with NolVarget when using IDynamicProperties (FoxPro dialect) (#1401)
- Fixed problem with Hex2C() giving different results with lower case letters than with upper case.
  Note that this bug existed also in VO, so now the behavior of Hex2C() with lower case hex letters in X# with is different to VO (#1402)
- Accessing properties on a closed DbServer object that was opened with the Advantage RDD could cause problems in the debugger. The DbServer class now returns empty values when the server is closed.

New features

- Implemented CREATE CURSOR command [FoxPro] (#247). Also implemented CREATE TABLE and ALTER TABLE (FoxPro dialect)
- Implemented INSERT INTO commands (FoxPro dialect for inserting variables from values, arrays, objects and memory variables. INSERT INTO from a SQL query does not work yet. (FoxPro dialect)
- Implemented new FoxPro-compatible version of Str() function in XSharp.VFP (#386)
- Now an error is thrown when opening an index file fails (#1358)
- Added AscA() function and made Asc() dependent on the SetAnsi() setting in the runtime (#1376)

Header files

Bug fixes

- Implemented several missing commands (#1407)
- Fixed typo in the SET DECIMALS TO command (#1406)
- Added missing clauses NAME and MEMVAR for the GATHER command (FoxPro) (#1409)

- Updated several commands to make some tokens optional and more compatible to various dialects (#1410, #1412)
- Fixed various incompatibilities with COMMIT command in various dialects (#1411)

Visual Studio Integration

Bug fixes

- Fixed problem with looking up public static field in type referenced by static using (#1307)
- Fixed intellisense problem with locals defined inside block statements (#1345)
- Fixed problem with intellisense incorrectly resolving type specified in code with full name to another from the usings list (#1363)
- Fixed problem with member completion incorrectly showing static methods after typing a colon (#1379)
- Fixed editor freezing with specific code (#1380)
- Fixed problem with Class Navigation bar not showing the method name in certain cases (#1381)

New features

- Added support for IEnumerable and DataTable Debugger visualizers (#1373).
  Please note that when browsing X# arrays the results in the visualizer are really ugly because the visualizer ignores attributes to hide properties and fields for our USUAL class.
- Adjusted the Globals, Workareas etc debugger windows to respect the global theme selected in VS (#1375). Also added status panel to the Workarea window, so you can see the workarea status or field names/values
- Added intellisense support for locals declared with USING VAR or USING (LOCAL) IMPLIED (#1390)
- Now the intellisense database uses an SQLite package that has ARM support, so the it will work also on a Mac and other platforms (#1397)

VOXporter

Fixes

- Fixed problem with VOXporter incorrectly modifying previously commented code with {VOXP:UNC} tags (#1404)

Documentation

Bug Fixes

- The documentation of functions in the runtime help was describing functions incorrectly.
  For example the topic title for the "Left" function was "Functions.Left Method" This has been changed to "Left Function"
- The "SingleLineEdit" class in the documentation was called "Real4LineEdit". This has been fixed.

New Features

- We have added additional documentation to the X# Programming guide about several subjects.

Changes in 2.18.0.4

Compiler

Bug fixes

- Fixed some preprocessor issues with XBase++ related commands (#1213, #1288, #1337)
- Fixed problem with implicit access to static class members (XBase++ dialect) (#1215)
- Fixed a parser error with the DIMENSION command (VFP dialect) (#1267)
- Fixed preprocessor problem with UDCs in code spanning in multiple lines (#1298)
- Now each "unused variable" warning is reported at the exact location of a variable definition, instread of always at the first one (#1310)
- Fixed bogus "unreachable code" warning in the SET RELATION command (#1312)
- Fixed a problem in generating XML documentation for the compiler generated <Module>.$AppInit and <Module>.$AppExit methods (#1316)
- Fixed problem with accessing hidden fields of another object (XBase++ dialect) (#1335)
- Fixed problem with calling parent methods with an explicit class indication (Xbase++ dialect) (#1338)
- Fixed problem with incorrectly calling function twice, in code like "SLen(c := SomeFunction())" (#1339)
- Fixed problem with parent Class methods not being visible in derived classes (Xbase++ dialect) (#1349)
- Fixed problem with ::new() not working properly in class methods (Xbase++ dialect) (#1350)
- Fixed an exception when an error occurred for a token that was defined in a header file
- Fixed a compiler error when returning super:Init() from a XBase++ method (#1357)
- Fixed a problem with STATIC DEFINEs in same named .prg files (#1361)

New features

- Introduced warning for not specifying the OUT keyword for OUT parameters (#1295)
- The parser rules for method and constructor calls without parameters have been updated. This may result in a bit faster compilation.
- SLen() is no longer "inlined" by the compiler. If you reference XSharp.Core in your app, SLen() now gets resolved to the SLen() function inside X# Core.
  If you compile without X# runtime, or compile against the Vulcan Runtime you now need to add a SLen() function to your code.
  This is the code inside X# Core that you can use as a template

```
FUNCTION SLen(cString AS STRING) AS DWORD
   LOCAL len := 0 AS DWORD
   IF cString != NULL
      len := (DWORD) cString:Length
```

```
        ENDIF
    RETURN len
```

- Added support for preprocessor commands #ycommand and #ytranslate that are also supported by Harbour. They work the same as #xcommand and #xtranslate, but the tokens are compared in case sensitive mode (#1314)
- Code generation for some of the Xbase++ specific features has changed.
- We have added several more UDCs with the IN <cursor> clause
- We have added UDC support for the FoxPro CAST expression
- Several more SET commands now also support the & operator
- The compiler now supports "Late bound names" in more locations, such as in the REPLACE command, With command etc. This now compiles without problems:

```
cVar := "FirstName"
WITH oCustomer
    .&cVar := "John"
END WITH
```
and this too

```
cVar := "FirstName"
REPLACE &cVar with "John"
```

Runtime

Bug fixes

- Fixed problem with incorrectly closing dbf file before relations are cleared (#1237)
- Fixed incorrect index scope visibility immediately after file creation (#1238)
- Fixed problem in FFirst()/FNext() not finding all files specified by filter (#1315)
- Fixed problem with DBSetIndex()/VoDbOrdListAdd() always reseting the controlling order to 1 (#1341)
- Fixed problem with updating index keys in the DBFCDX driver when the key expression was of type DATE.
- Fixed a problem when Str() and StrZero() had a built-in maximum string length of 30. (#1352)
- The RegisteredRDD Class now uses a ConcurrentDictionary.
- Fixed a bug in the RDD TransRec() method when a field is missing in the target table (#1372)
- Fixed a problem in the Advantage RDD to prevent ADS functions from being called when the table is closed
- Fixed a problem in the Advantage RDD that could occur when an field with an incorrect name was read
- Fixed a problem in the CurDir() function when the current directory is a UNCPath (\ \Server\Share\SomeDir) (#1378)

New features

- Added support for accessing indexers in the USUAL type (#1296)
- We have added a DbCurrency type that is returned from the RDD when a currency field is read.

- Implemented the TEXT TO FILE command (#1304)
- Now the RDD reports an error (dialog) when tagname > maximum length when creating an index order (#1305)
- Added a function _CreateInstance() that accepts a System.Type parameter
- The late binding code now detects from where Send(), IVarGet() and IVarPut() are called and allow access to private/hidden fields when the calling code is of the same type as the type where the class members were declared. This is used in some of the XBase++ related changes.
- The classes in the XBase++ have been restructured a bit.
- The mapping of several DBF / Workarea / Cursor related UDCs has been changed to be more FoxPro compatible.
- We have added runtime support for the FoxPro CAST expression
- We have done some small code optimizations w.r.t. dictionaries(#1371)
- Several DbServer properties no longer call into the RDD when the server is closed, but return blank values instead.

Typed SDK classes

- Added a DbServer:Append() overload without parametrs (#1320)
- Added missing DataServer:LockcurrentRecord() method (#1321)
- Fixed runtime error when creating a DataWindow with a ShellWindow as owner (#1324)
- Changed DataWindow:Show() method to CLIPPER for compatibility with existing code (#1325)
- Fixed exception when using a ComboBox on a VO Window (#1328)
- Fixed error when opening a datawindow with an assigned server (#1332)
- Fixed runtime error when instantiating a DBServer object with an untyped FileSpec object as first argument (#1348)
- Fixed problem with displaying items in Comboboxes and Listboxes (#1347)
- Several DbServer properties no longer call into the RDD when the server is closed, but return blank values instead.

Visual Studio Integration

Bug fixes

- Fixed problem with the "allow dot" setting in the project file (#1192)
- Several macros such as $CALLSTACK were not returning values in expected format. This has been fixed (#1236)
- Fixed build problem when there is a block comment in the first line of form.prg (#1334)
- Fixed probelm with block commenting a code snippet in a single line (#1336)
- Fixed failing project build when the project file contains a property <GenerateAssemblyInfo>True</GenerateAssemblyInfo> (#1344)
- Fixed a problem in the Parser that was causing errors parsing DebuggerDisplay attributes in the expression evaluator.
- The new debugger windows were not following the current windows theme. This is now partially fixed. (#1375)

VO Compatible Editors

- Fixed design time display issue with CheckBox and RadioButton captions with specific fonts in the VOWED (#796)
- Fixed problem with the VOWED editor changing all existing classes in the prg to PARTIAL (#814)
- Fixed problem with incorrectly adding constructor code to instantiate the DataBrowser in the VOWED, even when there are no (non-deleted) data columns (#1365)
- Fixed several problems in the VOMED with menu item define names in source code and resource files (#1374)

VOXporter

New features

- Introduced options (inline in existing code) to comment, uncomment and delete lines from the original VO code (#1303)
- {VOXP:COM} // comment out line
- {VOXP:UNC} // uncomment line
- {VOXP:DEL} and // {VOXP:REM} // remove line

Installer

New features

- The installer now detects if the required Visual Studio components "Core Editor" and ".Net Desktop Development" are installed.
  When it finds one or more VS installations but none of these installations has both the required components then a warning is shown.

Changes in 2.17.0.3

Compiler

Bug fixes

- Fixed several incompatibilities with XBase++ regarding using class members (#1215) **UNCONFIRMED**
- Fixed /vo3 option not working correctly in XBase++ dialect. Also added support for modifiers *final, introduce* and *override* (#1244)
- Fixed problem with using the NEW modifier on class fields (#1246)
- Fixed several preprocessor issues with XPP dialect UDCs (#1247, #1250)
- Fixed VO incompatibility with special handling of INSTANCE fields in methods and properties (#1253)
- Fixed problem with the debugger erratically stepping to incorrect lines (#1254, #1264)
- Fixed problem with showing the wrong error line number in some cases with nested statements (#1268)

- Fixed problem where a DO CASE statement without CASE lines was producing an internal error in the compiler (#1281)
- Fixed a couple preprocessor issues (#1284, #1289)
- Fixed missing compiler error on calling with SUPER a method that does not exist, when late binding is enabled (#1285)
- Fixed a Failed to emit Module error with CONST class field missing value assignment (#1293)
- Fixed a problem with repeated match markers (such as in the SET INDEX TO command) in the preprocessor.
- Fixed a problem that an property definition with an explicit interface prefix could lead to a compiler crash when the interface was "unknown" at compile time and/or the property name was not "Item" (#1306)

New features

- Added support for "classic" INIT PROCEDURE and EXIT PROCEDURE (#1290)
- Added warnings when statement list inside case blocks, if blocks and other blocks are empty. To suppress the warning you can add a NOP statement in your code.
- We have made some changes to the lexer and parser in the compiler. This may result in a bit smaller memory footprint and faster compilation speed for code with many nested blocks.

Runtime

Bug fixes

- Fixed several problems (incompatibilities with VO) in CToD() (#1275)
- Added support for 3rd parameter in AAdd() for specifying where to insert the new element (#1287)
- The Default() function now no longer updates usuals that have a value of NULL_OBJECT to be compatible with Visual Objects.(#1119)
- We have added support for parameters for the AdsSQLServer class (#1282)

Visual Studio integration

New Features

- We have added debugger pane windows for the following items:
  - Global variables
  - Dynamic memory variables (Privates and Publics)
  - Workareas
  - Settings
- You can open these windows from the Debug/XSharp menu during debugging. There is also a special "X# Debugger Toolbar" which is also only shown during debugging.
- These windows will only show information when the app being debugged uses the X# runtime (so they will not work in combination with the Vulcan Runtime).
  If you are debugging an application written in another language that uses the X# runtime then these windows will also show information.
  We have planned to add more features to these windows in future builds, like the properties of the current selected area and the field/values in the current selected workarea

- We have added support for "FileCodeModel" for X# files. This is used by the WPF designer and XAML editor.
  This now also fixes the Goto definition in the XAML editor (#1026)
- Several properties of X# projects are now cached. This should result in slightly faster performance.
- We have added support for "Goto Definition" for User Defined commands. For example choosing "Goto definition" on the USE keyword from the USE command will bring you to its definition in our standard header file.

Bug fixes

- Fixed member completion issue with Type[,] arrays (#980)
- Fixed missing member completion in class inside namespace when same named class exists without namespace (#1204)
- Fixed an auto indent problem when an entity has an attribute in the precessing line (#1210)
- Fixed intellisense problems with static members in some cases (#1212)
- Fixed some intellisense issues with code or declarations spanning in multiple lines (#1221, #1260)
- Fixed intellisense problem with nested classes inside a namespace (#1222)
- Fixed incorrect resolving of VAR local type, when using a type cast (#1224)
- Fixed several problems with collapsing/expanding code in the editor (#1233)
- Fixed showing of bogus member completion list with unknown types (#1255)
- Fixed some problems with auto typing text with Ctrl + Space (complete Word) (#1256)
- Fixed coloring of Text .. EndText statements (#1257)
- Fixed several issues with tooltip hints with generic types (#1258, #1259, #1273)
- Fixed problem with delegate signature not showing in intellisense tooltips (#1265)
- Fixed invalid coloring of code with multiline comments (#1269)
- Fixed invalid entries in member completion after typing "self." (#1270)
- Fixed problem with calling the disassembler when path specified (in option X# Custom Editors\Other Editors\Disassembler) with spaces (#1271)
- Fixed editor coloring completely stopping when using some UDC calls (#1272)
- Fixed problem with hint not showing on CONSTANT locals in FOR statements (#1274)
- Fixed auto indent problem when code contains a LOOP or EXIT keyword (#1278)
- Fixed an exception in the editor when typing a parenthesis under specific circumstances (#1279)
- Fixed problem with incorrectly trying to open in design mode files with filenames starting with an opening bracket (#1292)
- The "XSharp Website" menu option inside VS was broken (#1297)
- Fixed problem with the Match Identical Identifiers functionality that could slow down Visual Studio
- Fixed a VS lock up that could happen when a file was opened during debugging.
- Parameter tips for classes with a static constructor and a normal constructor were not processed correctly. This has been fixed.
- When a project was opened where the dependency between a dependent item (like a .resx file or a .designer.prg file) and its parent was missing, then an exception could occur, which prevented the project from opening. This has been fixed.
- When 2 compiler errors occurred on the same line with the same error code they were sometimes shown in the VS output window but not in the Error List. This has been fixed (#1308)

VOXporter

New Features

- Added support for special tags {VOXP:COM}, {VOXP:UNC} and {VOXP:DEL} / {VOXP:REM} to comment out, uncomment and remove lines from the original VO code (#1303)

Changes in 2.16.0.5

Compiler

New Features Xbase++ dialect

We have made several changes in the way how Xbase++ class definitions are generated. Please check your code extensively with this new build !

- We now generate a class function for all classes. This returns the same object as the ClassObject() method for Xbase++ classes.
  This class function is generated, regardless of the /xpp1 compiler option.
  The Class function depends on the function __GetXppClassObject and the XSharp.XPP.StaticClassObject class that both can be found in the XSharp.XPP assembly(#1235).
  From the Class function you can access class variables and class methods.
- In Xbase++ you can have fields (VAR) and properties (ACCESS / ASSIGN METHOD) with the same name, even with same visibility. Previously this was not supported.
  The compiler now automatically makes the field protected (or private for FINAL classes) and marks it with the [IsInstance] attribute.
  Inside the code of the class the compiler will now resolve the name to the field. In code outside of the class the compiler will resolve the name to the property.
- For derived classes the compiler now automatically generates a property with the name of the parentclass, that is declared as the parent class and returns the equivalent to SUPER.
- We have fixed an issue with the FINAL, INTRODUCE and OVERRIDE keywords for Xbase++ methods (#1244)
- We have fixed some issues with accessing static class members in the XBase++ dialect (#1215)
- You can now use the "::" prefix to access class variables and class methods inside class methods.
- When a class is declared as subclass from another class then the compiler generates a (typed) property in the subclass to access the parent class, like Xbase++ does. This property returns the value "super".
- We are now supporting the READONLY clause for Vars and Class Vars. This means that the variable must be assigned in the Init() method (instance variables) or InitClass() method (Class vars)

New Features other dialects

- Inside Visual Objects you could declare fields with the INSTANCE keyword and add ACCESS/ASSIGN methods with the same name as the INSTANCE field.
  In previous builds of X# this was not supported.

X#

The compiler now handles this correctly and resolves the name to the field in code inside methods/properties of the class and resolves the name to the property in code outside of the class.

- The PPO file now contains the original white space from user defined commands and translates.

Bug fixes

- Fixed some method overload resolution issues in the VO dialect (#1211).
- Fixed internal compiler error (insufficient stack) with huge DO CASE statements and huge IF ELSEIF statements (#1214).
- Fixed a problem with the Interpolated/Extended string syntax (#1218).
- Fixed some issues with incorrectly allowing accessing static class members with the colon operator or instance members with the dot operator (#1219,#1220).
- Fixed Incorrect visibility of MEMVARs created with MemVarPut() (#1223).
- Fixed problem with _DLL FUNCTION with name in Quotes not working correctly (#1225).
- If the preprocessor generated date and/or datetime literals, then these were not recognized. This has been fixed (#1232).
- Fixed a problem with the preprocessor matching of the last optional token (#1241)
- Fixed a problem with recognizing the ENDSEQUENCE keyword in the Xbase++ dialect (#1242).
- Using a default parameter value of NIL is now only supported for parameters of type USUAL. Using NIL for other parameter types will generate a (new) warning XS9117 . Also assigning NIL to a Symbol or using NIL as parameter to a function/method call that expects a SYMBOL will now also generate that warning (#1231).
- Fixed a problem in the preprocessor where two adjacent tokens were not merged into one token in the result stream. (#1247).
- Fixed a problem in the preprocessor where the preprocessor was not detecting an optional element when the element started with a Left parenthesis (#1250)
- Fixed a problem with interpolated strings that contained literal double quotes like in `i"SomeText""{iNum}"" "`
- Fixed a problem that was introduced in an earlier build of 2.16 with local functions / procedures.
- A warning generated at parse time could lead to another warning about a preprocessor define even when that is not needed. This has been fixed.
- Fixed issue with default parameter values for parameters declared as "a := NIL,b := NIL as USUAL" introduced in an earlier build of 2.16
- Fixed issue with erratic debugger behavior introduced in an earlier build of 2.16.
- When you are referring to a type in an external assembly that depends on another external assembly, but you did not have a reference to that other external assembly, then compilation could fail without proper explanation. Now we are producing the normal error that you need to add a reference to that other assembly.
- Omitting the type for a parameter for a function or method that does not have the CLIPPER calling convention is allowed. These parameters are assumed to be of type USUAL.
  This now produces a new warning XS9118.

Breaking changes

If you are using our parser to parse source code, please check your code. We have made some changes to the language definition for the handling of if ... else statements as well

as for the case statements (a new condBlock rule that is shared by both rules). This removes some recursion in the language. Also some of the Xbase++ specific rules have been changed. Please check the language definition online

Runtime

New Features
- Added the DOY() function.
- Addeding missing ADS_LONG and ADS_LONGLONG defines.
- Improved the speed of CDX skip operations on network drives (#1165).

Bug fixes

- Fixed a problem with DbSetRelation() and RLock() (#1226).
- Adjusted implicit conversion from NULL_PSZ to string to now return NULL instead of an empty string.
- Some initialization code is now moved from _INIT procedures to the static constructor of the SQLConnection Class, in order to make it easier to use this class from non-X# apps.
- Fixed an issue with the visibility of dynamic memory variables that were created with the MemVarPut function (#1223).
- Fixed a problem with the DbServer class in exclusive mode (#1230).
- Implicit conversions from NULL_PSZ to string were returning an empty string and not NULL (#1234).
- Fixed a problem in the CTOD() function when the day, month or year were prefixed with spaces.
- Fixed an issue with OrderListAdd() in the ADS RDD. When the index is already open, then the RDD no longer returns an error.
- Fixed an issue with MemRealloc where the second call on the same pointer would return NULL_PTR (#1248).

VOSDK
- Global arrays in the SDK classes are now initialized from the class constructor of the SQLConnection class to fix problems when the main app does not include a link to the SQL Classes assembly.

Visual Studio integration

Debugger
- The debugger expression evaluator now also evaluates late bound properties and fields (if that compiler option is enabled inside your project).
  If this causes negative side effects then you can disable that in the "Tools/Options Debugging/X# Debugger options screen".
- The debugger expression evaluator now is initialized with the compiler options from your main application (if that application is an X# project).
  The settings on the Debugger Options dialog are now only used when debugging DLLs that are loaded by a non X# startup project.
- The debugger expression evaluator now always accepts a '.' character for instance fields, properties and methods, regardless of the setting in the project options.
  This is needed because several windows in the VS debugger automatically insert '.'

characters when adding expressions to the watch window or when changing values for properties or fields.

New Features

- Added support for importing Indexes in the DbServer editor.
- The X# project system now remembers which Windows were opened in the Windows editor in design mode and reopens them correctly when a solution is reopened.
- We have added templates for a Harbour console application and Harbour class library.
- We have added item templates for FoxPro syntax classes and Xbase++ syntax classes.
- The Class templates for the FoxPro and XBase++ dialect now include a class definition in that dialect.
- We have improved the support for PPO files in the VS Editor.
- We have updated some of the project templates.

Bug fixes

- Fixed a problem with incorrectly showing member list in the editor for the ":=" operator (#1061).
- Fixed VOMED generation of menu item DEFINE names that were different to the ones generated by VO (#1208).
- Fixed VOWED incorrect order of generated lines of code in some cases (#1217).
- Switched back to our own version of Mono.Cecil to avoid issues on computers that have the Xamarin (MAUI) workload in Visual Studio.
- Fixed a problem opening a form in the Form Designer that contains fields that are initialized with a function call (#1251).
- Windows that were in [Design] mode when a solution is closed, are now properly opened in [Design] mode when the solution is reopened.

Changes in 2.15.0.3

Compiler

New Features

- Implemented the STACKALLOC syntax for allocating a block of memory on the stack (instead of the heap) (#1084)
- Added ASYNC support to XBase++ methods (#1183)

Bug fixes

- Fixed missing compiler error in a few specific cases when using the dot for accessing instance members, when /allowdot is disabled (#1109)
- Fixed some issues with passing parameters by reference (#1166)
- Fixed some issues with interpolated strings (#1184)
- Fixed a problem with the macro compiler not detecting an error with incorrectly accessing static/instance members (#1186)
- Fixed incorrect line number reported for error messages on ELSEIF and UNTIL statements (#1187)
- Fixed problem with using an iVar named "Value" inside a property setter, when option /cs is enabled (#1189)

- Fixed incorrect file/line info reported in error message when the Start() function is missing (#1190)
- Fixed bogus warning about ambiguous methods in some cases (#1191)
- Fixed a preprocessor problem with nested square brackets (in SUM and REPLACE commands) (#1194)
- Fixed incorrect method overload resolution in some cases in the VO dialect (#1195)
- Fixed incorrect ambiguous call error with OBJECT/IntPtr parameters (#1197)
- Fixed erratic debugging while stepping over code in some cases (#1200, #1202)
- Fixed a problem where a missing "end keyword", such as ENDIF, NEXT, ENDDO was not reported when the code between the start and end contained a compiler warning (#1203)
- Fixed a problem in the build system where sometimes an error message about an incorrect "RuntimeIdentifier" was shown

Runtime

Bug fixes

- Fixed runtime error in DBSort() (#1196)
- Fixed error in the ConvertFromCodePageToCodePage function
- A change in the startup code for the XSharp.RuntimeState could lead to incorrect codepages

Visual Studio integration

New Features

- Added VS option for the WED to manually adjust the x/y positions/sizes in the generated resource with multipliers (#1199)
- Added new options page to control where the editor looks for identifiers on the Complete Word (Ctrl+Space) command.
- A lot of improvements to the debugger expression evaluator (#1050). Please note that this debugger expression evaluator is only available in Visual Studio 2019 and later
- Added a debugger options page that controls how expression are parsed by the new debugger expression evaluator.
  You can also change the setting here that disallows editing while debugging.
- We have added context help to the Visual Studio source code editor. When you press F1 on a symbol then we inspect the symbol. If it comes from X# then the relevant page in the help file is opened. When it comes from Microsoft then we open the relevant page from the Microsoft Documentation online.
  In a next build we will probably add an option for 3rd parties to register their help collections too.
- When a keyword is selected in the editor that is part of a block, such as CASE, OTHERWISE, ELSE, ELSEIF then the editor will now highlight all keywords from that block.
- The Jump Keywords EXIT and LOOP are now also highlighted as part of the repeat block that they belong to.
- When a RETURN keyword is selected in the editor, then the matching "Entity" keyword, such as FUNCTION, METHOD will be highlighted too.

X#

- Added a warning to the Application project options page, when switching the target framework.

Bug fixes

- Fixed previously broken automatic case synchronization, when using the cursor keys to move to a different line in the editor (#722)
- Fixed some issues with using Control+Space for code completion (#1044, #1140)
- Fixed an intellisense problem with typing ":" in some cases (#1061)
- Fixed parameter tooltips in a multiline expressions (method/function calls) (#1135)
- Fixed problem with Format Document and the PUBLIC modifier (#1137)
- Fixed a problem with Go to definition not working correctly with multiple partial classes defined in the same file (#1141)
- Fixed some issues with auto-indenting (#1142, #1143)
- Fixed a problem with not showing values for identifiers in the beginning of a new line when debugging (#1157)
- Fixed Intellisense problem with LOGICs in some cases (#1185)
- Fixed an issue where the completionlist could contain methods that were not visible from the spot where the completionlist was shown (#1188)
- Fixed an issue with the display of nested types in the editor (#1198)
- Cleaned up several X# project templates, fixing problems with incorrect placement of Debug/Output folders (#1201)
- Undoing a case synchronization in the VS editor was not working, because the editor would immediately synchronize the case again (#1205)
- Rebuilding the intellisense database no longer restarts Visual Studio (#1206)
- Now the VO Menu Editor uses the same menu item DEFINE values, as those used in the original VO app (re-porting of the app is necessary for this to work) (#1207)
- A Change to our project system and language service could lead to broken "Find in Files" functionality in some versions of Visual Studio. This has been fixed.
- Fixed an issue where goto definition was not working for protected or private members
- Fixed an issue where for certain files the Dropdown combo boxes on top of the editor were not correctly synchronized.

Documentation

Changes

- Some methods in the typed SDK were documented as Function. They are now properly documented as Method
- Property Lists and Method lists for classes now include references to methods that are inherited from parent classes. Methods that are inherited from .Net classes, such as ToString() from System.Object are NOT included.

Changes in 2.14.0.2, 3 & 4

Visual Studio Integration

Bug fixes

- Fixed an exception in the X# Editor when opening a PRG file in VS 2017
- Selecting a member from a completion list with the Enter key on a line immediately after an entry that has an XML comment could lead to extra triple slash (///) characters to be inserted in the editor
- The triple slash command to insert XML comments was not working. This has been fixed.
- Fixed a problem with entity separators not shown on the right line for entities with leading XML comments
- Fixed a peek definition problem with types in source code that do not have a constructor
- Fixed a problem with the Implement Interface action when the keyword case was not upper case
- Fixed a problem that the keyword case was prematurely synchronized in the current line.
- Fixed a problem with indenting after keywords such as IF, DO WHILE etc
- Fixed a problem with selecting words at the end of a line when debugging
- Fixed a problem where Format Document could lock up VS
- Fixed a problem that accessors such as GET and SET were not indented inside the property block
- Fixed a problem that Format Document was not working for some documents
- Changed the priority of the background scanner that is responsible for keyword colorization and derived tasks inside VS.

Changes in 2.14.0.1

Compiler

Bug fixes

- Fixed a problem with date literals resulting in a message about an unknown alias "gloal" (#1178)
- Fixed a problem that leading 0 characters in AssemblyFileVersion and AssemblyInformationalVersion were lost. If the attribute does not have the wildcard '*' then these leading zeros are preserved (#1179)

Runtime

Bug fixes

- The runtime DLLs for 2.14.0.0 were marked with the TargetFramework Attribute. This caused problems. The attribute is no longer set on the runtime DLLs (#1177)

Changes in 2.14.0.0

Compiler

Bug fixes

- Fixed a problem resolving methods when a type and a local have the same name (#922)
- Improved XML doc messages for methods implicitly generated by the compiler (INITs, implicit constructors) (#1128)
- Fixed an internal compiler error with DELEGATEs with default parameter values (#1129)
- Fixed a problem with incorrect calculation of the memory address offset when obtaining a pointer to a structure element (#1132)
- Fixed problematic behavior of #pragma warning directive unintentionally enabling/ .disabling other warnings (#1133)
- Fixed a problem with marking the complete current executing line of code while debugging (#1136)
- Fixed incompatible to VO behavior with value initialization when declaring global MEMVAR (#1144)
- Fixed problem with compiler rule for DO not recognizing the "&" operator (#1147)
- Fixed inconsistent behavior of the ^ operator regarding narrowing conversion warnings (#1160)
- Fixed several issues with CLOSE and INDEX UDC commands (#1162, #1163)
- Fixed incorrect error line reported for error XS0161: not all code paths return a value (#1164)
- Fixed bogus filename reported in error message when the Start() function is missing (#1167)
- The PDB information for a command defined in a UDC now highlights the entire row and not just the first keyword
- Fixed a problem in the CLOSE ALL and CLOSE DATABASES UDC.

Runtime

New Features

- Added 2 new values to the DbNotificationType enum: BeforeRecordDeleted and BeforeRecordRecalled. Also added AfterRecordDeleted and AfterRecordRecalled which are aliases for the already existent RecordDeleted and RecordRecalled (#1174)

Bug fixes

- Added/updated several defines in the Win32API SDK library (#696)
- Fixed a problem with "SkipUnique" not working correctly (#1117)
- Fixed an RDD scope problem when the bottom scope is larger than the highest available key value (#1121)
- Fixed signature of LookupAccountSid() function in the Win32API SDK library (#1125)
- Improved exception error message when attempting to use functions like Trim() (which alter the key string length) in index expressions (#1148)

- Fixed a Macro Compiler runtime exception when there is an assignment in an IIF statement (#1149)
- Fixed a problem with resolving the correct overloaded method in late bound calls (#1158)
- Fixed a problem with parametrized SQLExec() statements in the FoxPro Dialect
- Fixed a problem in the Days() function where the incorrect number of seconds in a day was used.
- Fixed a problem in the Advantage RDD when a FieldGet returned fields with trailing 0 characters. These are now replaced with a space.
- Fixed a problem with DBI_LUPDATE in the ADS RDDs
- Fixed the Debugger display of the USUAL type.

Visual Studio integration

New Features
- Now using the "Reference Manager" instead of the "Add Reference Dialog Box" for adding References (#21, #1005)
- Added an option to the Solution Explorer context menu to split a Windows Form in a form.prg and form.designer.prg (#33)
- We have added an options page to the Tools / Options TextEditor/X# settings that allows you to enable/disable certain features in the X# source code editor, such as "Highlight Word", "Brace Matching" etc. The option to backup the source code for the Windows Forms Editor has been moved from the Texteditor options page to the Custom Editor options page. Search for 'Backup' in the Tools/Options dialog to find the setting.
- Tooltips for all source code items now contain the Location (file name and the line/column).
- We have added "search keywords" to all of our option page. you may be able to find a page by typing the keyword that you are looking for in the search control.

Bug fixes
- Fixed a problem renaming files when a solution is under SCC with Team Foundation Server (#49)
- The WinForms designer now ignores differences in the namespaces specified in the form.prg and designer.prg files (the one from form.prg is used) (#464)
- Fixed incorrect mouse tooltip for a class in some cases (#871)
- Fixed a code completion issue on enum types with extension methods (#1027)
- Fixed some intellisense problems with enums (#1064)
- Fixed a problem with Nuget packages in VS 2022 causing first attempts to build projects to fail (#1114)
- Fixed a formatting problem in XML documentation tooltips (#1127)
- Fixed a problem with including bogus extra static members in the code completion list in the editor (#1130)
- Fixed problem with Extension methods not included in Goto Definition, Peek definition, QuickInfo tips and Parameter Tips (#1131)
- Fixed a problem in determining the correct parameter number for parameter tips when a compiler pseudo function such as IIF() was used inside the parameter list (#1134)
- Fixed a problem with selecting words with mouse double-click in the editor with underscores while debugging (#1138)

- Fixed a problem with evaluating values of identifiers with underscores in their names while debugging (#1139)
- Fixed identifier highlighting causing the VS Editor to hang in certain situations (#1145)
- Fixed indenting of generated event handler methods in the WinForms designer (#1152)
- Fixed a problem with the WinForms designer duplicating fields when adding new controls (#1154)
- Fixed a problem with the WinForms designer removing #region directives (#1155)
- Fixed a problem with the WinForms designer removing PROPERTY declarations (#1156)
- Fixed a problem that the type lookup for locals was failing in some cases (#1168)
- Fixed a problem where the existence of extension methods in code was causing a problem filling the member list (#1170)
- Fixed a problem when completing the member completion list without selecting an item (#1171)
- Fixed a problem with showing member completion on types of static members of a class (#1172)
- Fixed a problem with the indentation after single line entities, such as GLOBAL, DEFINE, EXPORT etc. (#1173)
- Fixed a problem with parameter tips for extension methods (#1175)
- Fixed a problem with tooltips for namespaces and nested classes (#1176)
- Optional tokens in UDCs were not colored as Keyword in the source code editor
- Fixed a problem in the CodeDom provider that failed to load on a Build Server because of a dependency to Microsoft.VisualStudio.Shell.Design version 15.0 when generating code for WPF projects.

Changes in 2.13.2.2

Compiler

Bug fixes

- Class members declared with only the INSTANCE modifier were generated as public. This has been changed to protected, just like in Visual Objects (#1115)

Runtime

Bug fixes

- IVarGetInfo() returned incorrect values for PROTECTED and INSTANCE members. This has been fixed.(#1116)
- The Default() function was changing usual variables initialized with NULL_OBJECT to the new value. This was not compatible with Visual Objects (#1119)

Visual Studio integration

New Features

- The Rebuild Intellisense Database menu option now asks for confirmation before restarting Visual Studio (#1120)
- The "Include Files" node in the solution explorer can now be hidden (Tools/ Options X# Custom Editors/Other Editors)

Bug fixes

- The type information for variables declared in a CATCH clause was not available. This has been fixed (#1118)
- Fixed several issues with parameter tips (#1098, #1065)
- Fixed a performance issue when the cursor was on a undeclared identifier in a "global" entity such as a function or procedure in VERY large projects
- The "Include Files" node could contain duplicate references when the source code for an #include statement contained relative paths, such as `#include "..\GlobalDefines.vh"`
- Suppressed the expansion of the "Include Files" node in the Solution Explorer when a solution is opened.
- Single character words (like i, j, k) were not highlighted with the 'highlight word' feature
- The type 'ptr' was not marked in the keyword color in quickinfo tooltips
- The nameof, typeof and sizeof keywords were not synchronized in the keyword case

Changes in 2.13.2.1

Compiler

New Features

- The parser now recognizes AS <type> clause for PUBLIC and PRIVATE memory variable declarations but ignores these with a warning
- We have added support for AS <type> for locals declared with LPARAMETERS. The function/procedure is still clipper calling convention, but the local variable is of the declared type.

Bug fixes

- The PUBLIC and PRIVATE keywords are sometimes misinterpreted as memvar declarations when the /memvar compiler option is not even selected. We have added parser rules to prevent this from happening: when /memvar is not selected then PUBLIC and PRIVATE are only used as visibility modifiers
- Fix to an issue with selecting function and method overloads (#1096, #1101)
- Build 2.13.2.0 introduced a problem that could cause a big performance problem for VERY large source files. This has been fixed in 2.13.2.1.

Runtime

Bug fixes

- When the runtime cannot resolve a late bound call to an overloaded method it produces an error message that includes a list of all relevant overloads (#875, #1096).
- The .NULL. related behavior that was added for the FoxPro dialect was breaking existing code that involves usuals. In the FoxPro dialect DBNull.Value is now seen as .NULL. but in the other dialects as a NULL_OBJECT / NIL
- Several internal members of the PropertyContainer class in the VFP library are now public

Visual Studio integration

Bug fixes

- The lookup code for Peek definition, Goto definition etc. was filtering out instance methods and only returning static methods. This has been fixed (#1111, #1100)
- Several changes to fix issues with indentation while typing (#1094)
- Fixed several problems with parameter tips (#1098, #1066, #1110)
- A recent change to support the wizard that converts packages.config to package references has had a negative impact on nuget restore operations during builds inside Visual Studio. This was fixed. (#1113 and #1114)
- Fixed recognition of variables in lines such as CATCH, ELSEIF, FOR, FOREACH etc (#1118)
- Fixed recognition of types in the default namespace (#1122)

Changes in 2.13.1

Compiler

New Features
- The PUBLIC and PRIVATE statements in the FoxPro dialect now support inline assignments, such as in
  `PUBLIC MyVar := 42`
  Without initialization the value of the PUBLIC will be FALSE, with the exception of the variable with the name "FOXPRO" and "FOX". These will be initialized with TRUE in the FoxPro dialect

Bug fixes

- Fixed a problem with initialization of File Wide publics in the foxpro dialect
- Column numbers for error messages were not always correct for complex expressions. This has been fixed (#1088)
- Corrected an issue in the lexer where line numbers were incorrect when the source contains statements that span multiple lines (by using a semicolon as line continuation character) (#1105)

- Fixed a problem in the overload resolution when one or more overloads have a Nullable parameter(#1106), such as in

```
class Dummy
    method Test (param as usual) as int
    .
    method Test(param as Int? as int
    .
end class
```

- Fixed a problem with the code generation for late bound method calls and/or array access in the FoxPro dialect with the /fox2 compiler compiler option ("compatible array handling") for variables of unknown type (#1108).
An expression such as

```
undefinedVariable.MemberName(1)
```

was interpreted as an array access but it could also be a method call.
The compiler now generates code that calls a runtime function that checks at runtime if "MemberName" is either a method or a property.
If it is a property then the runtime will assume that it is an array and access the first element.
Code with more than 2 parameters or with non-numeric parameters, such as

```
undefinedVariable.DoSomething("somestring")
```

was not affected, since "somestring" cannot be an array index.
TIP: We recommend however, to always declare variables and specify their type. This helps to find problems at compile time and will generate MUCH faster code.

Runtime

New Features

- Added functions to resolve method calls or array access at runtime (#1108)
- Added GoTo record number functionality to the WorkareasWindow in the XSharp.RT.Debugger library

Visual Studio Integration

New Features

- Now the VS Project tree shows (in a special node) include files that are used by a project (#906).
This includes include files inside the project itself but also include files in the XSharp folder or Vulcan folder (when applicable).
- We are using the built-in images of Visual Studio in the project tree and on several other locations when possible.
- Our background parser inside VS is now paused during the built process to interfere less with the build.
- We have added a setting to the indentation options so you can control the indentation for class fields and properties separately from methods.

So you can choose to indent the fields and properties and to not indent the methods. This has also been added to the .editorconfig file

Bug Fixes

- Fixed problems with Peek Definition and Goto Definition
- When looking up Functions we were (accidentally) sometimes also including static methods in other classes.
- When parsing tokens for QuickInfo and Peek Definition then a method name would not be found if there was a space following the name and before the open parenthesis.
- Fixed a problem where project wide resources and settings (added from the project properties page) did not get the code behind file when saving
- Quick Info and Goto definition on a line that calls a constructor will now show / goto the first constructor of the type and no longer to the type declaration
- When the build process of a project was failing due to missing resources or other resource related problems, then the error list was not properly updated. This has been fixed (#1102)
- The XSharpDebugger.DLL was not installed properly into VS2017 and VS2019.

Changes in 2.13

Compiler

New Features

- We have implemented a new compiler option /**allowoldstyleassignments**, which allows using the "=" operator instead of ":=" for assignments.
  This option is enabled by default in the VFP dialect and disabled by default in all other dialects.
- We have **revised the behavior** of the /vo4 and /vo11 command line options that are related to **numeric conversions**.
  Before /vo4 only was related to conversions between integral numbers. It has now been extended to also include conversions between fractional numbers (such as float, real8, decimal and currency) and integral numbers.
  In the original languages (VO, FoxPro) you can assign a fractional number to a variable with integral value without problems.
  In .Net you can't do that but you will have to add a cast to the assignment:

```
LOCAL integerValue as INT
LOCAL floatValue := 1.5 as FLOAT
integerValue := floatValue         // no conversion: this will
not compile in .Net without conversion
integerValue := (INT) floatValue   // explicit conversion: this
does compile in .Net
? integerValue
```

If you enable the compiler option /vo4 then the assignment without the cast will also work.
The /vo4 compiler option adds an implicit conversion
In both cases the compiler will produce a warning:

warning XS9020: Narrowing conversion from 'float' to 'int' may lead to loss of data or overflow errors
The **value of the integer** integerValue above is controlled by the /vo11 compiler option:
By default in .Net conversions from a fractional value to an integer value will round towards zero, so the value will then be 1.
If you enable the compiler option /vo11 then the fractional number will be rounded to the nearest even integral value, so the value of integerValue in the example will be 2.
This is not new.
We have made a change in build 2.13, to make sure that this difference is no longer determined at runtime for the X# numeric types but at compile time.
In earlier builds this was handled inside conversion operators from the FLOAT and CURRENCY types in the runtime.
These classes choose the rounding method based on the /vo11 setting from the main program which is stored in the RuntimeState object.
However that could lead to unwanted side effects when an assembly was compiled with /vo11 but the main program was not.
This could happen for example with ReportPro or bBrowser.
If the author of such a library now chooses to compile with /vo11 then he can be certain that all these conversions in his code will follow rounding to zero or rounding to the nearest even integer, depending on his choice.

- The DebuggerDisplay attribute for Compile Time Codeblocks has changed. You now see the source code for compile time codeblocks in the debugger.

Bug fixes

- Fixed a code generation issue with ASYNC/AWAIT (#1049)
- Fixed an Internal compiler error with Evaluate() in CODEBLOCK in VFP dialect (#1043)
- Fixed an Internal compiler error with UDCs incorrectly inserted after an END FUNCTION statement
- Fixed a problem in the preprocessor with #region and #endregion in nested include files (#1046)
- Fixed some problems with evaluating DEFINEs based on the order they appear (#866, #1057)
- Fixed a compiler error with nested BEGIN SEQUENCE .. END SEQUENCE statements (#1055)
- Fixed some problems with codeblocks containing complex expressions (#1056)
- Fixed problem assigning function to delegate, when /undeclared+ is enabled (#1051)
- Fixed a bogus warning when defining a LOCAL FUNCTION in the Fox dialect (#1017)
- Fixed a problem with the Linq Operation Sum on FLOAT values (#965)
- Fixed a problem with using SELF in an anonymous method/lamda expression (#1058)
- Fixed an InvalidCastException when casting a Usual to a Enum defined as DWord (#1069)
- Fixed incorrect emitted code when calling AScan() with param nStart supplied and similar functions (#1062, #1063)
- Fixed a problem with resolving the correct one form overloads of the same function that span across different assemblies (#1079)

- Fixed unexpected behavior of the preprocessor with #translate for specific XBase++ code (#1073)
- Fixed a problem with unexpected behavior of "ARRAY OF" (#885)
- Fixed some issues with calling specific overloads of functions accepting an ARRAY as a first argument (#1074)
- Fixed a bogus XS0460 error when using the PUBLIC keyword on a method (#1072)
- Fixed incorrect behavior when enabling Named Arguments option (#1071)
- Fixed Access violation when calling a function/method with DECIMAL argument with default value (#1075)
- Fixed some issues with #xtranslate not recognizing the Regular match marker in the preprocessor. Also fixed an issue with recognizing the double colon (::) inside expression tokens in the preprocessor. (#1077)
- Fixed some issues with declaring arrays in the VFP dialect (#848)

Runtime

Bug fixes

- Fixed some incompatibilities with VO in the Mod() function
- Fixed an exception with Copy to array in the VFP dialect when dimensions do not match (#993)
- Fixed a seeking problem with SetDeleted(TRUE) and DESCEND order (#986)
- Fixed a problem with DataListView incorrectly showing (empty) deleted records with SetDeleted(TRUE) (#1009)
- Fixed problem with SetOrder() failing with SYMBOL argument (#1070)
- Reverted a previous incorrect change in the SDK in DBServer:FieldGetFormatted() (#1076)
- Fixed several issues with StrEvaluate(), including not recognizing MEMVARs with underscores in their names (#1078)
- Fix for a problem with InList() and string values (#1095)
- The Empty() function now returns false for the values `.NULL.` and `DBNull.Value` to be compatible with FoxPro
- Fixed a problem with GetDefault()/ SetDefault() to make them compatible with Visual Objects (#1099)

New Features

- Enhancements for Unicode AnyCpu SQL classes (#1006):
- Added a property to open a Sqlselect in readonly mode. This should prevent Append(), Delete() and FieldPut()
- Implemented delay creating InsertCmd, DeleteCmd, UpdateCmd until really needed
- Added callback mechanism so customers can override the commandtext for these command (and for example route them to stored Procedures)
- When a late bound method call cannot be resolved because the method is overloaded then a better error message is now generated that also includes the prototypes of the methods found (#1096)

FoxPro dialect

- Added ADatabases() function

Visual Studio Integration

New features

- You can now control how indenting is done through the Tools/Options Text Editor/X# option pages. We have added several options that control indenting of your source code. You can also set these from an .editorconfig file if you want to enforce indenting rules inside your company.
- We have now added extensive code formatting options to the source code editor. See Tools/Options/Text Editor/X#/Indentation for available options (#430)
- We have implemented the option "Identifier Case Synchronization". This works as follows: The editor picks up the first occurrence of an Identifier (class name, variable name etc) in a source file and make sure that all other occurrences of that identifier in the same source file use the same case. This does NOT enforce casing across source files (that would be way too slow)
- We have added color settings to the VS Color dialog for Matched braces, Matched keyword and Matched identifiers. Open the Tools/Options dialog, Choose Environment/Fonts and Colors and look for the colors in the listbox that start with the word "X#". You can customize these to your liking.
- X# projects that use the Vulcan Runtime now have a context menu item that allows you to convert them to the X# runtime. Standard Vulcan assemblies will be replaced with the equivalent X# runtime assemblies. If you are using 3rd party components such as bBrowser or ReportPro then you need to replace the references to these components yourself.(#32)
- We have added an option to the language page of the project properties to set the new /allowoldstyleassignments commandline option for the compiler

Bug fixes

- Fixed a problem with Get Latest Version for solution that is under TFS (#1045)
- Fixed WinForm designer changing formatting in main-prg file (#806)
- Fixed some problems with code generation in the WinForms designer (#1042, #1052)
- Fixed a problem with formatting of DO WHILE (#923)
- Fixed problem with Light Bulb "Generate default constructor" feature (#1034)
- Fixed problem with ToolTips in the Debugger. We now parse the complete expression from the first token until the cursor location. (#1015)
- Fixed some remaining intellisense issues with .Net array locals defined with VAR (#569)
- Fixed a problem with indenting not working correctly in some cases (#421)
- Fixed a problem with auto outdenting (#919)
- Several improvements to keyword pair matching (#904)
- Fixed a problem with Code Completion showing also static members after typing a dot in "ClassName{}." #1081
- Fixed a performance Issue when typing . for .and. (#1080)
- Fixed a problem with the navigation bar while typing new classes/methods (#1041)
- Fixed incorrect info tooltips on keywords (#979)
- Fixed a possible VS freeze when using "Clean Solution" (#1053)

X#

- Fixed incorrect positioning of caret in eventhandlers in the form designer (#1092)
- Fixed a problem with the form designer failing to open forms after creating a new one (#1093)
- Right Click on a packages.config file and choosing the option "Migrate to packagereferences" did not work because inside Visual Studio there is a hardcoded list of supported project types. We are now "faking" the projecttype to make VS happy and enable the wizard.

Build System

- The XSharp.Build.Dll, which is responsible for creating the command line when compiling X# projects in VS, was not properly passing the /noconfig and /shared compiler options to the compiler. As a result the shared compiler was not used, even when the project property to use the Shared Compiler was enabled. Also the compiler was automatically including references to all the assemblies that are listed inside the file xsc.rsp, which is located inside the XSharp\bin folder.
  You may experience now that assemblies will not compile because of missing types. This will happen if you are using a type that is inside an assembly that is listed inside xsc.rsp. You should add explicit references to these assemblies in your X# project now.

Changes in 2.12.2.0

Compiler

Bug fixes

- Fixed a bug in the code generation for handling FoxPro array access with parenthesized indices (#988, #991)
- The compiler was generating incorrect warnings for locals declared with IS. This has been fixed.
- The compiler was not reporting an error on invalid usage of the OVERRIDE modifier on ACCESS/ASSIGNs, this has been fixed (#981)
- Fixed inconsistent behavior in reporting warnings and errors in several cases when converting from various numeric data types to another (#951, #987)
- Fixed some "failed to emit module" issues with iif() statements in some cases (#989)
- Fixed a problem with compiling X# code scripts (#1002)
- Fixed a problem with using classes for some specific assemblies in the macro compiler (#1003)
- Fix an incorrect error message when adding an INT to a pointer in AnyCPU mode (#1007)
- Fixed a problem with casting STRING to PTR syntax (#1013)
- Fixed a problem with PCount() when passing a single NULL argument to a CLIPPER function/method (#1016)

New Features

- We have added support for the TEXT .. ENDTEXT command in all dialects. Please note that there are several variations of this command. One variation work in ALL dialects (TEXT TO varName). Other variations depend on the dialect chosen. We have moved the support for TEXT .. ENDTEXT now also from the compiler to the preprocessor. This

means that there are also 2 new preprocessor directives, #text and #endtext (#977, #1029)

- Implemented new compiler option /vo17, which implements a more compatible to VO behavior for the BEGIN SEQUENCE..RECOVER command (#111, #881, #916):
  - For code that contains a RECOVER USING, a check is made for wrapped exceptions. When the exception is not a wrapped exception then a function in the runtime is called (**FUNCTION** _SequenceError(e **AS** Exception) **AS USUAL**) that can process the error. It can for example call the error handler, or throw the error
  - When there is no RECOVER USING clause , then the compiler generates one and from within this generated clause detects if the RECOVER was reached with a wrapped exception or a normal exception. For wrapped exceptions it gets the value and calls a special function in the runtime (**FUNCTION** _SequenceRecover(uBreakValue **AS USUAL**) **AS VOID**). When the generated recover is called with a 'normal' exception then _SequenceError function from the previous bullet is called.
- We have added support for CCALL() and CCALLNATIVE()
- The #pragma directives are now handled by the preprocessor. As a result you can add #pragma lines anywhere in your code: between entities, inside the body of an entity etc.

Runtime

Bug fixes

- Changed the prototype for AdsGetFTSIndexInfo (#966)
- Fixed a problem with TransForm and decimal types (#1001)
- Added several missing return types in the VFP assembly
- Fixed a problem with browsing a DBFVFP table in the FoxPro dialect
- Fixed an inconsistency with handling values provided by BREAK commands inside surrounding BEGIN...RECOVER statements, depending on if early or late bound call is used (#883)
- Fixed a problem with floating point format when assigning a System.Decimal value to a USUAL var (#1001)
- Fixed a runtime error with DbCopyToArray() when copying to an array that has more columns than the table, in the FoxPro dialect (#993)
- Fixed a problem with the typecast expression and numeric literals with the +/- sign in the macro compiler (#1025)
- Fixed problem in the Late binding code where a string was sometimes passed in and not properly converted to symbol
- IVarPut()/IVarGet() now throw an appropriate exception when trying to use an inaccessible (due to limiting visibility modifiers) property getter/setter (ACCESS/ASSIGN) (#1023)
- Fixed an issue with IVarGet() and IVarPut() for properties that are redefined in a subclass with the NEW modifier (#1030)
- DbDataSource now tries to lock a record when deleting or recalling the record
- Foreach was not working correctly on properties containing collections that were returned from a late bound property access such as IVarGet()(#1033)

New Features

- You can now register a delegate in the runtime state that allows you to control how the macro compiler caches types from loaded assemblies(#998).
  This delegate has to have the format:

  DELEGATE MacroCompilerIncludeAssemblyInCache(ass **as Assembly**) **AS LOGIC**

  Example:

  ```
  XSharp.RuntimeState.MacroCompilerIncludeAssemblyInCache := { a  =>
  DoNotCacheDevExpress(a)}
  FUNCTION DoNotCacheDevExpress(ass as Assembly) AS LOGIC)
      // do not cache DevExpress assemblies
      RETURN ass:Location:IndexOf("devexpress",
  StringComparison.OrdinalIgnoreCase) == -1
  ```

Compatible VO SDK

- Fixed an issue where SetAnsi(FALSE) causes SingleLineEdit controls with pictures to show random characters when entering umlauts (#1038)

Typed VO SDK

There are 2 new properties for the SQLSelect class.
- ReadOnly - which makes the SQLSelect Readonly
- BatchUpdates - which controls how updates are handled

ReadOnly cursors and delayed creation of command objects

Previously the SQLSelect class created DbCommand objects to update, insert and delete changes made to a cursor immediately when the result set was opened.
That could cause problems when a complex query was used to select data, because the DbCommandBuilder object could not figure out how to create these statements.
We are now delaying the creation of these commands until the first time they are needed.
At the same time we have now added a ReadOnly property with a default value of FALSE.
If you set ReadOnly to true then:
- Calling FieldPut(), Delete() and Append() will generate an error with Gencode EG_READONLY.
- No Command objects will be created for the SQLSelect, because the cursor cannot be updated.

If ReadOnly remains FALSE then the command objects to update, insert and delete will be created the first time they are needed.
These commands are created in the __CreateDataAdapter() method.
You can override this method and create the commands in your own subclass when you want.
The command creation and the updates work as follows:
- First a DataAdapter (of type DbDataAdapter) is created using the CreateDataAdapter method from the SQLFactory class

- Then a CommandBuilder object (of type DbCommandBuilder) is created from the CreateCommandBuilder method of the SQLFactory class
- Then the Insert, Delete and Update Command objects (all of type DbCommand) are created from the GetInsertCommand() etc methods from the DbCommandBuilder object. The DBCommandBuilder object takes the Select statement and creates commands with parameters based on the SQLSelect command
- These command objects are assigned to the DataAdapter and then the DataAdapter:Update() method is called with the DataTable that is behind the SQLSelect as argument.

Batch Updates

Normally updates in a SQLSelect will be sent to the server when you move the record pointer to a new row, or when you call Update()
If you set the BatchUpdates property to TRUE then the SQLSelect will delay sending updates to the server and will not do that for each record movement, but will wait until you call the Update() method with an argument TRUE. This will then write all the buffered changes to the server. This may then also trigger the creation of the DBCommand objects (see before).
If your table has autoincrement fields then you may want to call Requery() afterwards to see the newly assigned key values.

Visual Studio Integration

Bug fixes

- Fixed the handling for project property pages for flavored projects (#992)
- When trying to start the debugger with a non existing working directory or program file name, now an appropriate error is displayed (#996)
- Fixed a problem with the form designer generating sometimes invalid code with #regions (#1020, #935)
- The WinForms designer now by default adds the OVERRIDE keyword modifier to the generated Dispose() method (was added in the template) (#1004)
- Due to a changed threading model inside the latest VS2022 releases, error messages were sometimes not shown in the output window and the error list. This has been fixed
- Fixed a problem in the windows forms designer code generation with nested classes inside the main form class (#1031).
- Fixed problem with windows forms editor failing to open form with command based on UDC (#1037).

Sourcecode Editor

- Type lookups on full names were sometimes failing because the fullname was defined as case sensitive (#978)
- Nested type lookup was sometimes failing. This has been fixed.
- The indenting options can now also be overridden in the .editorconfig file (#999)
- When a source file was loaded in the editor then the combo boxes with types and members were not activated until the caret was moved in the buffer (#995)
- The member combobox in the editor was getting confused for code that contains local functions or local procedures.

- Fixed a lookup problem for expressions inside a conversion or cast with a keyword, such as DWORD(SomeExpression). There were no quick info tips for the expression inside the parentheses for the conversion (#997)
- Fixed an intellisense problem with DATATYPE(<expression>) conversion expressions (#997)
- Fixed a problem with properties declared with the => symbol in their implementation causing Navigation Bar contents to be incomplete (#1008)
- Fixed several issues with code folding and formatting (#975)
- Fixed problem with typing a comma inside an argument list did not invoke the Parameters Tooltip (#1019)
- Fixed some issues with the detection of variable types for the VAR keyword (#903)
- Fixed an Intellisense problem with typing ":" or "." inside a string literal (#1021)
- Fixed a problem with unknown identifiers sometimes causing bogus member completion list to show (#1022)
- Pressing CTRL+SPACE in the editor now always invokes a code completion list (#957)

New Features

- Added options to insert page and reorder pages in a tabcontrol, in the VOWED (#1024)
- We have updated the WPF Application template. The Main window is now called "MainWindow".
- Added the following new settings to the .editorconfig file to set indentation options (#999).
  - indent_entity_content (true or false)
  - indent_block_content (true or false)
  - indent_case_content (true or false)
  - indent_case_label (true or false)
  - indent_continued_lines (true or false)

VOXporter

- The VOXporter now correctly enabled or disables the Allow MEMVAR/Undeclared vars compiler options, if they were enabled in the VO app (#1000)

Changes in 2.11.0.1

Compiler

Bug fixes

- Fixed an internal compiler error with CLIPPER calling convention delegates (#932)
- Fixed an AccessViolationException at runtime with the Null-conditional operator ?. on a usual property (#770)
- [XBase++ dialect] Fixed a problem with parsing method declarations with parentheses (#927)
- [XBase++ dialect] Fixed a problem with parsing the (obsolete in X#) ANNOUNCE and REQUEST statements (#929)
- [XBase++ dialect] Fixed a problem with parsing INLINE ACCESS and ACCESS ASSIGN statements (#926)

- [VFP dialect] Fixed a problem with parsing FOR EACH statements containing "M." variables usage where the variable was not typed in the FOR EACH line (#911) .
- Fixed a problem where the PPO files contains some output twice, when a single UDC was producing several statements (#933)
- Fixed some issues with the "FIELDS" clause in several UDCs (#931, #795)
- Fixed a problem in the preprocessor with parentheses in #xtranslate directives (#963)
- Fixed several more issues with #command and #translate directives (#915)
- In some cases, the compiler would emit code that does not throw a runtime exception, when casting/converting from one type to an incompatible one. This has been fixed (#961, #984)
- The compiler was not reporting narrowing conversion warnings in several cases, this has been fixed (#951)
- The compiler was not reporting signed/unsigned conversion warnings. This has been fixed (#971)
- Fixed a problem that could lead to the "Could not emit module" error message, caused by NULL values inside IIF() expressions(#989)

New features

- Added compiler option /noinit to not generate $Init calls for libraries without INIT procedures for the sake of postponed loading (#854)
- Added preprocessor support for #stdout and #if. (#912)
- The full contents of #include files is now written to the ppo file (#920)
- When a parser error occurs because an identifier was replaced by a define with the same name, then the compiler will now generate a second warning.
- If a header file contains actual code and this code is called during debugging then the debugger will now step into the header file when debugging this code.
  Previously all statements were linked to the #include line from the place where the header was included. (#967)
- When you are suppressing compiler errors with the /vo11 (Compatible numeric conversion) compiler option you will now see a XS9020 "narrowing" warning indicating that a runtime error may happen or that data may be lost.
- When you are suppressing conversion errors between signed and unsigned integers with /vo4 then you will now see as XS9021 warning indicating that data may be lost or an overflow error may occur.

Visual Studio Integration

New features

- The source code editor now also supports the new #if and #stdout preprocessor commands (#912)
- There is new "Lightbulb" option to generate constructors for classes.

Bug Fixes

- Fixed a problem with specifying custom preprocessor defines in the project properties (#909)
- The VO-style editors now retain existing "CLIPPER" clause to methods/constructors when generating code (#913)
- Fixed incorrect parsing of classes as nested to each other (#939)

- Fixed a problem with using embedded variables in the form of $(SomeName) in the project settings (#928)
- Fixed a problem where deleting items from a project would fail.
- Fixed a problem resolving the DLL produced by project files from other development languages, in particular SDK style C# projects (#950)
- Fixed a problem with quick info tooltip after an unrecognized identifier (#894)
- Fixed a problem with the editor incorrectly adding parentheses after auto typing a property (#974)
- Fixed extremely slow editor response when creating a new line after an #endif directive (#970)
- Fixed some intellisense issues with .Net array types (#569)
- Fixed a problem with the DevExpress DocumentManager control at design time (#976)
- Fixed an ArgumentNullException in the Output window when "Show output from" is set to "Extension" (#940)

Runtime

New features

- Added a constructor with IEnumerable to the array class (#943)
- Implemented missing functions AdsSetTableTransactionFree() and AdsGetFTSIndexInfo() (#966)
- Moved functions GetRValue(), GetGValue() and GetBValue() from the Win32API library to XSharp.RT, so they can be used by AnyCPU code (#972)
- [VFP dialect] Implemented function APrinters() (#952)
- [VFP dialect] Implemented function GetColor() (#973)
- [VFP dialect] Implemented functions Payment(), FV() and PV() (#964)
- [VFP dialect] Implemented commands MKDIR, RMDIR and CHDIR (#614)

Bug fixes

- Fixed a problem with the ListView TextColor and TextBackgroundColor ACCESSes in the SDK (#896)
- Fixed a problem with soft Seek not respecting order scope when to strict key is found (#905)
- Fixed DBUseArea() search logic for files in various folders. Also SetDefault() is no longer initialized with the current directory (for VO compatibility) (#908)
- Fixed problem with creating dbfs with character fields with length > 255 (#917)
- Fixed a problem with the buffered read system in some cases when a dbf was being read, closed, overwritten and then reopened (#968)
- Fixed a VO compatibility problem with how DBSetIndex() changes the active order when opening index files (#958)
- Fixed a problem with db append, copy etc, when both source and destination files have the same structure and include a memo file (#945)
- Fixed an incorrect result of DBOrderInfo(DBOI_ORDERCOUNT) with a non existing or not open index file (#954)
- [VFP dialect] Added optional parameter to Program([,lShowSignature default=.f.]) (#712)
- [VFP dialect] Fixed several issues with the Type() function (#747, #942)
- [VFP dialect] Fixed a problem with ExecScriptFast() (#823)

- [VFP dialect] Fixed a problem with SQLExec() not putting the record pointer on the first record (#864)
- [VFP dialect] Fixed a problem with SQLExec() with null values (#941)
- [VFP dialect] Fixed a write error in the buffer returned from SqlExec() (#948)
- [VFP dialect] Fixed a problem with the DBFVFP RDD and null columns (#953)
- [VFP dialect] Fixed a problem with SCATTER TO and APPEND FROM ARRAY (#821)

Typed SDK

- Fixed a problem with the FileName property of standard open dialogs
- Fixed a problem with a FOREACH inside the Menu constructor causing handled exceptions

RDD

Bug fixes

- Fixed a problem in the DBFVFP RDD with the calculation of the keysize of nullable keys (#985)

VOXporter

Bug fixes

- Fixed incorrectly detecting pointers to functions inside literal strings and comments (#932)

Changes in 2.10.0.3

Compiler

Bug fixes
- Fixed some problems with COPY TO ARRAY command in the FoxPro dialect (#673)
- Fixed a problem with using a System.Decimal type on a SWITCH statement (#725)
- Fixed an internal compiler error with Type() in the FoxPro dialect (#840)
- Fixed a problem with generating XML documentation (#783, #855)
- Prevented a warning from appearing for members of SEALED classes when /vo3 (all members VIRTUAL) is enabled (#785)
- Fixed problems with assigning and comparing "ARRAY OF <type>" vars to NULL_ARRAY (#833)
- Fixed some issues with passing arguments by reference with the @ operator and/or using it as the AddressOf operator (#810, #899, #902)
- Fixed a problem resolving parameters passed by reference with the @ operator when the function/ method had a parameter of the pointer type (#899, #902)

New features

- Added compiler option (-enforceoverride) to make the OVERRIDE modified mandatory when overriding a parent member (#786, #846)
- The compiler now reports an error when using String2Psz() and Cast2Psz() in a non local context (since such PSZs are being released on exiting the current entity) (#775)
- **FUNCTION**s and **PROCEDURE**s now support the **ASYNC** modifier (#853)
- You can now suppress the automatic generation of the $Init1() and $Exit() functions by passing the compiler commandline
  option -noinit (#854). This is NOT yet supported in the VS Properties dialog
- Added support for the **ASTYPE** operator also for USUAL vars (#857)
- Allowed specifying **AS** <type> clause in **PUBLIC** var declarations (ignored by the compiler, but used by the editor in the future for intellisense) (#853)
- The **AS** <datatype> **OF** <classlib> clause is now also supported for several other FoxPro compatible commands, such as PARAMETERS and PUBLIC.
  Since these variables are untyped at runtime by nature, the clause is ignored by the compiler and a warning is shown.

Build System

Bug Fixes

- Running MsBuild on a X# WPF project could fail (#879)

Visual Studio Integration

New features

- We have added Visual Studio integration for **VS 2022**
- We have added support for **Package References**
- Now XML comments are automatically inserted in the editor when the user types "///". (#867, #887) Conditions:
  - Cursor must be on a line before the start of an entity
  - Cursor must NOT be before a comment line
- Now the tooltip on a class includes also information about the parent class and implemented interfaces (if any) (#860)
- We have added tooltips, parameter completion etc for the pseudo functions that are built into the compiler, such as PCount() and String2Psz().
- We have added a first version of Lightbulb tips. For now to implement missing interface members and to convert a field to a Property. More implementations and configuration options will follow
- We have added a new dialog to configure source code formatting with visual examples of the effects of the options.
- We have added the ability to log operations of the X# VS integration to the Windows debug window and/or a logfile.
  If you are experiencing unexplainable problems we will contact you and tell you how to enable these options, so you can send us a log file that shows what happened before a problem occurred inside Visual Studio. We have used Serilog for this.
- The Highlight Word feature now is case insensitive and no longer hightlights words that are part of a comment, string or inactive editor region

- We have added 'Brace Completion' to the editor

Bug Fixes

- Fixed some problems with the Format Document command (#552)
- Fixed several issues with Parameter Tooltips (#728, #843)
- Fixed problem with code completion list showing even for not defined vars/identifiers (#793)
- Fixed member completion and parameter tooltips with chained expressions (#838)
- Fixed recognition of type for VAR locals in some cases (#844)
- Fixed member completion and tooltip info problems with VOSTRUCT vars (#851)
- Fixed a problem with ignoring Line Breaks in XML Comments (#858)
- Fixed some WinForms designer problems with CHAR properties (#859)
- Fixed a problem with Goto Definition not working correctly with SUPER() constructor calls (#862)
- Fixed an error with the Rebuild Intellisense Database command, when the solution contains a space in the path (#865)
- Goto Definition for types from external assemblies was failing when there was more than one copy of VS running at the same time.
- Fixed a problem with a VOSTRUCT some times confusing the parser (#868)
- Fixed some more problems with quickinfo and member completion (#870)
- Fixed a problem in the Windows Forms designer (#873)
- Fixed an intellisense problem with ENUMs using no MEMBER keywords (#877)
- Fixed a member completion problem with inherited exception types (#884)
- If an XML topic had sub elements of type <see> or other these were not shown in the editor. This has been fixed (#900)
- Unbalanced braces were sometimes matched in the editor with keywords. This has been fixed (#892)
- Line separators were sometimes flickering. This has been fixed (#792)
- When parsing for local variables we were not processing the include files. This could lead to a situation where a local that was declared in a conditional block (#ifdef SOMEVAR) was not found. This has been fixed. The editor parser now includes the header files and #defines and #undefines found in the code even when parsing a part of the source file (#893)
- #include lines are now included in the fields/members combobox in the editor (when fields are shown). They are also saved to the intellisense database.
- The editor was trying to show QuickInfo tooltips when the cursor was over an inactive preprocessor region (#ifdef). This no longer happens.

Runtime

Bug fixes

- Fixed DBFCDX corruption that could happen with simultaneous updates (#585)
- Fixed a problem opening FoxPro tables with indexes on nullable fields (#631)
- The BlobGet() function was returning a LOGIC instead of the actual field value (#681)
- Greatly improved speed of index creation with large number of fields in the index expression (#711)
- Fixed some problems with FieldPutBytes() and FieldGetBytes() (#797)
- DBSeek() with 3rd param (lLast) TRUE had incorrect behavior in some cases (#807)

- Fixed a potential NullreferenceException that could happen when creating indexes (#849)
- Improved indentation in the text produced by the method Error.WrapRawException() (#856)
- Fixed a runtime problem when converting .Net Array <-> USUAL (#876)
- DbInfo() was returning TRUE even when an info enum was not supported.(#886)
- Fixed also a possible DBFNTX corruption problem (#889)
- DbEval() could fail in FoxPro when the codeblock was returning NIL or was VOID (#890)
- Fixed a problem with Softseek and descending indexes.
- Fixed a problem where incorrect scope expressions could lead to unexpected results. Now the server goes to (and stays at) EOF with an incorrect scope.
- Fixed a problem with accessing FoxPro arrays with the parenthesis operators in a macro expression (#805).
  Please note that for this to work you have to compile the main program with /fox2

Changes in 2.9.1.1 (Cahors)

Compiler

Bug fixes

- Fixed a problem introduced in 2.9.0.2 with define symbols not respecting the /cs compiler option in combination with the /vo8 compiler option (#816)
- Fixed an internal compiler error with assignment expressions inside object initializers when the /fox2 compiler option is enabled (#817)
- Fixed some problems with DATEs in VOSTRUCTs (#773)
- Fixed a problem in the preprocessor that would occur when using a list rule like **FIELDS** <f1> [,<fn>] in the middle of a UDC.
- Fixed a problem compiling UDCs such as **SET CENTURY** &c0n because c0n was not parsed as an identifier but as a keyword.

New features

- There is a new result marker (the NotEmpty result marker) in the preprocessor that does the same as the regular result marker, but writes a NIL value to the output when the (optional) match marker is not found in the input.
  This can be used when you want to make the result a part of an IIF() expression in the output, since the sections inside an IIF expression may not be empty.
  The result marker looks like this: `<!marker!>`
- Using a Restricted match marker as the first token in an UDC was not allowed before. This has been fixed. You can now write a rule like this, which will output the keyword (SCATTER, GATHER or COPY) followed by the stringified list of options.

```
#command <cmd:SCATTER,GATHER,COPY> <*clauses*> => ?  <"cmd">,
<"clauses">
```

```
FUNCTION Start AS VOID
    SCATTER TO TEST  // is preprocessed into ? "SCATTER" , "TO
TEST"
    RETURN
```

Visual Studio Integration

Bug Fixes

- Fixed a problem introduced in 2.9.0.2 with code generation for WPF projects (#820)
- Fixed a VS freezing problem after building (#819)
- Fixed some problems with code collapsing and the navigation bar for source files that contains a SELF property (#825)
- Fixed a problem with the form designer emitting invalid code when the form prg contains nested classes (#828)
- Fixed a problem with code completion showing the wrong members when opened just left to a closing paren (#826)
- Fixed a VS crash when clicking on a generic class (#827)
- Fixed a problem with the keyword colorization for expressions such as **SET CENTURY** &cOn, where &cOn was colored in the keyword color.
- Parameter tips for nested function calls required an extra space before the name of the nested function (#728)
- Fixed a problem with the form designer deleting delegates and other nested types in the form.prg (#828)
- The background process to load the types in the ClassView / ObjectView windows was slowing down the VS performance. This has been disabled for now.
- Fixed type lookup for Generic types.
- Hovering the mouse over a constructor keyword was showing a tooltip for the class and not for the constructor. This has been fixed.
- Fixed an issue in the code generator for Windows Forms for literal characters with special values (such as '\0') (#859)
- Fixed an exception in the project system when the project system was initialized in the background (for example when no X# projects were opened) (#852)
- Fixed missing code completion for the LONGINT and SHORTINT keywords (#850)
- The context menu option "View in Disassembler" is now only shown for X# projects
- Fixed code generator problem with ARRAY OF <type> (#842)
- Fixed a performance problem when clicking on code in the editor (#829)
- Fixed a problem with loading Windows Forms when the lookup of a nested type failed.

New features

- We have added a context item to the project context menu in the solution explorer to edit the project file. This will unload the project when needed and then open the file for editing.
- The Rebuild Intellisense Database menu option in the Tools/XSharp menu now unloads the current solution, deletes the intellisense database and reopens the solution to make sure that the database is recreated correctly.

- We have made some changes to the process that parses the source code for a solution in the background.
- Generic Typenames are now stored in the Name`n format in the Intellisense database, for example IList`1 for IList<T>

Runtime

New features

- Added missing ErrorExec() function (#830)
- Added support for BlobDirectExport, BlobDirectImport, BlobDirectPut and BlobDirectGet (#832)
- Fixed a problem with creating DBF files with custom file extension. Also added support for _SET_MEMOEXT (#834)
- When you do a numeric operation on two USUALs of different types we now make sure that decimal values are no longer lost (#808). For example a LONG + DECIMAL will result in a DECIMAL. See the table in the USUAL type page in this help file for the possible return values when mixed types are used.

Bug Fixes

- Fixed a problem with _PrivateCount() throwing an InvalidateOperationException (#801)
- Fixed a problem with member completion in the editor sometimes showing methods of the wrong type (#740)
- Fixed some problems with the ACopy() function (#815)
- Fixed a few issues that were remaining related to DATEs in VOSTRUCTs (#773)

Macro compiler

New features

- Added support for the & operator (#835)
- Added support for parameters by reference (both @ and REF are supported) for late bound method calls (#818)

VOXporter

Bug Fixes

- Fixed problem with incorrectly prefixing PUBLIC declarations with "@@"

Changes in 2.9.0.2 (Cahors)

Compiler

New features

- The parser now supports class variable declarations and global declarations with multiple types(#709)
  ```
  EXPORT var1 AS STRING, var2, var3 as LONG
  GLOBAL globalvar1 AS STRING, globalvar2, globalvar3 as LONG
  ```
- If you are using our parser you should be aware that the ClassVarList rule has disappeared and that the ClassVars, VoGlobal and ClassVar rules have changed.
- We have added a command to fill a foxpro array with a single value
  ```
  STORE <value> TO ARRAY <arrayName>
  ```

- When you create a VOSTRUCT or UNION that contains a DATE field, then the compiler will now use the new __WinDate structure that is binary compatible with how DATE values are stored inside a VOSTRUCT or UNION in Visual Objects (#773)
- It is now possible to use parentheses for (instead of brackets) accessing ARRAY elements in the FoxPro dialect. The compiler option /fox2 must be enabled for that to work (#746)
- We have added support (for the FoxPro dialect only) for accessing WITH block expressions inside code of a calling function / method. So you can type .SomeProperty and access the property that belongs to a WITH BLOCK expression inside the calling code. To use this Late Binding must be enabled, since the compiler does not know the type of the expression from the calling code (#811).

Bug fixes

- When you use the NEW or OVERRIDE modifier for a method where no (virtual) method in a parent class exists an error will now be generated (#586, #777)
- Fixed a problem with LOGICAL AND and OR for USUAL variables in an array (#597)
- Error messages and Warnings for some compiler generated code (such as Late bound code) were not always pointing to the right line number, but to the first line in the body of the method or function. This has been fixed. (#603)
- Fixed a problem incorrect return values for IIF expressions (#606)
- Fixed a problem in the compiler when parsing multiple method names on a DECLARE METHOD line (#708)
- Fixed a problem in the FoxPro dialect with assigning a single value to an array to fill the array (#720)
- Fixed a problem with the calculation of VOSTRUCT sizes when the structure contained a member of type DATE (734)
- The previous problem caused runtime errors (#735)

- Fixed a problem in code like this (#736)
  ```
  var aLen := ALen(Aarray)
  ```
- Fixed a compiler crash when overriding CLIPPER method with STRICT for methods with typed return value (#761)
- When the interface implementation had different casing then the definition then an incorrect error message was shown (#765)
- Fixed a compiler crash with incorrect function parameters inside a codeblock (#759)
- Recursive definitions of DEFINEs could result in an infinate loop inside the compiler causing a StackOverflowException(#755)
- Fixed a problem with late bound calls and OUT parameters (#771)
- If you compile with warning level 4 or lower then certain warnings for comparing value types to null are not shown. We have changed the default warning level to 5 now. (#772)
- Fixed a compiler crash with multiple PRIVATE &cVarName statements in the same entity (#780)
- Fixed a problem with possibly corrupting the USUAL NIL value when passing USUAL params by reference (#784)
- Fixed a problem with declared PUBLIC variables getting created as PRIVATE in the FoxPro dialect (#753)
- Fixed a problem with using typed defines as default arguments (#718)
- Fixed a problem with typed DEFINEs that could produce constants of the wrong type (#705)
- Fixed a problem with removing whitespace from #warning and #error directive texts (#798)

Runtime

New Features

- We have added several strongly typed overloads for the Empty() function that should result in a bit better performance (#669)
- We have added an event handler to the RuntimeState class. This event handler is called "StateChanged" and expected a method with the following signature:
  ```
  Method MyStateChangedEventHandler(e AS StateChangedEventArgs) AS VOID
  ```
  The StateChangeEventArgs type has properties for the Setting Enum, the OldValue and the NewValue.
  You can use this if you have to synchronize the state between the X# runtime and an external app, for example a Vulcan App, VO App or for example (this is where we are using it) with an external database server, such as Advantage.
- We have added a new (internal) type __WinDate that is used when you store a DATE value into a VoStruct or Union. This field is binary compatible with the Julian date that VO stores inside structures and unions.
- We have added an entry to the RuntimeState in which the compiler stores the current /fox2 compiler setting for the main app.
- Added runtime support to support filling FoxPro arrays by assigning a single value.

Bug Fixes

- Fixed a problem (incompatibility with VO) in the Descend() function (#779) - **IMPORTANT NOTE: If you are using Descend() in dbf index expressions, then those indexes need to be reindexed!**

- Late bound code that was returning a PSZ value was not correctly storing that inside a USUAL (#603)
- Fixed a problem in the Cached IO that could cause problems with low level file IO (#724)
- The VODbAlias() function now returns String.Empty and not NULL when called on an area where no table is open. (#733)
- Fixed a compatibility problem with the MExec() function (#737)
- The M-> prefix was not recognized correctly inside codeblocks (#738)
- The Explicit DATE -> DWORD cast was returning an incorrect value for NULL_DATE.
- Fixed a problem with late bound calls and OUT parameters (#771)
- Added a new __WinDate type that is used to store DATE values inside a VOSTRUCT or UNION. (#773)
- Fixed several problems with FoxPro arrays
- Removed TypeConstraints on T for functions that manipulate __ArrayBase<T>
- Fixed a problem with Directory() including files that match by shortname but not by longname (#800)

RDDs

- When creating a new DBF with the DBFCDX driver an existing CDX file is not automatically deleted anymore (#603)
- Fixed a problem with updating memo contents in DBFCDX (#782)
- Fixed a runtime exception when creating DBFCDX index files with long filenames (#774)
- Fixed a problem with with DBSeek() with active OrderDescend() finding even deleted records
- Fixed a problem with a missing call to AdsClearCallbackFunction() in the ADS RDD in OrderCreate() (#794)
- Fixed a problem with VODBOrdCreate function failing it the cOrder parameter contains an empty string (#809)

Macro compiler

- Fixed a problem in the Preprocessor
- Added support for parameters passed by reference with the @ operator
- Added support for M->, _MEMVAR-> and MEMVAR-> prefixes in the macro compiler
- When the Macro compiler finds 2 or more functions with the same name it now uses the same precedence rules that the compiler uses:
  o Functions in User Code are used first
  o Functions in the "Specific" runtimes (XSharp.VO, XSharp.XPP, XSharp.VFP, XSharp.Data) take precedence over the ones inside XSharp.RT and XSharp.Core
  o Functions in XSharp.RT take precedence over functions inside XSharp.Core

Visual Studio Integration

In this build we have started to use the "Community toolkit for Visual Studio extensions" that you can find on GitHub. This toolkit contains "best practices" for code for VS Extension writers, like we are. As a result more code is now running asynchronously which should result in better performance.

We have also started to remove 32 bit specific code that would become a problem when migrating to VS 2022 which is a 64 bits version if Visual Studio that is expected to ship in November 2021.

New features

- Added several new features to the editor
  - ○ The editor can now show divider lines between entities. You can enable/disable this in the options dialog (#280)
  - ○ Keyword inside QuickInfo tooltips are now colored (#748)
  - ○ Goto definition now also works on "external" types. The editor generates a temporary file that contains the type information for the external type. In the options dialog you can also control if the generated code should contains comments (as read from the XML file that comes with an external DLL). (#763)
  - ○ You can control which keyword is used for PUBLIC visibility from the Tools/Optons menu entry (PUBLIC, EXPORT or No modifier at all)
  - ○ You can control which keyword is used for PRIVATE visibility from the Tools/Optons menu entry (PRIVATE or HIDDEN).
- The various code generators inside VS now follow the capitalization rules from the source code editor.
- The intellisense database now has views that return the unique namespaces in the source code and in the external assemblies
- The X# specific menu points in the Tools menu have been moved to a separate submenut
- Added option for the WinForms designer to generate backup (.bak) files of form.prg and form.designer.prg files when saving (#799)

Bug Fixes

- Fixed several problems in the editor:
  - ○ We have made several improvements to increase the speed inside the editor (#689, #701)
  - ○ Fixed a problem in the type lookup of variables for FOREACH loops (#697)
  - ○ Parameter tips were not shown for methods selected from a completion list (#706)
  - ○ Keyword case synchronization did not work when the keyword was not followed by a space (#722)
  - ○ Goto definition always went to line 1 / column 1 in the file where a function was defined (#726)
  - ○ Code completion for Constant members of classes (#727)
  - ○ QuickInfo for DEFINES (#730, #739)
  - ○ VOSTRUCT Member completion with the '.' operator (#731)
  - ○ The ENUM and FUNC keywords are now recognized as identifier and not case synchronized in these cases.(#732)
  - ○ Fixed a problem when opening files (#742)
  - ○ Fixed parameter tip display for default values NULL, NULL_DATE and NULL_OBJECT (#743)
  - ○ Fixes broken parameter tips for constructors (#744)
  - ○ Nested classes were not always handled correctly by the intellisense (#745)
  - ○ Fixed a problem in the type lookup of variables declared with ARRAY OF <something> (#749)
  - ○ The Editor could sometimes "freeze" when the buffer contained invalid code (#751)
  - ○ Non-existing namespaces would produce a bogus completion list (#760)
  - ○ Fixed an editor exception in some cases when typing invalid code (#791)
- The code generator for Windows Forms was replacing tab characters with spaces. This has been fixed.(#438)

- Fixed a problem with the Form Designer corrupting code that contains EXPORT ARRAY OF <type>
- Fixed a problem with the Form Designer that when removing an event handler in the editor, some code was deleted (#812)
- Fixed a problem with the Form Designer converting EXPORT, INSTANCE and HIDDEN keywords to PUBLIC and PRIVATE (#802)

VO-Compatible Editors

- Now all VO-compatible editors support full Undo/Redo functionality. Also added cut/copy/paste functionality to the Menu editor
- Fixed several visual problems with VOWED controls in Design and Test mode (#741)
- Fixed a VS crash when Alt-Tabbing out of the editors, with the Properties window having focus (#764)
- Adjusted ComboBoxEx controls to have the same fixed height, as in VO. Also allowed the previous behavior, when the user has manually increased the height by more than 50 pixels, then this height is being used instead (#750)
- Added a bitmap thumbnail for the "Button Bmp" property of the Menu Editor in the Properties Window
- Added support for specifying a Ribbon in the Menu Editor. The ribbon (bitmap) to be used needs to be specified as a filename in the properties of the Menu's main item (#714)
- Fixed some issues with event code generation in the Window Editor (#441, #46)

Changes in 2.8.3.15 (Cahors)

Compiler

New features

- You can now use the .AND. logical operator and .OR. logical operator between variable names or numbers without leading or trailing whitespace (a.AND.b)
- The PRIVATE declaration in the FoxPro dialect no longer allows an initializer.
- Added support for the FoxPro NULL date ({ / / }, { - - } and { . . }) in the FoxPro dialect

Bug fixes

- Fixed a problem with a DIM array that uses a DEFINE for its dimension (#638)
- Fixed a problem with the FoxPro PUBLIC ARRAY command (The ARRAY keyword is no longer mandatory) (#662).
- Fixed a problem with DEFAULT(Usual) expressions as parameters for function / method calls (#664)
- Fixed a problem with variables declared with the LOCAL declaration and dimensioned with the DIMENSION command (#683)
- Fixed issue with overloads with the same name in different X# runtime assemblies that manifested itself with problems with FRead()(#686)

- Fixed a problem with passing PRIVATE and PUBLIC memory variables by reference (#691)
- Fixed a problem with PARAMETERS statement (#691)
- Fixed a problem with real numbers (#704) that was caused by the change in handling of .AND. and .OR.
- Fixed a problem parsing the DECLARE METHOD / ACCESS / ASSIGN lines inside class declarations.
- Fixed a problem with truncating results for binary operators (+, -, *, /) for mixed integral types (e.g. int and word)

Runtime

Bug Fixes

- The _shutdown flag in the Runtime State is now set when the system shuts down.
- Fixed a problem with the FoxPro ALen() function (#650)
- Added default values on several locations (#678)
- Fixed a problem where FRead() on a file opened by an RDD would go into an endless loop (#688)
- Fixed a problem with FieldGet() when the file is at EOF (#698)
- Fixed a scope problem when the scope was empty and a record matching the scope was added in another workarea or by another workstation (#699)
- Fixed a problem with the BOF setting after a Skip(0) (#700)

MacroCompiler

New features
- You can now use the .AND. operator between variable names or numbers without leading or trailing whitespace
- Added support for the FoxPro NULL date ({ / / }, { - - } and { . . }) in the FoxPro dialect
- Strings containing .AND. and .OR. are no longer reformatted by the macro compiler (#694)
- We have added an experimental new faster script compiler. This script compiler allows to compile statements, so no functions, classes etc.
  This new script compiler is much faster than the existing script compiler and uses a lot less memory.
  To call this script compiler use the new function ExecScriptFast() which has the same parameters as ExecScript().
  You can compile multiline scripts. The compiler should recognize all statements including PARAMETERS and LPARAMETERS to receive parameters.
  If you are using scripts in your code we would love to hear feedback.
  An example of code that should work:

```
FUNCTION Start() AS VOID
LOCAL ctest AS STRING
TRY
    cTest := "? 'Hello world'"
    ExecScriptFast(cTest)
    cTest :=String.Join(e"\n",<STRING>{;
```

```
        "PARAMETERS a,b,c",;
        "RETURN CallMe(a,b,c)"})
    ? ExecScriptFast(cTest,1,2,3)
    cTest :=String.Join(e"\n",<STRING>{;
        "LPARAMETERS a,b,c",;
        "RETURN CallMe(a,b,c)"})
    ? ExecScriptFast(cTest,1,2,3)

CATCH e AS Exception
    ? e:ToString()
    END TRY

wait
RETURN


FUNCTION CallMe(a,b,c) AS USUAL
    ? "Inside function, parameters received",a,b,c
    RETURN a+b+c
```

Please test this new functionality and let us know what you think of it.

Visual Studio Integration

New Features

- "Highlight word" now highlights words in the whole file when the cursor is outside of an entity (for example on the USING statements in the start of the file).

Bug Fixes

- Fixed a problem with displaying names of custom controls in the toolbox of the VO compatible Windows Editor
- Fixed a problem with extra spaces when loading settings from cavowed.inf for the VO compatible Windows Editor
- Fixed a problem with an incorrect completion list after an assignment statement (#658)
- Fixed an exception in the editor after deleting code (#674)
- Fixed a "freeze" problem in the VS IDE when attaching a file to the shell window (#676)
- Fixed a problem when using dot instead of colon in VO Dialect with AllowDot (#679)
- Fixed a problem with showing a completion list inside class (#685)
- Fixed a performance problem in the editor (#689)
- Fixed a problem with showing function overloads in the editor (#692)
- Fixed a problem with intellisense after a !, .NOT. or other operator (#693)
- Fixed a problem where the incorrect methods were shown in the completion list (#695)

Tools

- Fixed an issue in VOXPorter with resources and the copying to the Resources subfolder

Changes in 2.8.2.13 (Cahors)

Compiler

- Fixed issues with extension methods that were not marked as STATIC (#660)
- Fixed problem with IIF() expressions that returned an OBJECT and were assigned to a Decimal
- The pragma commands were not checking for the current dialect
- Fixed an exception in the preprocessor
- The FoxPro LOCAL ARRAY was not generating a LOCAL variable but a PRIVATE variable.
- Functions in XSharp.RT that are overridden in XSharp.VO, XSharp.VFP or XSharp.RT will no longer generate a warning. The version that is not in XSharp.RT will have preference
- Enumerating a USUAL variable in a FOREACH loop will now call a runtime function that returns the ARRAY inside the USUAL or throws an error otherwise
- Implicit conversions from OBJECT -> NUMERIC are now supported when /vo7 is enabled.

Runtime

- Enumerating a USUAL variable in a FOREACH loop will now call a runtime function that returns the ARRAY inside the USUAL or throws an error otherwise (#246)
- Fixed a problem creating index with an Eval block and 0 records (#619)
- Fixed an incompatibility with the ALen() function and array handling compared to FoxPro (#642)
- We have fixed some issues in FoxPro AIns() function (#650)
- We have added a ShowFoxArray() function that will be automatically called when you call ShowArray() on a FoxPro array (650)
- Added support for OClone()
- The _Quit() function now closes all databases and then kills the current running process (#665)
- Fixed a problem with DbOrderInfo (#666)
- Fixed a problem with the unary minus operator for currency values (#670)
- Fixed a problem in the Integer function when a Currency value was sent in (#671)
- We have added an implementation of MemCheckPtr() (#677)

Macro compiler

- Fixed a problem calling functions after a new assembly was loaded with Assembly.Load()
- Added support for passing variables by references (not yet for functions with Clipper calling convention) (#653)

VO SDK

- Fixed a problem in GetObjectByHandle() in the GUI Classes(#677)

Visual Studio Integration

- Fixed an exception on the Build Options page inside VS (#654)
- The project system did not write back the right property for the XML documentation generation (#654)
- Intellisense could crash in header files (#657)
- We have added #defines and user defined commands (#command, #translate) to the members combobox in the editor as members of the global type. You can now also do a Goto definition on a value defined with #define.
- We have fixed a problem with member completion for enums (656)
- We have fixed a problem with the Windows Forms Editor that could happen if another VS extension had loaded an older version of Mono.Cecil (#661)
- Code completion was not showing instance members when the project option "Allow dot" was enables (#679)
- The "header" new item template had a .VH extension. This has been changed to .XH
- Fix a crash in the VO Compatible windows editor that happened with an incorrect CAVOWED.INF
- Code completion inside parentheses for a method or function call was not working correctly
- Improved Build Speed in Visual Studio when no files are changed (#675)

Tools

- VO Xporter was generating 2 lines in the .xsproj file for the output folder (#672)

Changes in 2.8.1.12 (Cahors)

Compiler

- Fixed issues with interpolated strings (#598, #622):
  - The script compiler now correctly sets the AllowDot compiler option from the current active dialect in the runtime (Core & FoxPro: AllowDot = true)
  - When compiling with DOT(.) as instance method separator then the ":" character is used inside interpolated strings to prefix the format string.
  - When compiling with COLON (:) as instance method separator then the colon can not be used to separate expressions from the format string inside interpolated strings. In that case we now support a double colon (::) between the expression and the format string. For example

```
LOCAL num as LONG
num := 42
? i"num = {num::F2}" // this diplays num with 2 decimals
WAIT
```

- You can now use DATE fields inside VOSTRUCT and UNION (#595)
- Fixed an assertion error 'UnconvertedConditionalOperator' (#616)
- Fixed an assertion error in the compiler when the namespace "xsharp" is used (#618)
- Fixed an "failed to emit" problem for methods defined in COM assemblies with default arguments and arguments passed by reference (#626)
- Fixed a problem with the handling of default parameters and method calls (#629)
- Fixed a problem where the _SizeOf() operator was not calculating the right size for a VOSTRUCT (#635).
  Please note that _SizeOf() can only be calculated at compile time when your application is compiled for x86 or x64 mode. When compiling for AnyCpu we will be calculating _SizeOf() at runtime.
- Fixed a problem where the "IS Pattern" was not always working correctly for variables of type USUAL (#636)

Runtime

- Implemented the FoxPro Evl() function (#389)
- DbCloseArea() was returning TRUE even when no area was open. This was incompatible with VO. We are returning FALSE now.(#611)
- Macro compiler was not able to find functions in assemblies that were loaded dynamically (#607)
- When a DBF file was opened "readonly" and then an index was created, then a runtime error would happen when the file was closed, because the RDD was trying to set the "production index" flag in the DBF header. This flag is no longer set for files that are opened "readonly" (#610)
- Fixed an exception (that was caught) inside DbOrderInfo(DBOI_KEYCOUND) (#613)
- Fixed a problem with the Workareas debug window (#625)
- DbOrderInfo() was returning incorrect values when an index was not abailable (#627)
- Fixed a problem with TransForm() and symbol arguments (#628)
- Fixed a problem with the StrZero function (#637)
- Fixed a problem with the AELement() function (#639)

RDD System

- Fixed a problem with indexes on workareas/cursors created with the SqlExec() function when the index expression contained "nullable" fields (#630)

Macrocompiler

- The macro compiler had problems finding functions that were inside an assembly that was loaded later (#607)

Visual Studio Integration

- Fix problem with saving dialect from General Page
- Quick info and Goto definition were not working for members inside the same class when they were not prefixed with SELF:
- Fix code completion for nullable types with the '?' syntax (#567)
- Methods combobox was not correctly synchronized (#602)
- Todo comments were not always parsed correctly, They were also included when they were part of another word or when they were not the first word on the line. This has been fixed.(#617)
- Fix problem that "warnings as errors" was not saved from the Build properties page (#621)
- Fix problems that would start occurring after editor window was split (#641)
- After selecting a member of type "Assign" from the completion list the editor was incorrectly inserting a '(' character (#643)
- Typing '(' on the declaration line of an entity (function, method) would trigger parameter completion. This has been fixed.(#643)
- Parameter tips were not shown for Constructor calls (#645)
- Completion list was incorrectly including static members (#646)
- QuickInfo for external types was not including "AS Type" for the parameters (#647)
- Fixed a problem when resolving parser options for a project that was not yet completely loaded (#649)
- Local variables were not always recognized with their correct type in the editor (#651)

Installer

- The installer was adding an incorrect version of XSharp.CodeAnalysis.dll to the Global Assembly Cache. This has been fixed.

Changes in 2.8.0.0 (Cahors)

Compiler

General

- We have migrated to the latest version of the Roslyn source code.
- Passing a typed variable by reference to a function/method with clipper calling convention (untyped parameters) was not updating the local variable. This has been fixed.
- Using the @ operator in a program in the VO Dialect when the /vo7 compiler option is NOT enabled could generate code that produces an error "Cannot be boxed". (#551)
- The generated code for NULL_PSZ and NULL_SYMBOL has been optimized (#398)
- The generated VoStructAttribute on structures and unions had the wrong size when an element with the PSZ type was used. This has been fixed.

- Fixed an internal compiler error when converting NULL to LOGIC
- The _SIZEOF() operator will generate a constant now for VOSTRUCTS and UNIONS. (#545)
- Using a keyword as field name could cause problems. For example FIELD->NEXT was not handled properly. The compiler now allows that. Of course you can also use the @@ prefix to tell the compiler that in a particular case you do not mean the keyword but an identifier.
- Parenthesized expression that contained an expression list were not compiled correctly. This has been fixed.
  This could happen when you wanted to have more than one expression as part of an IIF() expression.

```
LOCAL l AS LOGIC
LOCAL v AS STRING
l := TRUE
v := "abcd"
? iif (l, (v := Upper(v), Left(v,3)), (v := Lower(v), Left(v,4)))
```

  Since Roslyn (the C# compiler) does not allow an expression list inside a conditional expression, we are converting the parenthesized expression now to a function call to a local function. The expressions inside the Parenthesized expression become the body of the new local function and the compiler calls the generated local function.
- The compiler now warns if you call a Function in a class that has a member with the same name. For example

```
CLASS Test
METHOD Left(sValue as STRING, nLen as DWORD) AS STRING
    RETURN "Test"
METHOD ToString() AS STRING
    RETURN Left("abc",2)   // This will generate a warning that the
function Left() is called and not the method Left().
                          // if you want to call the method you
will have to prefix the call with SELF:
END CLASS
```

New language features

- We have added support for **LOCAL FUNCTION** and **LOCAL PROCEDURE** statements.
  These functions and procedures become part of the statement list of another function, procedure, method etc. They have the following restrictions:
  o A LOCAL FUNCTION **must be** terminated with END FUNCTION, a LOCAL PROCEDURE must be terminated with END PROCEDURE
  o The full "signature" of normal functions is supported, so Parameters, Return type, Type Parameters and Type Parameter constraints.
  o They cannot have Attributes (they are not compiler into methods but in a special kind of Lambda expression)
  o The only valid Modifiers for a local function are UNSAFE and ASYNC

- o Because they cannot have Attributes, we also do not support untyped parameters, so all parameters must be typed
- o If you need a local function with a variable number of parameters then you can define default parameter values or use a PARAMS array
- o Local functions can access local variables from its surrounding code. Roslyn creates a special structure where it stores the variables that are shared between the local function and its surrounding code.

- Added support for **Expression bodied members**. Expression body definitions let you provide a member's implementation in a very concise, readable form. You can use an expression body definition whenever the logic for any supported member, such as a method or property, consists of a single expression. An expression body definition has the following general syntax:

```
MEMBER => expression
```

An expression-bodied method consists of a single expression that returns a value whose type matches the method's return type, or, for methods that return void, that performs some operation. For example, types that override the ToString method typically include a single expression that returns the string representation of the current object.
An example of this could be

```
CLASS MyClass
METHOD ToString() AS STRING => "My Class"
END CLASS
```

The result of this code is exactly the same as

```
CLASS MyClass
METHOD ToString() AS STRING
    RETURN "My Class"
END CLASS
```

So you could say that the => operator replaces the RETURN keyword.

- We have added support for the **Null Coalescing Operator** (??) like C# has as well as the Null Coalescing assignment operator (??=).
  This operator does a check for != null. The operator will only work on Reference types so not on value types like USUAL, DATE and the built-in types like INT.

```
FUNCTION Start() AS VOID
LOCAL s := NULL AS STRING
s := s ?? "abc"          // The ?? is the Null Coalescing
Operator
s ??= "abc"             // This is the same as the line
before but compacter
? s
RETURN
// So this will not compile
LOCAL i := 0 AS LONG
i := i ?? 42       // Operator '??' cannot be applied to
operands of type 'int' and 'int'
```

```
// But this will compile
LOCAL i := NULL AS LONG?   // Nullable LONG
i := i ?? 42
```

- We have added support for the **Properties with INIT accessors**. These accessors allow you to assign a value to a property but only in the constructor. The property will be read only outside of he constructor of the class / structure.
- We have added a new compiler option **/enforceself**. When this option is used then all calls to instance methods inside a class must be prefixed with SELF (or SUPER). In the FoxPro dialect THIS is supported too. Please note that some generated code, such as inside the Windows Forms editor does not use SELF: and applying this compiler option may force you to change the generated code, or may force you to add an #pragma options("enforceself", disable) to the code to disable the option for that file.
- We have added a new compiler option **/allowdot**. With this option you can control if the DOT (".") operator is allowed to be used to access instance members. The default for the Core and FoxPro dialect is /allowdot+. The default for the other dialects is /allowdot-. You can also use this with a #pragma: #pragma options("allowdot", enable)
- XML comments in the source code no longer require fully qualified cref names (#467)

Preprocessor

- The preprocessor now automatically declares a match marker with the name **<udc>**. This match marker will contain all the tokens that were matched with the UDC by the preprocessor. This can be used for example to add the original source as string to the result:
  ```
  #command INSERT INTO <*dbfAndFields*> FROM MEMVAR =>
      __FoxSqlInsertMemVar(<"udc">)
  ```
- Wildcard markers (such as the dbfAndFields marker in the previous bullet) now can also appear in the middle of a UDC. They will continue to match until the first token after the Wildcard marker (in the above example the FROM keyword) is found.
- The standard header files (from the XSharp\Include folder) are now also included in the compiler as resource. When the file is missing then these files will be loaded from the resource.
- The preprocessor was not generating macros for __FOX2__ . This has been fixed (but it is now obsolete, see the FoxPro dialect)
- When wildcard tokens are included with a stringify result marker then the white space between these tokens is correctly included in the output of the UDC.

FoxPro dialect

- The feature to allow parentheses as array delimiters for the FoxPro dialect that was added in the previous build had too many side effects. We have removed this feature for now. You have to use bracketed array arguments again.
- The /fox2 compiler option is no longer needed (and ignored by the compiler).
  The compiler now checks to see if a runtime function is marked NeedsAccessToLocalsAttribute, which is defined in the XSharp.Internal namespace.
  If the compiler finds a function that is marked with this attribute, such as the Type() function or the SQLExec() function then it will add some code before and after the function call to allow these functions to access the locals on the stack. This will only happen if the /memvar compiler option is enabled and only in the FoxPro dialect.
  The NeedsAccessToLocalsAttribute has a mandatory parameter which indicates if a

function is expected to write to the locals or only read the locals.
When the function is expected to write to locals then the compiler will generate extra code after the call to make sure that the locals are updated when needed.

- We have added a standard header file for the FoxPro embedded SQL statements. This header file should parse embedded SQL but will output warnings that the embedded SQL is not yet supported.
- We have added the FoxPro array support, with a special subtype of the Array type in the runtime and support for DIMENSION and (Re)DIMENSION and filling arrays by assigning a single value. (#523)

Runtime

General

- Fixed a problem with the return value of FSeek() and FSeek3()
- AsHexString() and AsString() were not displaying the same result for PTR values as Visual Objects.
- Fixed a problem with SetScope() for the DBFCDX RDD when the previous scope was empty. (#578)
- Adjusted the Secs() function to make it more Visual Objects compatible.
- The enumerator for Array and Array Of now returns an enumerator for a clone of the internal data, to prevent runtime errors when you are modifying the array from within your code.
- The various Xbase types (DATE, FLOAT, CURRENCY, BINARY, ARRAY, USUAL etc.) are now marked with a [Serializable] attribute and implement ISerializable. They all work fine with the BinaryFormatter() classes, since that class not only stores the values but also the values in the stream. Most of the types also work with the JsonSerializer, however not all of values can be correctly deserialized with the Json serializers. (#529)
- The CompareTo() operator on the Date type was not sorting the values correctly because it was making an incorrect assumption about the memory layout of the elements in the structure. This has been fixed.
- We have made some changes in the error handling for functions such as DbUseArea() and DbSkip(). They were not always behaving the same as Visual Objects when an error occurs (for example when a file could not be opened). We have now also added an error handler similar to the default error handler in Visual Objects with a dialog that has the Abort, Retry and Ignore buttons. The Retry button is only enabled when the error object has the property "CanRetry" set to TRUE. (#587, #594)
- Fixed Val() incompatibility with string that has more than one decimal place (#572)
- Fixed a problem with comparing dates "in reverse" (#543)
- We have added a couple of functions that bring up dialogs to display the current open workareas, settings, globals and private and public memory variables. See DbgShowGlobals(), DbgShowWorkareas(), DbgShowMemvars() and DbgShowSettings()

RDD system

- DbCommit and DbCommitAll were failing when a workarea is opened Read only. This has been fixed. (#554)
- When FoxPro CDX file has more than one tag and one of the tags has an invalid index expression (for example a missing closing parenthesis, which was accepted by Visual Objects) then the RDD system did not open the CDX at all. We now open the CDX with the exception of the tag with the corrupted index expression. (#542)

- Added support for Advantage GUID and Int64 columns. GUIDs are returned as string and INT64 as INT64. We have also added some missing DEFINE values from the ACE header file.
- Fixed a problem with incorrect negative Lock Offsets in the DBFNTX driver.
- We have fixed several "exotic" problems with index "information" (KeyCount, KeyNo) etc. with indexes with Scopes, Descending indices etc. (#423, #578, #579, #580, #582, #583, #593, #599)
- Fixed problems when opening MEMO files (Fpt and DBT) from different threads and different workstations (#577)
- We have fixed a locking and corruption problem that could occur when 2 stations were frequently writing to the same CDX file. (#575, #592)
- Improved Locking Speed when a lock fails. (#576)
- SetOrder(0) was not working for ADS tables (#570)
- Changed several method prototypes for ADS to have the correct IN / OUT modifiers (#568)

Macro Compiler

- Fixed a problem in the FoxPro dialect assigning a value to an expression in the form of `VariableName.PropertyName`
- The X# macro compiler was allowing to reference GLOBAL and DEFINE values in macros. This made the compiler incompatible with VO and this would cause problems when indexing on a field with the same name as a GLOBAL or DEFINE. The support to reference GLOBALs or DEFINEs has been removed from the macro compiler. (#554)
- The Macro compiler had a problem with a variable name was surrounded with parentheses. It was seeing that as a typecast. This has been fixed. (#584)

Visual Objects SDK

- Added some missing defines to the Win32APILibrary assembly, such as DUPLICATE_SAME_ACCESS.
- DbServer:Filter was sometimes returning NIL instead of an empty string (#558)

VO Dialect

- We have added support for SysObject (#596)

Xbase++ dialect

- Fixed a problem with XPP Collation tables that was introduced in 2.7

FoxPro dialect

- Added the NeedsAccessToLocalsAttribute for the /fox2 compiler option
- Adjusted the code that exposes the values of LOCAL variables to functions such as Type() and SqlExec().
- Several functions have been marked with the new attributes so they will be able to "see" local variables.
- Added an overload of TransForm() with a single argument.
- Fixed a problem with the SQLExec() function and sql statements that contain a ":" (colon) character.
- We have added the Bit..() functions (thanks Antonio)
- We have added CapsLock(), NumLock() and InsMde (hanks Karl-Heinz)
- We have improved the FoxPro array code (#523)

Runtime Scripting

This build introduces Runtime Scripting through the ExecScript() function. At this moment you will have to include the Full macro compiler (XSharp.MacroCompiler.Full.dll) and its support DLLs (XSharp.Scripting.dll , XSharp.CodeAnalysis.dll) if you use Runtime scripting,
We are working on a light weight version of the Runtime Scripting which will be included in one of the next builds.
See the topic Runtime Scripting for more information.

Visual Studio Integration

The Visual Studio integration in this build no longer supports Visual Studio 2015. Only Visual Studio 2017 and 2019 are supported.

General

- New code templates in a subfolder were generated with a namespace name that starts with "global::". This has been fixed.
- Added support for LOCAL FUNCTION and LOCAL PROCEDURE.
- Adding an item from the Class template in a folder prefixed the namespace with "global::". This has been fixed.
- When the intellisense database file on disk was corrupted then an error occurred. Now the file is deleted and all code information is collected again.
- The Editor options in the Tools/Options dialog are now marked with "X#" and no longer with "XSharp".
- We have added a window under Tools/Options where you can set several values for our VO compatible editors, such as the grid size, paste offset etc. Look for X# in the tools options dialog.(#279, #440)
- We have added 2 new options to the formatting options for the editor: "Trim Trailing Whitespace" and "Insert Final Newline"
- Loading a MsTest project did not always work. The project file for MsTest projects will be adjusted when opening the project. (#563)
- We have added support for t4 templates (text files with a .tt extension containing scripts to generate code)
- Adding an existing .resx file did not make it a child of a parent form.prg (#197)
- The project property dialogs have been completely redesigned.

Source code editor

- Longer QuickInfo tooltips are now shown over multiple lines to make them easier to read.
- Refactored the "type lookup" code to improve the speed of the source code editor
- Member completion in the source code editor was not always working for variables declared with the VAR keyword where there were nested curly braces and/or parentheses. (#541, #560)
- Fixed a problem with member completion for project references (#540)
- Fixed an exception when uncommenting a block of lines when one of the lines in the block was an empty line.

- We have added support for .editorconfig files. See the chapter about .editorconfig files in the documentation file.
- Collapsing the last entity om the editor did not work correctly (#564)
- Fixed a problem with syntax highlighting after line continuation comments(#556)
- Added parameter completion for delegates (#581)
- Fixed a problem with certain cyrillic characters in QuickInfo tooltips (#504)

Code generator

- Character literals are now always prefixed with the 'c' prefix and values > 127 are written in Hex notation to make sure they work in all codepages.

Windows Forms Editor

- We have fixed several issues with DevExpress controls.
- Fixed a problem with a control that has the same name as a X# keyword (#566)
- Fixed a problem with a control that has a property of type DWORD (#588)
- Fixed a problem with the code generation for character literals (#550)
- The .designer.prg no longer has to have the "INHERIT FROM " clause. (#533)

Object Browser

- Goto definition was not working when you had performed a search first (#565)

VO Compatible Forms editor

- Added support in the WED for correctly visually displaying custom controls that do not have the expected control class inheritance defined
- Fixed a problem with custom controls in cavowed.inf not recognized that are not data aware
- Added support for Cloning Windows (#508)
- Fixed a problem with the display of Checkboxes (#573)
- Fixed a problem with the code generation (#553)
- There is a menu option in Tools/Options to set several settings (#279, #440)

Debugger

- The debugger now fully supports 64 bits debugging
- Added support for the new type names for CURRENCY and BINARY

Templates

- We have made adjustments to several VS item templates and project templates (#589)
- We have added a new X# t4 template (.tt file)

Changes in 2.7.0.0 (Cahors)

Compiler

General

- Fixed a problem with Nullable types that were missing an explicit cast for an assignment
- Fixed a problem with calling a parent constructor in a class hierarchy where a parent level was being skipped and the constructor for the grandparent was called instead.
- The /usenativeversion commandline option was not checking +/- switches. This has been fixed.
- Fixed a problem with PCall() and PCallNative() in source files with an embedded DOT in the filename (my.file.prg)
- We have added a new header files to the files in the XSharp\Include files that helps to add custom User Defined Commands or defines that you want to include in every project. This file (CustomDefs.xh) will be automatically include by our XSharpDefs.xh. The default contents of this file is just some comments.
  The installer will NOT overwrite the file in this folder and will not delete it when the product is uninstalled.
  You can choose to customize this file in the Include folder under Project Files. However you can also add a file with the same name to your project folder or to a common include folder for your project/solution. That last location allows you to keep the header file under source code control with the rest of your source code.

FoxPro dialect

- The compiler now allows a M Dot (M.) prefix in LOCAL, PRIVATE and PUBLIC declarations. (**LOCAL** m.Name)
- The compiler now also accepts parentheses as array delimiters in the Foxpro Dialect (aMyArray(1,2))
- The compiler now allows (and ignores) **AS** Type **OF** Classlib clauses for PRIVATE, PUBLIC, PARAMETERS and LPARAMETERS declarations.
- Support for TO keyword in CATCH clause of TRY CATCH
- Added support for the ASSERT command and SET ASSERT
- Added support for SET CONSOLE and SET ALTERNATE
- Assignments to macros with a single equals operator were not working (&myVar = 42). This has been fixed.
- Added support for zero length binary literals (0h)

Build System

- Added a project property to control if RC4005 errors (duplicate defines) should be suppressed for the Native Resource compiler

Runtime

General

- IsMethod() now returns TRUE for overloaded methods.
- AbsFloat() was "losing" the settings for # of decimal places. This has been fixed.
- Binary:ToString() was using single digit numbers for binary values < 15. This has been fixed.
- Added an implicit operator to assign a usual to a binary.
- Added an implicit conversion for USUAL values that contain an integer to an Intptr.
- Some low level functions now set the OS error number FERROR_EOF when operations fail, just like in VO.
- Exceptions in late bound code were not always showing the correct location where the error occurred. This has been fixed.
- We have added support for DataSessions. The list of open workareas/cursors in the runtime is now called "DataSession" (the old name Workareas is still available).
  You can have multiple datasessions. You can also swap the "active" DataSession in the RuntimeState with a new method SetDataSession on the RuntimeState class.
  FoxPro databases are opened in their own datasession.
  You can inspect the open DataSessions in the debugger by adding the watch expression: `XSharp.RDD.DataSession.Sessions`
  Each DataSession is associated with a Thread. When the Thread is stopped or aborted then the DataSession will be closed, which also closes all of its tables.
  At program shutdown all DataSessions are closed including their tables. This is done through an AppDomain:ProcessExit event handler.
- Low level File IO functions (including the RDD system) that open a file in Exclusive mode now use "buffered IO". This should result in faster performance.
- The (undocumented) functions to convert a Stream from/to a MemoryStream have been removed. This is replaced with the buffer I/O from the previous bullet.
- We have added System.Enum types for the FoxPro CursorProperties, DatabaseProperties and SQLProperties.
- We have added a DatabasePropertyCollection type. This type is used to add "additional" properties to Fields, such as the DBF fields for FoxPro tables.

Terminal API

- We have added support for Alternate files. `SET ALTERNATE TO` SomeFile.txt. Also `SET ALTERNATE ON` and `SET ALTERNATE OFF`
- The ? statement now respects the sessions for Set Console and Set Alternate .
- We have added support for the `SET COLOR` command. Only the fist color in the settings is used and the blink attribute is ignored and interpreted as "highlight". For example `SET COLOR TO` w+/b
- We have added a `CLEAR SCREEN` command

FoxPro dialect

- Added an Assert dialog
- Added support for DBC files. This includes the SET DATABASE to commands, DbGetProp() and reading properties for files that are part of a database without explicitly opening the database first. DbSetProp() does not do anything yet. Also functions like DbAlias() and similar have been implemented.

- The Runtime now works with DataSession object. The DBC files are opened in their own datasession as well of the files per thread. Each datasession has a list of open tables and a unique list of aliases and cursor/workarea numbers.
- AutoIncrement columns in cursors returned by SqlExec() now have a numbering scheme that starts with -1 and subtracts 1 for every new row added.
- Several settings needed for the FoxPro dialect have been added to the Set Enum.

Macro Compiler

- Until now the macro compiler was producing runtime codeblocks that take an array of objects and return an object return value. There was a class in the runtime that wrapped this and took care of usual -> object conversion for the parameters and for object-> usual conversion of the return value. This caused a problem when macros were returning a NIL value because that was converted to NULL_OBJECT.
  The reason for the OBJECT API is that the macro compiler needs to be used in the Core dialect (in the RDD system) and this dialect does not support the USUAL type. We have now added a new IMacroCompilerUsual interface in the XSharp.RT assembly that allows you to compile a string into a codeblock that supports USUAL arguments and a USUAL return value. The macro compiler now supports both this interface as well as the 'old' interface. As a result you may see a (very small) performance improvement when compiling macros.
- Calling Altd() and _GetInst() inside a macro was not supported. This has been fixed.
- The macro compiler was reporting an error when you had overridden a built-in function in your own code. We have now implemented a default MacroCompilerResolveAmbiguousMatch delegate in the runtime that now gives preference to functions that are defined in your code over functions in our code.
- When choosing between 2 overloads of a method or function the Macro compiler now chooses the method with a USUAL parameter over a method without a USUAL parameter
- Fixed a problem with calling functions/methods with a parameter by reference or an out parameter
- Added support for the CURRENCY and BINARY types to the macro compiler.

RDD System

- Exclusive DBF access now works in "buffered" mode which should make it a lot faster
- Internally the RDDs now work with the Stream objects, which makes it a bit faster.
- Fixed a problem when updating a key in an index where many duplicate key values existed.
- Removed duplicate Foxpro "machine" collations for several codepages, since they were all the same.
- For VFP compatible DBF files with field names > 10 characters you can now use the short (10 char) or the full fieldname to retrieve the values.
- The DBFVFP driver now uses the built in DBC support in the runtime to read "extended" properties for DBF files. These properties are the longer fieldname, but also the Caption etc. When a DBFVFP table is used as datasource for a DbDataSource or DbDataTable and when this data source is assigned to a Grid then the columns headers in the Grid should show the Captions from the DBC.
- DBF files with an empty codepage byte are now opened as DOS - US just like in VO and Vulcan.
- GoTop(), GoBottom() and other operations were failing for when a DBFCDX/DBFVFP area was a child in a SetRelation and when the previous parent value was resulting in an "empty" resultset.

- We have added structures and functions for the ADS Management API to the RDD assembly.

Visual Studio integration

- When creating a new VO compatible UI form in the VS IDE you can now clone an existing form.
- Fixed some problems with custom controls in the VO compatible form editor.
- Fixed several problems in the Windows Forms editor for the code parsing and code generation for DevExpress controls
- Solutions with "flavored" projects (such as MsTest projects) were not always opened correctly. An exception could occur.
- We have added an (internal) property FieldValues() to the workarea class that allows you to inspect the fieldnames and their values for the current record in the debugger. To see the current workarea in the debugger you have to add a watch expression: `XSharp.RuntimeState.DataSession.CurrentWorkarea`
- Added Project property to set the new flag to suppress RC4005 (duplicate defines) errors for the resource compiler.

Changes in 2.6.1.0 (Cahors)

This is a bug fix release with fixes for some issues found in 2.6.0.0

Compiler

- Fixed problems with passing typed variables by reference to late bound code and to untyped constructors
- Fixed an internal compiler error in code where a define containing a logic was cast to a byte
  `BYTE(_CAST, LOGICDEFINE)`.
  Of course this is code that should be avoided at all times, but unfortunately even the VO SDK is full of code like this..
  The example above should be written as `IIF(LOGICDEFINE, 1,0)` for example. The compiler will see that the define is constant and will replace that code with either 1 or 0.
- The compiler was not recognizing `$.50` as a valid Currency literal (because the 0 is missing). This is now accepted.

Runtime

- Updated the code in the runtime that handles late bound calls to improve the handling of parameters by reference
- Fixed a problem in late bound code when accessing properties such as fInit, dwFuncs and dwVars in the OleAutoObject class
- Added operator TRUE and operator FALSE to the Usual type
- Calling Val() with a NULL_STRING could cause an exception. This has been fixed.
- String properties returned by DbDataTable() and DbDataSource() are now trimmed with the TrimEnd() method of the string class.

- Added a DbTableSave() function to save changes in a DbDataTable to the current workarea.

Visual Studio integration

- Opening and upgrading project files that are under Scc could sometimes cause problems. This has been fixed
- Fixed a regression introduced in 2.6.0.0. causing the task list to no longer be updated.
- Opening a solution that referenced X# projects that do not exist on disk could cause an exception. This has been fixed.
- Opening a X# Project file that is not part of a solution could also cause an exception. This has been fixed. We'll assume the project is part of a solution file in the same folder as the project and with the same name (but different extension) as the project.
- The project system no longer makes backup files of projects that are updated. We assume you're all making backups yourself or using some kind of SCC system.
- Fixed a regression that caused the VS Tasklist not to work for X# projects.

Changes in 2.6.0.0 (Cahors)

Please note that there are some breaking changes in this build.
Therefore the Assembly version number of the Runtime Components has been changed and you will need to recompile all your code and you need new versions of 3rd party components!

Compiler

- The compiler was ignoring a (USUAL) cast. This has been fixed.
- When the compiler detects a TRY .. ENDTRY without CATCH and FINALLY then it automatically adds a CATCH class that catches all exceptions silently. This was already the case, but we now generate a warning XS9101 when this happens.
- Passing parameters by reference with an @ sign was not working correctly for late bound method calls. This has been fixed.
- Compiler option vo15 and compiler option vo16 can now also be set with a #pragma
- When /vo16 (Automatically generate Clipper calling convention constructors) was enabled then the compiler was also adding constructors to classes that are marked with the [COMImport] attribute. This has been fixed.
- Currency literals ($12.34) were not compiled into the Currency type but were stored as System.Double. This has been fixed.
- Fixed a problem with automatic version number generation for version numbers that are specified as `[assembly: AssemblyVersion ("1.0.*")]` or `[assembly: AssemblyVersion ("1.0.0.*")]`.
  If you are building with the /deterministic compiler option then an error message XS8357 is shown.
- Fixed a problem when passing a single USUAL argument to a constructor with a parameter array.
- Fixed a problem when calling an overloaded method in a class tree where one level has a parameter of one type and another level the same method name but a parameter of

another type and when there is an implicit typecast from the one type to the other (like between Date and Datetime, or between String and Symbol). The compiler now first looks to see if there is an overload with exactly the same type and where there is not then it looks for overloads for which the argument can be passed with an implicit conversion.

- The __CastClass() pseudo function can now be used to box a usual into an object or to unbox a usual from an object.
  __CastClass(USUAL, <objectValue>) unboxes the usual that is inside the object.
  __CastClass(OBJECT, <usualValue>) boxes the usual into an object.
- The <usualValue> IS SomeType VAR <newVariableOfTypeSomeType> clause was boxing the Usual into the Object before assigning it to the new variable instead of extracting the object from the usual. This has been fixed.
- Late bound assignments (such as `obj.&prop = "Jack")` were failing when the assignment operator was a single equals character. This has been fixed.
- Aliased Expressions such as SomeArea->(SomeExpression()) were returning an error on the incorrect source code line when SomeArea was not open. This has been fixed.
- We have added support for the BINARY type and BINARY Literals. See the topics about binaries and binary literals in the documentation.
- Expressions such as
      LOCAL dwDim := 512 IS DWORD
  were parsed as and compiled into
      LOCAL dwDim := (512 IS DWORD) AS USUAL
  As a result dwDim contained a USUAL with a LOGICAL value.
  This has been fixed and this code will now throw an error that DWORD variables cannot be declared with the IS keyword.
  This also happens for GLOBAL variables and Class Variables.
- We have added a MatchLike preprocessor token to match expressions that contain wildcard characters, such as in the UDC
  SAVE ALL LIKE a*,*name TO SomeFileName.
  The token to use for MatchLike is **<%name%>**
- Added support for pattern matching (WHEN clauses) in TRY .. CATCH statements, such as in the example below. The WHEN keyword is positional, so it can also be used as a variable name like in the example.

```
FUNCTION Test AS VOID
    local when := 42 as long
    TRY
        THROW Exception{"FooBar"}
    CATCH e as Exception WHEN e:Message == "Foo"
        ? "Foo", when, e:Message
    CATCH e as Exception WHEN e:Message == "Bar"
        ? "Bar", when, e:Message
    CATCH WHEN when == 42
        ? "No Foo and No Bar", when

    END TRY
    RETURN
```

- Added support for pattern matching and filters for SWITCH statements. We support both the "Identifier AS Type" clause as well as the "WHEN expression" filter clause, like in the examples below

```
VAR foo := 42
VAR iValues := <LONG>{1,2,3,4,5}
FOREACH VAR i IN iValues
    SWITCH i
    CASE 1                          // This is now called the
'constant pattern'
        ? "One"
    CASE 2 WHEN foo == 42           // Filter with a constant
pattern
        ? "Two and Foo == 42"
    CASE 2
        ? "Two"
    CASE 3
        ? "Three"
    CASE 4
        ? "Four"
    OTHERWISE
        ? "Other", i
    END SWITCH


VAR oValues := <OBJECT>{1,2.1,"abc", "def", TRUE, FALSE, 1.1m}
FOREACH VAR o in oValues
    SWITCH o
    CASE i AS LONG          // Pattern matching
        ? "Long", i
    CASE r8 AS REAL8    // Pattern matching
        ? "Real8", r8
    CASE s AS STRING  WHEN s == "abc" // Pattern matching with
filter
        ? "String abc", s
    CASE s AS STRING       // Pattern matching
        ? "String other", s
    CASE l AS LOGIC    WHEN l == TRUE   // Pattern matching with
filter
        ? "Logic", l
    OTHERWISE
        ? o:GetType():FullName, o
    END SWITCH
NEXT
```

- Please note that the performance of these patterns and filters is just like normal IF statements or DO CASE statements.
  The difference is that the compiler checks for duplicate CASE expressions so you are less likely to make mistakes.

- We have added support for the **IN** parameter modifier. This declares a parameter that is a REF READONLY parameter. You could consider to use this when passing large structures to methods or functions. Instead of passing the whole structure then the compiler will only pass the address of the structure which is 4 bytes or 8 bytes depending on if you are running in 32 bits or 64 bits.
We are planning to use this in the X# runtime for functions that accept USUAL parameters which should give you a small performance benefit (Usual variables are 16 bytes in 32 bits mode and 20 bytes when running in 64 bit mode).

Runtime

- Error messages in Late Bound code were not always showing the error causing the exception. We now retrieve the "inner most" exception so the message shows the first exception that was thrown.
- We have added runtime state settings for Set.Safety and Set.Compatible and the functions for SetCompatible and SetSafety
- A UDC used to save and restore workareas for various Db..() functions was incorrect, causing the wrong area to be selected after the function call. This has been fixed.
- The VFP MkDir() function has been added.
- Fixed a problem in late bound IVarGet() / IVarPut() when a subclass of a type implements only the Getter or the Setter and the parent class implements both.
- We have added a IDynamicProperties interface and added an implementation of this on the XPP DataObject, VFP Empty and VO OleAutoObject classes. This interface is used to optimize late bound access to properties in these classes.
- An Exception in OleAutoObject.NoMethod was not forwarded "as is" but as an argument exception.
- The Select() function now behaves differently in the FoxPro dialect to be compatible with FoxPro (no exception is thrown when the alias that is passed does not exist)
- When an Error object is created from an exception then the innermost exception is used for the error information.
- The casing of the Default() function has changed.
- We have added a new XSharp.__Binary type. See the compiler topic above for more information.
- We have added the CLOSE ALL UDC to dbcmd.xh as synonym for CLOSE DATABASES.

RDD System

- Fixed a problem in the Advantage RDD for the ADSADT driver when field names were > 10 characters.
- In the Advantage RDD the EOF, BOF and FOUND flags for tables that are a child in a relation were not properly set. This has been fixed.
- In the FoxPro dialect the 'AutoOrder' behavior has changed. In this dialect no longer the first order in the first index is selected. The index file is opened but the file stays in natural order and when opening the file the cursor is positioned on the record number 1.
- When exporting to CSV and SDF there was an exception for empty dates. This has been fixed.
- When an CDX is opened for which one of the order expressions could not be compiled (because a function is missing) then previously the complete CDX was ignored. Now the other tags are opened succesfully. The RuntimeState.LastRddError property will

contain an Exception object that contains the error message for the tag that failed to open.
- The calculation of Index keys for fields of type "I" (in the DBFVFP driver) was incorrect. This has been fixed.
- Fixed a problem with the OrdDescend() function/

Visual Studio integration

- Fixed a problem in the VS parser for default expressions in parameter lists
- Parameters for external methods/functions were not always showing the right "As"/"Is" modifiers
- The location on the QuickInfo tooltip is now shown on its own line inside the tooltip.
- Fixed a problem where the XML tooltips or parameter tips for the first member in a XML file were not shown.
- We have made a change to the project file format (see comment below). All project files will be updated when opened with this build of X#.
- Improved the speed of closing a solution inside Visual Studio.
- The project system will no longer try to update SDK style project files.
- When looking for a method such as Foo.SomeMethod() the codemodel sometimes returned a method Bar.SomeMethod().
  This was leading to problems when opening forms in the Windows Forms editor. This has been fixed.

VO Compatible editors

- Code generated from the VO Compatible editors now preserves the INTERNAL or other modifiers as well as IMPLEMENTS clauses for classes.
- We have fixed the display of "LoadResString" captions in PushButton controls

Foxpro commands

- We have added support for several new Foxpro compatible commands:
- [CLOSE ALL](#)
- [SCATTER](#)
- [GATHER](#)
- [COPY TO ARRAY](#)
- [APPEND FROM ARRAY](#)
- [COPY TO](#)   SDF|CSV|DELIMITED|FOXPLUS|FOX2X
- [APPEND FROM](#) SDF|CSV|DELIMITED|FOXPLUS|FOX2X
- All variations support a fields list, FIELDS LIKE or FIELDS EXCEPT clause and the relevant commands also support the MEMO and BLANK clauses.
- Not all variations from COPY TO and APPEND FROM are supports, such as copying to excel and sylk
- The Database and name clause in the COPY TO command are ignored for now as well as the CodePage clause

Build System

- We have prepared the X# Build System to work with SDK type projects that are used by .Net 5 and .Net Core. See the topic below for what this means for the project files.
- Please note that the source code for the Build System has been moved to the Compiler repository on GitHub, since the build system is also needed for automated builds that run outside of Visual Studio.

Changes to project files

- We are now no longer deploying our MSBuild support to a folder inside each VS version separately but we are only deploying it once in a folder inside the XSharp installation folder.
  The installer sets an environment variable XSharpMsBuildDir which points to that folder. As a result all project files will be updated when opened with this version of X#.
- The change that we make is that the macro "$(MSBuildExtensionsPath)\XSharp" is replaced with "$(XSharpMsBuildDir)" which is an environment variable that points to the location of the X# MsBuild support files on your machine. If you are running X# on a build server you can set this environment variable in your build scripts when needed.
- The installer automatically adds this environment variable and points it to the <XSharpDir>\MsBuild folder.

Changes in 2.5.2.0 (Cahors)

Compiler

- When a define contains an expression that contains the _Chr() function with a value > 127 then a warning is generated about possible code page differences between the development machine and the end users machine
- Fixed an issue where a define was defined as PTR(_CAST,0) and this define was also used as a default value for a function/method.

Runtime

- Calling IsAccess, IsAssign and similar methods on a NULL_OBJECT was causing an exception. This has been fixed.
- EmptyUsual now also works for the type OBJECT
- When a float division was returning an Infinite value then no divide by zero exception was generated. This has been fixed.
- When a parameter is skipped in a late bound call, and when that parameter has a default value, then we will now use the default value instead of NIL
- The default value of the 5th parameter (uCount) of StrTran() was "only" 65000 replacements. The default value now takes care of replacing all occurences.
- The variable name passed to NoIVarGet() and NoVarPut() is now converted to Uppercase.

RDD System

- Fixed a problem with skipping forward when a Scoped Descending Cdx was at Eof()

VOSDK

- Several DbServer methods were calling a method to write changes before the correct workarea was selected. This was an old bug originating in VO and has been fixed.

Visual Studio integration

- Looking up XML documentation was sometimes not working in VS 2019. This has been fixed.
- ClassView and Objectview are working "somewhat" now. This needs to be improved.
- Improved the loading of so called "Primary Interop Assemblies"
- Fixed a problem in the Type and Member dropdown bars in the editor window
- Improved the renaming of controls when applying copy/paste in the VO Compatible window editor.
- The X# toolbar for the VO Window editor is now automatically visible when the VO Window editor is opened
- The position and size of the property window and toolbox of the VO Window editor (and the other VO Editors) is now saved between sessions of Visual Studio.

Build System

- The generated XML files were generated in the project folder and not in the intermediate folder. This has been fixed.

Documentation

- The [Source] links were missing for most topics. This has been fixed.
- Corrected some docs

Changes in 2.5.1.0 (Cahors)

Compiler

- no changes to the compiler in this build (it is still called 2.5.0.0)

Runtime

- (VO Compatibility) Fixed a VO compatibility issue for arrays . Accessing an single dimensional array with an index with 2 dimensions now returns NIL and does not generate an exception. This is stupid but compatible.

- (VO Compatibility) Comparing a usual with a numeric value with a symbol no longer generates an exception. The numeric value is now casted to a symbol and that symbol is used for the comparison.
- (XPP compatibility) Accessing a USUAL variable with the index operator (u[1]) is not allowed for usuals containing a LONG. This will return TRUE or FALSE and is a simple way to check if a bit is set.
- The Literals for "DB" and "CR" are now stored in the resources and may be changed for other languages.
- Added some optimizations to the support code for late binding

Visual Studio integration

- Reading type information for external assemblies would fail when the external assembly contained 2 types for which the names were only different in case.
- The entity parser did not recognize GET and SET accessors that were prefixed with a visibility modifier (PROTECTED SET)
- The entity parser did not recognize ENUM members that did not start with the MEMBER keyword
- Added support for the Visual Studio Task Window. Source code comments containing the words TODO or HACK (this is configurable in the Tools/Options window) are now added to the Task List. These tasls are persisted in the intellisense database, so all tasks are immediately visible after opening a solution without (re)scanning the source files.
- Fixed a problem in the Type and Member lookup for the WIndows Forms editor
- Fixed a problem in the VS debugger where we were subtracting one from index operators for arrays and collections. This was not correct (obviously).

Build System

- The file name of the generated XML file was derived from the project file name instead of the output assembly name. This has been fixed.

Changes in 2.5.0.0 (Cahors)

Compiler

- #pragma lines that were followed by incorrect syntax would "eat" the incorrect syntax causing entire methods to be excluded from compilation. This has been fixed.
- Multiline compile time codeblocks in a method /function with a VOID return type were not being compiled correctly. This has been fixed.
- The compiler now allows to type the parameters in a codeblock. Since the codeblock definition requires parameters of type USUAL this gets transformed by the compiler. The parameters will still be of type USUAL, but inside the codeblock a local variable of the proper type will be allocated. So this compiles now

```
{ | s as string, i as int| s:SubString(i,1) }
```

- The code to fill in missing parameters was causing problems when passing parameters to COM calls (Word Example from Peter Monadjemi)
- Fixed a problem passing an IntPtr, Typed pointer of the address of a VOSTRUCT to a function that accepts an object.
- We have added code to add an integer value to a PSZ, which results in a new PSZ that starts at a relative location in the original PSZ. No new buffer is allocated.
- We have fixed a problem with complex collection initializers.
- Chr() and _Chr() with an DEFINE as argument, such as _Chr(ASC_TAB) were not properly resolved by the compiler.
- The compiler was not properly parsing the syntax **PUBLIC** MyVar[123]. This has been fixed.
- Some special characters (such as the Micro Character, U+00B5) were not recognized by the compiler as valid identifiers. We have now adopted the same identifier rules that C# uses.
- Passing a pointer or PSZ in a value of type OBJECT is now handled by "boxing" the variable. So a NULL_PTR is no longer passed as NULL_OBJECT but as an object containing an IntPtr.Zero value.
- The compiler now allows to store IntPtr.Zero to a constant variable
- The compiler now allows to embed quotes inside a string by writing double quotes. So this works:

```
? "Some String that has an embedded "" character"
```

- When you declare a MEMVAR with the same name as a function, the compiler will now have no problem anymore resolving the function call. Please note that you HAVE to declare the memvar for this resolution to work.
  For example

```
    FUNCTION Start() AS VOID
    MEMVAR Test
    Test := 123      // assign to the memory variable
    Test(Test)       // call the function 'Test' with the value of
'Test'
    RETURN
    FUNCTION Test(a)
    ? a
    RETURN a
```

Common Runtime
- The Workareas class no longer has an array of 4096 elements, but uses a dictionary to hold the open RDDs. This reduces the memory used by the runtime state.
- Fixed a problem in the WrapperRDD class
- OrdSetFocus() now returns the previous active tag as STRING
- Fixed a problem in FRead() , it was not ignoring the SetAnsi() setting as it should
- Added operators on the PSZ type for PSZ + LONG and PSZ + DWORD.

- The Usual class now implements the IDisposable() interface. When it contains an object that implements IDisposable then it will call the Dispose method on that object.
- We have added Array index properties with one and two numeric indices to make code that accesses array elements a bit faster
- The code **SELECT** 10, was not working properly. This has been fixed. Thanks Karl Heinz.
- The return value of VoDbOrdSetFocus() was TRUE even when trying to set the order to a non existing index. This has been fixed.
- We fixed a problem with Set(_SET_CENTURY) when the parameter passed was a string in the "ON" or "OFF" format
- VODbOrdSetFocus() was returning TRUE even when the selected order could not be selected.
- ArrayCreate<T> was not filling the array. This has been fixed.
- Trailing or Leading spaces are now ignored by the CToD() function.
- Calling VoDbSeek() with 2 parameters now does not set lLast to FALSE but to the LAST value from the Current Scope.
- In the previous build the format for the stack trace for errors was changed (the names are all uppercase like in VO). You can now choose to enable or disable this. We have added a function SetErrorStackVOFormat() that takes and returns a logical value. The default format for the error stack is the VO format for the VO and Vulcan dialects and the normal .Net format for the other dialects.
- We have implemented the StrEvaluate() function.
- We have implemented the PtrLen() and PtrLenWrite() functions. These only work on the Windows OS() when running in x86 mode.
  For other OSes or for apps running in 64 bits these functions returns the same value as MemLen().
- When dividing 2 float numbers results in a NaN (Not a Number) value because the divisor is zero, then a DivideByZero exception will now be generated.
- When dividing 2 usual numbers results in a NaN (Not a Number) value because the divisor is zero, then a DivideByZero exception will now be generated.
- Please note that dividing 2 REAL8 (System.Double) values can still result in a NaN, because we are not "intervening" with this division.
- The OS() function now returns a more appropriate version description when running on Windows. It reads the version name from the registry and also includes a x86 and x64 flag in the version.

RDD System

- The DBF RDD Now forces a disk flush when writing a record in shared mode.
- Fixed a problem in the DBFCDX rdd that could corrupt indexes.
- We have built in a validation routine inside the DBFCDX RDD that validates the integrity of the current tag. To call this routine call DbOrderInfo with the DBOI_VALIDATE constant.
  This will validate:
    - If all records are included exactly once in the index
    - If the values for each record in the index are correct
    - If the order of the index keys in a page is correct
    - If the list of index pages in the index is correct
  When a problem is found then this call returns FALSE and a file will be written with the name <BagName>_<TagName>.ERR containing a description of the errors found.

- Most exported variables inside the Workarea class (inside XSharp.Core) and other RDD classes have been changed to PROTECTED.
We have also added some properties for variables that need to be accessed from outside of the RDD
- Fixed a problem that occurred when skipping back repeatedly from the BOF position in a scoped CDX index.
- The Zap() operation for DBFCDX was not clearing one of the internal caches. This has been fixed.
- The DBFCDX driver now closes and deletes a CDX file when the last tag in that CDX has been deleted.

Macro compiler

- The macro compiler was not recognizing 0000.00.00 as an empty date. This has been fixed.
- The macro compiler now also exotic characters in identifiers like the normal compiler. We have added the same identifier name rules that the C# compiler uses.

XBase++ Functions

- Fixed a problem in the XPP function SetCollationTable()
- DbCargo() can now also set the cargo value for a workarea to NULL or NIL
- We have added several functions, such as PosUpper(), PosLower(), PosIns() and PosDel().

VFP Functions

- Added AllTrim() , RTrim(), LTrim() and Trim() variations for FoxPro (thanks Antonio)
- Added StrToFile() and FileToStr() (thanks Antonio and Karl Heinz)

VOSDK

- We have created a Destroy() method on the CSession and CSocket class, so you can 'clean up' objects (in VO you could call Axit(), but that is no longer allowed). The derstructor on these classes will also call Destroy().
- Fixed a problem in TreeView:GetItemAttributes. It can now also be called with a hItem (which happens inside TreeViewSelectionEvent:NewTreeViewItem)
- The OpenDialog class is now resizable.
- Fixed a problem in FormattedString:MatchesTemplChar(), that was causing problems with edit controls with a picture
- Calling DataWindow:__DoValidate() late bound was not working because there are 2 overloads. This has been fixed. Please note that in the VO SDK DataWindow:__DoValidate() expects a parameter of type Control, but inside the DataBrowser code it is called with a parameter of type DataColumn. VO does not complain but in .Net that does not work !
- Fixed a problem in GetMailTimeStamp() in the Internet classes.
- We have included "typed" versions of Consoleclasses, SystemClasses and RDD classes. These are mostly strongly typed and can run in AnyCPU mode.
The SQL classes and GUI classes will follow.

Visual Studio Integration

Code Model

- We have totally rewritten the background parser and code model that is used to parse "entities" in the VS editor and that is used to build a memory model of the types, methods, functions etc in your VS solution. This parser now uses the same lexer that the compiler uses, but the entities are collected with a hand written parser (since the code in the editor buffer may contains incomplete code we can't reliably use the normal parser).
- We are now using a SQLite database to persist the code model between sessions. This reduces the memory needed by the X# project system. We are no longer keeping the entire code model in memory.
- This also means that when you reopen an existing solution we will only have to parse files that have changed since the last time they were processed. That should speed up loading of large VS solutions.
- We are now also reading type information from external code (assembly references and project references to non X# projects) using the Mono.Cecil library instead of the classes in the System.Reflection namespace. This is faster, uses less memory and, most important, we can easily unload and reload assemblies when they were changed.
- As a result of all of this, opening VS solutions should be faster and "lock up" VS less often (hopefully not at all). Also code completion and other intellisense features should be improved.

Source code editor

- Fixed a problem with the dropdown comboboxes above the editor when the cursor is in a line of code before the first entity.
- Fixed a problem that functions in the editor after a class declaration had no collapsible regions
- The code completion inside the editor now also picks up extension methods for the types themselves, but also extension methods for interfaces implemented by these types.
- The editor code now properly recognizes variables declared with the VAR keyword when they are followed by a constructor call
- If you have XML comments in your source code for entities in your solution, then these comments should be picked up by the tooltips inside Visual Studio and by the parameter completion.
- Fixed several problems in the "reformatting" code

Windows Forms editor

- Some inline assignments to fields inside classes that are used by the Windows Forms could make the form unusable by the form editor. This has been fixed.
- The Windows Forms editor was sometimes removing blank lines between entities. This has been fixed.
- User Defined Commands in code parsed by the Windows Forms Editor were not recognized and disappeared when the form was changed and saved. This has been fixed.

- Fixed a problem with setting images and similar properties with resources stored in the project resources file (which are prefixed with "global::" in the source code)

VOXporter

- We have added support to export VO Forms from the AEFs to XML format
- We have added support to export VO Menus from the AEFs to XML format

Changes in 2.4.1.0 (Bandol GA 2.4a)

Compiler

- Bracketed strings are now no longer supported in the Core dialect to avoid problems with single line external property declarations that contain attributes between the GET and SET keywords
- The PROPERTY keyword was not properly recognized after an EXTERN modifier.
- Fixed a XS9021 warning for a IIF expressions with 2 numeric constants
- In the FoxPro dialect late bound calls are now always allowed on certain types (even when the /lb compiler option is not enabled), such as USUAL and the Empty class. These types are marked with the AllowLateBound attributes in the runtime. They WILL generate a new compiler warning (XS9098).
- We have added a new compiler option -fox2. This option makes local variables visible to the macro compiler and should also be used when you use SQL statements with embedded parameters. This compiler option must be used in combination with -memvar and the FoxPro dialect

Runtime

- Fixed a problem in the DELIM Rdd that would occur when using DbServer:AppendDelimited() and DbServer:CopyDelimited().
- Fixed a problem with DbSetOrder() returning TRUE even when the order was not found.
- Fixed a problem where the File() function would return FALSE when using wildcard characters
- SqlExec() now returns columns of type Date for SQL providers that have a separate Date type
- Workareas/Cursors created with SqlExec() now have the NULL flags, Binary flags etc. set properly according to the settings read from the backend.
- Fixed and added implementation of VFP functions (Gomonth, Quarter, ChrTran, At in various variations, RAt in various variations, DMY, MDY). Thanks Karl Heinz.
- First work on parameterized SQL functions. Not finished yet.
- Some types in the runtime are now marked with a special "AllowLateBound" attribute. These types will be accepted in the FoxPro dialect as candidates for compiling latebound even when the /lb compiler option is not enabled.
- We have added support for the macro compiler to access local variables by name. This is built into the VarGet() and VarPut() functions and also the MemVarGet() and

MemVarPut() functions. Local variables will have preference over same named private or public variables. You have to enable the -fox2 compiler option for this.

- ValType() now returns "Y" for currency values and "T" for DateTime values
- No copy of the runtime state is created when that state is accessed in the Garbage collector thread.
- SQLExecute() now returns -1 when an invalid SQL statement is executed.
- Added the VarType() function
- IVarGet() and Send() now return Empty strings when a method returns a NULL_STRING and the return type is STRING

RDD

- Getting the OrdKeyNo for a scoped index was resetting the index position to the top of the index. This would affect scrollbars in browsers for scoped indexes

VOSDK

- The Console classes assembly is now marked as AnyCpu.
- Fixed a problem introduced in the previous build with the calling convention for certain functions imported from Shell32.DLL such as the Drag and Drop support.
- Fixed a problem in the PrintingDevice constructor for reading of printers when running on a Remote Desktop
- We have changed several calls to IsInstanceOf with <var> IS <Type> constructs
- Fixed typo in several IsInstanceOf() calls
- Improved "column scatter" code for the DataBrowser class

Visual Studio Integration

- If you removed all the characters from the "Commit Completion List" control in the XSharp editor options, then after restarting VS all default characters would appear. We now remember that you have cleared the list and will not refill the list again.
- Fixed a problem that caused the editor not to rescan the current buffer for changed entities
- Added project property for the new -fox2 compiler option
- The VO MDI template now has Drag and Drop enabled
- Fixed a problem in the Debugger with some of the runtime types, such as DATE that could cause an exception while debugging in VS 2019
- Fixed a problem in the part of the editor code that is responsible for showing collapsible regions and updating the comboboxes with type names and member names.
- Fixed the code generation for Tab pages in the VO compatible forms editor

Changes in 2.4.0.0 (Bandol GA 2.4)

Compiler

- Fixed problems where certain operations on integers would still return the wrong variable type
- The Unary Minus operator on unsigned integral types (BYTE, WORD, DWORD, UINT64) was returning the same type as the original, so it was not returning a negative value. This has been changed. The return value of this operator is now the next larger signed integral type.
- Using a compiler macro, such as __VERSION__ in an interpolated string was causing an internal error in the compiler. This has been fixed.
- The vo11 compiler option now only works for operations between integral and non integral types. Other behavior has been removed because the VO behavior for mixing integral types was confusing and impossible to emulate.
- Bracketed strings are now also recognized after a RETURN and GET keyword.

Runtime

- Fixed problems when subtracting a dword from a date (related to the signed/unsigned problems in the compiler)
- LUpdate() now returns a NULL_DATE for workareas have no open table.
- Added the missing ErrorStack() function (thanks Leonid)
- Added the Stack property to the Error class
- Added the SQL..() functions from Visual FoxPro. Please note that SQLExec() and SQLPrepare() with embedded parameters in the SQL statements are not supported yet. This requires a change in the compiler that is planned for the next build.
- Added a DbDataTable() function that returns a (detached) DataTable with the data from the current workarea
- Added a DbDataSource() function that returns a BindingList attached to the current workarea. Updates to properties in the bindinglist will be directly written to the attached workarea.
- Added 2 classes DbDataTable and DbDataSource that are returned by the functions with the same name.
- Fixed a problem with incorrectly formatted USUALs with numeric values
- We have added the defines from FoxPro.h to the VFP assembly
- We have added the VFP MessageBox functions, including a message box that automatically closes when a timeout has reached.
- Fixed AsHexString() to display large DWORD values that are stored inside USUALs
- Fixed several incompatibilities with VO for FLOAT->STRING conversions

RDD System

- Fixed a problem with skipping backward in a DBFCDX table with a scope

- Fixed a problem with creating unique indexes with the DBFCDX and DBFNTX drivers
- Writing NULL values to DBF columns is now always supported. When the column is a Nullable column in a DBFVFP table then the null flags are set. For other RDDs a NULL value will be written as a blank value.
- Fixed a performance issue in Append operations for all DBF based RDDs
- Fixed a problem with the DBFCDX driver that could happen when index pages were nearly full with key-value pairs with all blanks
- Fixed a problem in WrapperRDD:Open()
- Added the SDF RDD
- Added a special DbfVFPSQL RDD that is used by the SQL..() functions in the VFP support to store the results from SQL queries. The column information describing the original column from the Sql Resultset can be retrieved with the DbFieldInfo() and the `DBS_COLUMINFO` define. The return value for this call is an object of the type XSharp.RDD.DbColumnInfo.
- Added the DELIM RDD and 2 subclasses (CSV and TSV). These RDDs all return separated values. The default format for the DELIM RDD is to use the comma as separator. CSV uses semi colons and TSV uses Tabs to delimit the fields. On top of that CSV and TSV write a header row with field names.
  The "normal" Delimited operations still use DELIM. If you want to use the CSV or TSV RDDs you need to set a global setting:

```
RddSetDefault("DBFNTX")
DbUseArea(TRUE,"DBFNTX", "c:\Test\TEST.DBF")
DbCopyDelim("C:\test\test.txt")          // this uses the DELIM
RDD

RuntimeState.DelimRDD := "CSV"           // Tell the runtime to
use the CSV RDD for delimited writes
DbCopyDelim("C:\test\test.csv")          // this uses the CSV
RDD

RuntimeState.DelimRDD := "TSV"           // Tell the runtime to
use the TSV RDD for delimited writes
DbCopyDelim("C:\test\test.tsv")          // this uses the TSV
RDD
DbZap()

RuntimeState.DelimRDD := "CSV"           // Tell the runtime to
use the CSV RDD for delimited reads
DbAppDelim("C:\test\test.csv")           // this uses the CSV RDD
```

VO SDK

- PrintingDevice:Init() no longer tries to read the default printer from win.ini but from the registry
- Several other locations where the code was still accessing win.ini (with the GetProfile..() funcitons) have been updated.
- The GUI Classes were delay loading several calls to common dialog DLL and winspool.drv. This has changed because that is no longer needed in .Net.

- Cleaned up all PSZ(_CAST operations in GUI Classes.

Visual Studio integration

- Parameter tips for OUT variables were shown as REF
- XML descriptions for member with REF or OUT parameters were not found
- Fixed an exception in the VS Editor

VOXporter

- No changes in this build

Changes in 2.3.2 (Bandol GA 2.3b)

Compiler

- Added support for Bracketed Strings ([Some String containing quotes: " and '])
- Added support for Support for PRIVATE/PUBLIC syntax with &Id and &Id.Suffix notation
- EXE files were created without manifest before, unless you were using a WIN32 resource with a manifest. This manifest is now correctly added to exe files when no manifest is supplied.
- The handling of unmanaged resources in relation to version resources and manifests has changed:
    o When the compiler detects native resources it will now check to see if there is a version and/or manifest resource included.
    o When there is no manifest resource, then the default manifest resource will be added to the resources from the Win32 resource file.
    o When there is a version resource then this version resource will be replaced by the version resource that the compiler generates from the Assembly attributes.
    o This should help people coming from VO, so they can use AssemblyVersion etc for all their assemblies, also the ones that have menu and window resources.
       If there happens to be a versioninfo resource in the source then this is ignored.
    o Of course we have added a command line option to suppress this: if you use the commandline option "-usenativeversion" then the native version that is included in the Win32 resource will be used. If there is no version resource included in the Win32 resource file, then this commandline option is ignored.
- PCOUNT() and ARGCOUNT() are now supported inside ACCESS/ASSIGN methods. The number of parameters that you can pass is still fixed, but both functions will now return the # of parameters defined in ACCESS and/or ASSIGN methods.
- We fixed a problem that a compiler error "Failed to emit module" was produced instead of showing the real problem in the code (a missing type) .
- Extended match markers in the preprocessor, such as <(file)> in the USE udc, now also properly match file names.
- Improved the detection algorithm that distinguishes parenthesized expressions and typecasts. This algorithm is now:

- Built in type names between parentheses are always seen as a typecast. For example `(DWORD)`, `(SHORT)` etc.
- Other type names between parentheses may be treated as typecast but also as parenthesized expression. This depends on the token following the closing parenthesis. When this token is an operator such as +, -, / or * then this is seen as parenthesized expression. When the token following the closing parenthesis is an opening parenthesis then the expression is seen as a typecast. Some examples:

```
? (DWORD) +42        // this is a typecast
? (System.UInt32) +42  // this is a parenthesized expression and
will NOT compile
? (System.UInt32) 42   // this is a typecast because there is no
operator before 42
? (System.UInt32) (+42) // this is a typecast because +42 is
between parentheses
```

- Code that calls the Axit() method now generates a compiler error.
- We have implemented the /vo11 compiler option
- We have fixed several signed/unsigned warnings
- You can now use PCall() on typed function pointers stored inside structures (this is used in the VO Internet Server SDK)
- The lexer now recognizes (in the FoxPro dialect) the For() and Field() functions and you do not need to prefix these with @@ anymore.

Runtime

- Fix for StrZero() with negative values
- Fix for IsSpace() crashing with empty or null string
- AFill() in the VFP dialect now fills also elements in subarrays (for multi dimensional arrays)
- NoIVarGet() and NoIvarPut() no longer convert the IVar names to Symbol. That way the original casing is kept when calling the NoIVarGet() and NoIVarPut() methods in a class
- The VFP and XPP Abstract classes are now really abstract.
- Implemented VFP Empty class.
- Implemented VFP AddProperty and VFP RemoveProperty functions.
- Fixed a typo in PropertyVisibility enum name
- Fixed several errors when calling DBF related functions for a workarea that did not contain an open table.
- The Seconds() function now returns 3 decimals when running in the FoxPro dialect. Please note that you have to add SetDecimal(3) to actually see the 3rd decimal
- The Like() function is now case sensitive in the FoxPro dialect and case insensitive in all other dialects. The _Like() function is case sensitive in all dialects.
- ASort() was not accepting a 4th argument of type Object(). This has been correct: when you pass an object that has an Eval() method then this method will be called to determine the right sort order.
- When setting/restoring global State with the Set() function, some values that are synchronized by the runtime could get out of sync. This could result in incorrect date formats or similar errors. This has been fixed.

- Several VFP compatibility functions have been added (some contributed by Thomas Ganss).
- We have added several VFP functions such as
- When you set a "global setting" using the Set() function the runtime now makes sure that related settings are set accordingly. For example Setting Set.DateFormat now also updates the DateFormatNet and DateFormatEmpty.
- Fix for PadC() function with non standard filler
- We have added DBOI_COLLATION and DBS_CAPTION for FoxPro specific properties

VO SDK

- We have removed the versioninfo resource from the GUI classes sourcecode. The version info is now generated from the Assembly attributes
- We have cleaned up the code and removed the warnings 9020 and 9021 from the suppressed warnings, since the compiler now handles this correctly.

RDD system

- The DBFVP driver no longer fails to open a DBF when the DBC file is used exclusively by someone else
- Added support for reading captions with DBS_CAPTION and collations with DBOI_COLLATION
- The DBFNTX driver was not setting the HPLocking flag properly when creating new indexes

Visual Studio integration

- The type lookup for variables declared with a VAR keyword could sometimes go into an infinite loop. This has been fixed.
- Members starting with '__' are now only hidden from completion lists when the 'Hide Advanced members' checkbox in the general editor options is checked
- Added support for colorizing BRACKETED_STRING constants
- Fixed a bug in the keyword case synchronization code.
- The code behind the VS Form editor had problems with methods declared without return type. As a result forms could not be opened. This has been fixed.
- Improved intellisense info for Defines and Enum members
- You can now enable/disable /vo11 in the project properties dialog

VOXporter

- When porting from Clipboard contents, now VOXporter puts back the modified code to the clipboard
- Added option to remove ~ONLYEARLY pragmas

Installer

- The installer now has a new command line parameter "-nouninstall" that prevents the automatic installation of a previous version. This allows you to install multiple versions of X# side by side.

Please note that the installer sets a registry key to the location where X# is last installed. This location will be used by the Visual Studio integration to locate the compiler.
If you don't change this then all VS installations will always use the version of X# that is last installed. See the topic about the build process in VS and with MsBuild for information about how this mechanism works.
Also if you choose to install the X# runtime assemblies in the GAC then newer versions of these runtime DLLs will/may overwrite older versions. This depends on the fact if the newer DLLs have a new Assembly version.
At this moment all X# runtime DLLs (still) have version 2.1.0.0 even when X# itself is now on version 2.3.2.

- The installer now lists all found instances of VS 2017 and VS 2019, including the Visual Studio Buildtools, so you can choose to install in a particular instance of these versions of Visual Studio or simply in all instances.
Please note that when you run X# with the -nouninstall command line option, this will prevent the installer from removing X# from VS installations where it was previously installed.
- We have added some documentation about all the installer command line options to the help file.

Documentation

- Fixed errors in the documentation of escape codes
- We have added a chapter with tips and tricks that contains the following topics at this moment.
- Added description of the installer command line arguments
- Added description of the Build process in VS and with MsBuild
- Added topics describing the dialect "incompatibilities" in the X# runtime. Please note that this topic is not complete yet.
- How to catch errors at startup
- Compiler magic in the startup code
- Special classes generated by the compiler.

Changes in 2.3.1.0 (Bandol GA 2.3a)

Compiler

- When compiling in case sensitive mode, the compiler now checks to see if a child class declares a method that only differs from a method in its parent class by case
- The warning message about assigning to a foreach iterator variable has been changed from "Cannot assign" to "Should not assign"
- #pragma warnings was not working with the xs1234 syntax but only with numbers. This has been corrected

Runtime

- Added the SetFieldExtent method to the IRdd interface
- The USUAL type no longer "caches" the dialect setting
- Fixed some problems with ACopy() with skipped or negative arguments.
- The return value for Alias() is now in upper case.

VO SDK

- The VO SDK Console class now uses the System.Console class internally. The only functionality that is no longer available is:
  o It does not respond to the mouse anymore
  o Creating a "new" console window is not supported.

RDD system

- Fixed a problem in the Advantage RDDs that was caused by a casing problem (a method in a child class had a different case than the method in the parent class that it tried to override). This is why we also added a check to the compiler.
- Creating an NTX with the DBFNTX driver could fail in some situations due to timing issues. This has been fixed.

Visual Studio integration

- Fixed a problem in keyword case synchronization that could corrupt the editor contents.

Changes in 2.3.0.0 (Bandol GA 2.3)

Compiler

- Syntax errors (1003) or Parser errors (9002) in a source file could lead to multiple errors in the error list. We are now only reporting the first of these error types in a source file.
- Implemented the -cs (Case Sensitive identifiers) compiler option
- The compiler now includes the source for a compiletime codeblock as string in that codeblock. Calling ToString() on a compile time codeblock will retrieve this string.
- Fixed a problem that memory variables were not updated when passed to a **DO** `<proc>` **WITH** statement
- Accessing or assigning undefined properties or calling undefined methods in typed code was generating a compiler error. The compiler now detects if the type has a NoIVarGet(), NoIVarPut() or NoMethod() method, and when it finds the appropriate methods then a compiler warning (XS9094) is generated instead of a compiler error.
- Casting a numeric to a LOGIC with the **LOGIC**(**_CAST,** `numValue`) construct was only looking at the lowest byte of numValue. If the lowest byte was zero and a higher byte

was non zero the result would be FALSE. The compiler now compiles this into
(numValue <> 0).

- The compiler now supports an (optional) THEN keyword for the IF statement
- Added support for the FoxPro CURRENCY type.
- The Value keyword is always compiled in lower case in PROPERTY SET methods
- Unterminated strings are now detected at the end of the line.
- Added ENDTRY UDC for FoxPro
- Added support for #pragma warning(s). See the #pragma warnings topic in the help file for more info.
- Added support for #pragma options. See the #pragma options topic in the help file for more info.

Runtime

- Added XSharp.Data.DLL which contains support code for .Net SQL based data access used by the RDD system and the new Unicode SQL classes.
- DbEval() was throwing an exception when no FOR block or no WHILE block was passed
- DbEval() was throwing an exception when the block that is evaluated was not returning a logical expression
- The workarea event for OrdSetFocus() had an error which would result in an "Operation Failed" error for this event, even when the event succeeded.
- The index operator on USUALs containing STRINGS (which is only supported in the Xbase++ dialect) was not taking into account that the indices were already ZERO based,
- Calling DbCreate() with incorrect lengths for Date or Logic fields was throwing an exception, these are now automatically corrected
- Added a fix for converting USUAL values of type STRING with NULL to STRING
- Fixed a problem in __FIeldSetWa() when the area was NIL or "M".
- Added the FoxPro CURRENCY type. These are also supported in USUAL variables. Internally the values of a CURRENCY variable are stored as Decimal but rounded to 4 decimal places.
- Most runtime DLLs are now compiled in Case Sensitive mode.
- Fixed a problem in the STOD() function, so it allows strings that are longer than 8 characters.
- We have added some VFP functions to the runtime, such as the Just..() functions and AddBs(). Several other functions are there but not implemented. They are marked with an [Obsolete] attribute and will throw a NotImplementedException when called.
- When running on windows the low level File IO system now uses native windows File access in stead of the managed access. This also affects the RDD system.
- Fixed problems in ACopy(), Transform(), Str()

VOSDK Classes

- Added DbServer:FieldGetBytes() and DbServer:FieldPutBytes() to read the 'raw' bytes of a string field. Please note that (in ccOptimistic mode) the bytes value is NOT cached and that you have to manually lock and unlock the server when calling FieldPutBytes().
- Added several missing defines

- Synchronized the VO SDK to the VO 2.8 SP4 SDK. The only changes that are not included are the ones from the DateTimePicker class. These changes were causing conflicts with the existing code in the X# VOSDK.

RDD System

- Querying the header size for the Advantage RDD would cause an exception. This has been fixed
- Fixed a problem with DbRlockList() and the advantage RDD
- Skipping in a cursor for the Advantage RDDs was not refreshing the EOF and BOF flags for related tables
- Fixed a problem writing strings in FPT files
- The AX_Get.. Handle() functions were not properly returning the handles
- We have added several missing Advantage related functions.
- The DBFVFP driver was not writing the block for DBC backlinks to the file header when creating a new file, which resulted in negative record numbers.
- We have added (temporary) support for reading field names from DBC files for the DBFVFP driver. As a result CDX files which use the long field names in index expressions are now also opened correctly
- Fixed a problem in the CopyDb() code for the DBF RDD
- The DBFCDX RDD now implements the BLOB_GET and also BlobExport() and BlobImport()
- Packing, Zapping or Rebuilding a CDX index with a Custom or Unique flag would not keep these flags. This has been fixed.
- When you create a file with the DBFVFP driver you can now include Field Flags in the field type of the DbCreate() array by following the type with a colon and one or more flags, where flags is one of:
  N or 0:   Nullable
  B   Binary
  +   AutoIncrement
  U   Unicode. (not supported by FoxPro)
  Other flags may follow (for example Harbour also has E = Encrypted and C = Compressed)
  Note:
  o Please note that the size of the field is the # of bytes, so {"NAME","C:U",20,0} declares a Unicode character field of 10 Unicode characters and 20 bytes.
  o We do not validate combinations of flags. For example AutoIncrement is only working for fields of type Integer.
- DbFieldInfo(DBS_PROPERTIES) returns 5 for all RDDs with the exception of the DBFVFP driver. That driver returns 6. The 6th property is the FLAGS field. This field is a combination of the DBFFieldFlags enum values.
- Fixed a problem with AppendDb() and CopyDb() for the Advantage RDD
- Fixed a problem in the Append() code of the DBF RDD. When Append() was called and no data was written then the record that was written to disk could be corrupted. The Append() method now directly writes the new record with blanks..
- The Fully qualified names of the Advantage RDDs inside XSharp.RDD.DLL are now the same as in the AdvantageRDD.DLL for Vulcan.
- We have added a FileCommit event to the notifications. This sent when a workarea is committed.

Macro compiler

- The macro compiler now also recognizes the Array(), Date() and DateTime() functions.
- Fixed problems with Aliased expressions
- On the place where the macro compiler expects a single expression you can now also have an expression list between parentheses. The last expression in the list is seen as the return value of the expression list

Visual Studio integration

- The option to compile case sensitive has been enabled in the VS project system
- The speed of 'Format Document' has improved a lot.
- The XSharp Intellisense Optionspage in Tools/Options now has a scroll bar when needed
- The ToolPalette in the VO Window editor now has icons
- We have added templates for VO MDI windows and VO SDI windows.

Build System

- When compiling native resources the resource compiler now automatically includes a file with some defines such as VS_VERSION_INFO

Debugger

- When you enter a watch expression in the debugger or a breakpoint condition, you can now use 1 based array indices. Our debugger will now automatically subtract 1 when evaluating the expression.

VOXporter

- Fixed a problem in the Windows Forms code generation
- You can now also export single MEF files, single PRG files and data from the Clipboard.
- Code between #ifdef .. #endif is not touched by VOXPorter

Changes in 2.2.1.0 (Bandol GA 2.2a)

Compiler

- When compiling code that contained an assign and not an access then trying to read the access could lead to a compiler exception. This has been fixed.

Runtime

- Added a missing _Run() function

Visual Studio integration / Build system

- Fixed a problem that caused a dialog to be shown with the message "The 'XSharp Project System' package did not load correctly."
- Fixed a problem with writing response files for the resource compiler when the source file names contained ASCII characters with accents or other characters > 128. Even though this is now fixed we still recommend not go to crazy with file names, because these names have to be converted from Unicode to Ansi, since the resource compiler can only read response files in Ansi format.
- Fixed a problem for certain QuickInfo / Tooltip windows
- The VO item templates now have a condition around the #include statements for the Vulcan include files, since these are no longer needed when compiling for the X# runtime.
- Added Support for the "Auto" window in the debugger
- Expressions in the Watch window, Breakpoint conditions etc may now contain SELF, SUPER and a colon separator. Unfortunately they are still case sensitive.

VOXPorter

- we now detect that a class has fieldnames and accesses/assigns with the same name. This was allowed in VO but no longer in .Net. The field names will be prefixed with an underscore inside the class.
- We now prefix the name "Trace" with @@ because this is quite often used to conditional compile tracing code in VS.

Changes in 2.2.0.0 (Bandol GA 2.2)

Compiler

- The compiler now recognizes the functions Date(), DateTime() and Array(), even though their names are the same as type names.
  Date() with 1 parameter will still be seen as a cast from that parameter to a Date(), like in the following example
  ```
  LOCAL dwJulianDate    AS DWORD
  LOCAL dJulianDate     AS DATE
  dwJulianDate        := DWORD( 1901.01.01)
  dJulianDate         := DATE(dwJulianDate) // This is still a cast
  from Date to DWORD
  ```
  However when Date is called with 0 or 3 parameters then either the current date is returned (like with Today()) or a date is constructed from the 3 parameters (like in ConDate())
  The DateTime() function takes 3 or more parameters and constructs a DateTime() value.
  The Array() function takes the same parameters as the ArrayNew() function.

X#

- When choosing overloads for String.Format() and a usual expression is passed as first reference we no longer allow the compiler to choose one of the overloads that expects an IFormatProvider interface.
- Parameters passed by reference to untyped methods/functions now have the IsByRef flag set. You can query for "By Reference" parameters by checking the parameter with IsByRef(uParameter). Please note that after assigning a new value to a parameter, this flag will be cleared.
- The compiler now also allows to pass aliased fields and memvars by reference to untyped functions. Even undeclared memvars are allowed.
  Please note that the assignment back to the field and memvar will happen after the call to the function returns. So inside the function the field or memvar will still have its original value.
- Using ':' as send operator in Interpolated strings is ambiguous because ':' is also used add format specifiers to interpolated strings. The compiler now detects and allows "SELF:", "SUPER:" and "THIS:".
  If you want to be safe use the '.' as send operator inside interpolated strings for other variables, or simply don't use interpolated strings, but use String.Format like in:
  ? String.Format("{0} {1}", oObject:Property1, oObject:Property2)
  in stead of
  ? i"{oObject:Property1} {oObject:Property2}"
  This is the code that the compiler will produce anyway

Macrocompiler

- The macro compiler now recognizes and compiles nested codeblocks, such as
  LOCAL cb := {|e| **IIF**(e, {||SomeFunc()}, {||SomeOtherFunc}) } **AS CODEBLOCK**
  cb := Eval(cb, TRUE)   *// cb will now contain {||SomeFunc()}*
  ? Eval(cb)
- In the FoxPro dialect the macro compiler now recognizes AND, OR, NOT and XOR as logical operators

Runtime

- Added some Xbase++ compatible functions, such as DbCargo(), DbDescend() and DbSetDescend().
- The DateCountry Enum now also the values System and Windows, which both read the date format from the Regional settings in the System.
- We have added a WrapperRDD class that you can inherit from. This allows you to wrap an existing RDD and subclass methods of your choice. See the documentation of WrapperRDD for an example.
- We had added a XPP member to the CollationMode enum with the same number as Clipper. This was confusing to some users. We have now give the XPP member a new number.
- OleAutoObject:NoMethod() now behaves different in the Vulcan dialect (to be compatible with Vulcan). In the Vulcan dialect the method name is inserted at the beginning of the list of arguments. In the other dialects the arguments are unchanged, and you need to call the NoMethod() function to retrieve the name of the method that was originally called.

- All settings in the runtime state are now initialized with a default value, so the Settings() dictionary in the runtimestate will have values for all Set enum values.
- The previous change has fixed a problem with the Set() function when setting values for logical settings with a string "On" or "Off". Because some settings were not initialized with a logic this was not working.
- When creating indexes with SetCollation(#Ordinal) the speed is a bit better now.
- The runtimestate now has a setting EOF. When this is TRUE (which is done automatically for the FoxPro dialect) then MemoWrit() will write a ^Z (chr(26)) after a text file, and MemoRead() will remove that character when it finds it.
- The runtimestate now has a setting EOL. This defaults to CR - LF (chr(13+chr(10)). This setting is used for line delimiters when writing files with FWriteLine().

RDD system

- Fixed locking problems in the DBFCDX RDD that were causing problems when opening files shared between multiple apps but also between multiple threads. The RDD now should properly detect that the CDX was updated by another process or thread.
- Fixed a problem with the File IO system when running multiple threads
- Fixed a problem with the File() and FPathName() functions when running multiple threads
- Added support for Workarea Cargo (See DbCargo())
- Numeric columns with trailing spaces were returned as 0. This has been fixed.
- Fixed a problem in the DBFCDX driver that was happening when many keys were deleted / updated and index pages were deleted.
- Fix a read error at EOF for the DBF RDD.

VOSDK

- Fixed a problem in the DbServer destructor when called at application shutdown for a server that was already closed.

Visual Studio integration

- Fixed speed problem in the "Brace Matching" code with the help of a user (thanks Fergus!)
- You should no longer be able to edit source code when the debugger is running.
- We have added a property "Register for COM Interop" to the build options of the Project Properties.
- We have updated the assembly info templates . They now have a GUID and Comvisible attribute.
- Empty lines in the editor buffer could sometimes trigger an exception. This has been fixed
- Text between TEXT .. ENDTEXT is no longer changed by formatting options in the editor, such as indenting or case synchronization.
- Incomplete strings will have the color of normal strings in the editor.
- QuickInfo and Completion lists will follow the "format case" setting of the editor for keywords.

- If a certain option from the Tools/Options was not set then loading a project that was saved with files open in the editor could result in an exception, causing the project to be loaded with no visible items. Unload and Reload would fix that. This will no longer happen.
- We have made some changes to make solutions open and close faster.
- Some colors were difficult to read when the Visual Studio Dark theme was selected. This has been fixed.
- Brace matching was sometimes incorrectly matching an END CLASS with the BEGIN NAMESPACE. This should no longer happen.
- Fixed an exception when opening a solution under certain circumstances which would display an error inside VS that the XSharp Project System was not loaded correctly.
- The Code Generator for Windows Forms, Settings and Resources now respect the keyword case setting from the Tools - Options TextEditor/XSharp page.

VOXPorter

- Folder names ending with a backslash could confuse VOXPorter

Changes in 2.1.1.0 (Bandol GA 2.11)

Compiler

- We have added new syntaxes for OUT parameters. You can now use one of the following syntaxes

```
LOCAL cString as STRING
cString := "12345"
IF Int32.TryParse(cString, OUT VAR result)
    // this declares the out variable inline, the type is
derived from the method call
    ? "Parsing succeeded, result is ", result
ENDIF
IF Int32.TryParse(cString, OUT result2 AS Int32)
    // this declares the out variable inline, the type is
specified by us
    ? "Parsing succeeded, result is ", result2
ENDIF
IF Int32.TryParse(cString, OUT NULL)
    // this tells the compiler to generate an out variable, we
are not interested in the result
    ? "Parsing succeeded"
ENDIF
IF Int32.TryParse(cString, OUT VAR _)
    // this tells the compiler to generate an out variable, we
are not interested in the result.
```

```
    // The name "_" has a special meaning "ignore this"
    ? "Parsing succeeded"
ENDIF
```

- The compiler now allows the function names Date(), DateTime() and Array(). The runtime has these functions (see below)
- Fixed a preprocessor problem where the <token> match marker inside UDCs was stopping matching tokens when the .not. or ! operator was found after another logical operator such as .AND. or .OR..
- Added support for <usualValue> IS <SomeType>. The compiler will automatically extract the contents of the USUAL and wrap it in an object and then apply the normal IS <SomeType> operation.
- Fixed a problem with Interpolated strings where the '/' character was not properly recognized.
- The compiler now supports the FoxPro syntax for cursor access. When dynamic memory variables are disabled this always gets translated to reading a field from the current cursor/workarea.

```
USE Customer
SCAN
    ? Customer.LastName
END SCAN
USE
```

When memory variables are enabled then this code could also mean that you are trying to read the Lastname property of a variable with the name "Customer" like in the example below:

```
USE Invoices
Customer = MyCustomerObject{}
SCAN
    ? Customer.LastName, Invoice.Total
END SCAN
USE
```

You can also use the M prefix to indicate a local variable or memory variable. The compiler will try to resolve the variable to the local first and when that fails it will try to resolve the variable to a memory variable (when dynamic memory variables are enabled).

Runtime

- We have added support functions for the FoxPro cursor access syntax.
- In the Vulcan dialect the NoMethod() method now receives the methodname as first parameter (this is NOT compatible with VO)
- Added functions Date() (can have 0 or 3 parameters, equivalent to Today() and ConDate()), DateTime() and Array().
- Added fixes and optimizations for functions such that take an area parameter such as Used(uArea) and Eof(uArea).
- AScan() and AScanExact() now return 0 when a NULL_ARRAY is passed.

RDD

- There was a problem reading negative numbers from DBFs. This has been fixed
- Fixed an exception when FPT drivers were writing data blocks in the FPT file with a 0 byte length.
- The DBF() function returns the Full filename in the FoxPro dialect and the alias in the other dialects.
- When creating an CDX index for a completely empy DBF file then an index key would be inserted for the phantom record. This has been fixed.

Changes in 2.1.0.0 (Bandol GA 2.1)

Compiler

- We have added support for parameters by reference to function and method calls for untyped parameters
- In the Xbase++ and FoxPro dialect arguments passed with '@' are always treated as BY REF arguments because these dialects do not support the 'AddressOf' functionality
- When /undeclared was used and an entity added a new private then this private was not cleared when the entity went out of scope. This has been fixed.
- Compiling oObject?:Variable was not handled correctly by the compiler
- Fixed an internal compiler error when calling SELF:Axit()
- Parameters for the DO statement are now passed by reference
- Changed the order of 'necessary' assembly names when compiling for not core dialect.
- We have added support for several SET commands, such as SET DEFAULT, SET PATH, SET DATE, SET EXACT etc.

Runtime

- We have made some changes to get XSharp.Core to run on Linux
- We have fixed a problem in the Subtract operator for the Date type. This changes the signature of the Subtract operator which has forced us to increase the Assemblyversion of the runtime.
- The Xbase++ dialect now allows the [] operator on a string inside a usual. This returns a substring of 1 character for the given position.
- We have fixed an incorrect event for the OrderChanged event
- CoreDb.BuffRefresh was sending an incorrect enumerator value to the IRDD.RecInfo() method.
- The IVarList() function was including protected Fields and Properties. This has been fixed.
- IsInstanceOfUsual() could not be used if an objects was of a subclass of CodeBlock. This has now been fixed.

- We have added many overloads of workarea related functions with an extra parameter to indicate a workarea number or workarea name. For example for the EoF(), Recno(), Found() and Deleted() functions
- We have added Xbase++ collation tables. The SetCollationTable() function now selects the right collation.
- Several Array related functions now have better checks for NULL arrays
- The SubcodeText property in the error class is now Read/Write. When the value has not been written then the subcode number is used to lookup the value of the property.
- MExec() was not always evaluating the compiled codeblock. This has been fixed.
- We have added some missing Goniometric functions, such as ACos(), ASin() and more.
- In the Xbase++ dialect the FieldGet() and FieldPut() functions no longer throw an error for incorrect field numbers
- We have added a missing MakeShort() function and SEvalA() function.
- The DateCountry settings now include a System setting which will read the date format from the settings for the current culture.

Macrocompiler

- When the macro compiler detects an ambiguous method or constructor it now includes the signatures of these in the error message
- We have added a new IMacroCompiler2 interface that adds an extra property "Resolver". This property will may receive a Delegate of type "MacroCompilerResolveAmbiguousMatch". This delegate has the following prototype:
  ```
  DELEGATE MacroCompilerResolveAmbiguousMatch(m1 as MemberInfo, m2
  as MemberInfo, args as System.Type[]) AS LONG
  ```
- The delegate will be called when the macro compiler detects an ambiguous match and receives the System.Reflection.MemberInfo for possible candidates and an array of the detected types of the arguments (detected at compile time). The delegate can return 1 or 2 to choose between either candidate. Any other value means that the delegate does not know which of the ambiguous members to choose.
  If the macro compiler finds more than 2 alternatives, it first calls the delegate with alternatives 1 & 2, and then the selected delegate from these 2 and alternative 3 etc.
- You can register a function or method as delegate with the new function `SetMacroDuplicatesResolver()`
- We are now handling (one level of) nested Macros. So the macro compiler correctly compiles a codeblock like
  ```
  {|e| iif(e, {||TRUE}, {||FALSE})}
  ```
- The macrocompiler now allows comparisons between Integers and Logics (just like the Usual type in the runtime). This is still not recommended !
- The macrocompiler now allows the use of '[' and ']' as string delimiters. This is NOT allowed in the normal compiler because these delimiters will be impossible to differentiate from attributes.
- We have fixed a problem when a late bound call was needed for method names that were matching method names or property names in the Usual type (such as a method with the name Item()).

- PCount() for macro compiled codeblocks was always returning 1. This has been fixed.

VOSDK

- Fixes a problem with DbServer objects that were not closed in code.
  The existing code was trying to close the workarea from the destructor. But in .Net the destructor runs in a separate thread and in that GC Thread there where no files open...
- Removed unneeded calls to DbfDebug()
- The AdsSqlServer class is now added to the VORDDClasses assembly

RDD

- We have fixed a problem with parsing incorrect or empty dates
- We have fixed a problem with reading Dates in the Advantage RDD that could cause a Heap error when reading dates.
- We have added several 'missing' functions for Advantage support that were in the 'Ace.Aef' for VO
- We have added support for Character fields > 255 characters
- DbSetScope() now moves the record pointer to the first record that matches the new scope.
- DbCreate() for the DBFNTX driver with SetAnsi(TRUE) was creating a file with a first byte of 0x07 (or 0x87) .
  This no longer happens in the Xbase++, FoxPro and Harbour dialects because this first byte is VO specific only
- Some FoxPro memo values are written with an extra 0 byte at the end. This extra byte is now suppressed when reading these values.
- We have fixed a problem with the version numbers in CDX files not being updated and also improved CDX locking.
- Xbase++ was not recognizing NTX indices when the tag name in the index header was not in uppercase. This has been fixed.
- We have fixed a (performance and size) problem when creating CDX indexes.
- When opening a DBF file that does not have a codepage byte, we default to the current Windows or DOS codepage, depending on the current SetAnsi() setting.
- Optimized reading numeric, date and logical columns
-

Visual Studio integration

- The WCF Service template has been fixed
- We have migrated the project system to the Asynchronous API. This should make loading of solutions with a large number of X# projects a bit faster.
- Fixed a problem in the Keyword Case synchronization that could lock up the UI for several seconds
- Fixed an exception in the BraceMatching code.
- Uncommenting a block of lines was sometimes leaving the comments in front of empty lines. This has been fixed.
- We have improved the (XML) documentation lookup for types, methods, fields, properties and parameters.

- We have improved the type lookup between X# projects.

VOXPorter

- DbServer and FieldSpec entities are now also exported
- VOXPorter now also can genarate a separate project/application that contains Windows Forms versions of the VO GUI windows found in the VO Applications.
- When running VOXPorter you now can choose to export to XIDE, Visual Studio or Both.

Changes in 2.0.8.1 (Bandol GA 2.08a)

Compiler

- Fixed a recursion problem in the preprocessor
- MEMVAR-> and FIELD-> were no longer correcty detected This has been fixed.
- We have fixed several issues in dbcmd.xh
- Fixed a problem with return statements inside Lambda expressions.
- The = Expression() statements (FoxPro dialect) was not generating any code. This has been fixed.

Runtime

- XPP.Abstract.NoMethod() and XPP.DataObject.NoMethod() were still expecting the method name as 1st parameter.This has been fixed.
- StretchBitmap() was doing the same as ShowBitmap() because of an incorrect parameter. This has been fixed.

Visual Studio integration

- Improved the Format-Document code
- Fixed a problem in the VS Parser when looking up the type for variables defined with the VAR keyword which could send VS in an endless loop.
- The contents of the TEXT .. ENDTEXT block and the line after the \ and \\ tokens now has its own color

Changes in 2.0.8 (Bandol GA 2.08)

Compiler

- The compiler had a problem with the "return" attribute target
- Errors inside the "statementblock" rule are now better detected and the compiler will no longer report many errors after these for correct lines of code.
- Fixed a problem with Casts to logic
- Fixed a problem with undeclared variables used as counter for For Loops
- Improved the code generation for FIELDs, MEMVARs and undeclared variables for prefix operation, postfix operations and assignments.
- Improved the code generation for method calls where the parameter is a ref or out variable when default parameters are involved. The compiler now generates extra temporary variables for these calls.
- In the dialects where this relevant the compiler now also supports ENDFOR as alias for NEXT and FOR EACH as alias for FOREACH.
- Added support for the DO <proc> [WITH arguments] syntax

Runtime

- The DbCreate() function now creates a unique alias when the base filename of the file to create is already opened as an alias
- The Numeric overflow checking for USUAL values now follows the overflow checks of the main app
- DbUnLock() now accepts an (optional) record number as parameter
- XMLGetChild() was throwing an exception when no elements were found
- XMLGetChildren() was throwing an exception
- Fixed a problem in 2 rules inside "dbcmds.xh"
- The XSharpDefs.xh file now automatically includes "dbcmd.xh"
- Some datatype errors were reported incorrectly.
- The "NoMethod" method for late bound code was called with incorrect parameters. This has been fixed.
- Fixed some problems with translating usuals with a NIL value to string or object.
- In Xbase++ the Set() function also accepts strings with the value "ON" or "OFF" for logical settings. We are now allowing this too.
- Set(_SET_AUTOORDER) now accepts a numeric second parameter just like in VO (Vulcan was using a Logic parameter)
- We have added some support classes to the FoxPro class hierarchy for the FoxPro class support (Abstract, Custom and Collection). More classes will follow later.
- Fixed a problem with transform and "@ez" picture.

VOSDK

- Fixed a problem in the SQLSelect class when re-opening a cursor.

RDD System

- Fixed a problem reading Advantage MEMO fields
- Improved the error messages when an index cannot be opened due to an index expression with an error (for example a missing function)
- We have added the option to install an event handler in the RDD system. See the topic Workarea Events for more information.
- Skip, Gobottom and other workarea operations that change the current record will no longer set EOF to FALSE for workareas with 0 records.
- Clearing the scope in an Advantage workarea would throw an exception when there was no scope set. This has been fixed.
- Unlocking a record in an Advantage workarea would throw an exception when there was no record locked. This has been fixed.
- DbSetRelation() was not working correctly. This has been fixed.

VS Integration

- Fixed a problem with the code generation for DbServer and FieldSpec entities
- Added support for the Import and Export buttons in the DbServer Editor
- Improved entity parsing inside the editor in the Xbase++ dialect.
- The VS Parser was not colorizing the UDC tokens (including ENDFOR) unless the source file had preprocessor tokens itself. This has been fixed.
- Improved block detection for new END keywords.
- The VS Integration now recognized the class syntax for VFP type classes.
- Fixed a problem in the code that was checking to see which project system "owns" the PRG extension.
- Added compiler option to the Project Property pages to suppress generating a default Win32 manifest.

VOXporter

- VOXPorter was ignoring entities that were not properly prototyped in VO. This has been fixed

FoxPro dialect

- We have added a compiler option /fox1 that controls the class hierarchy for objects. With /fox1 enabled (the default in the FoxPro dialect) all classes must inherit from the Custom class. The code generation for properties stores the values for properties in a collection inside the Custom class. With /fox1- properties will be generated as "auto" properties with a backing field.
- We have added support for FoxPro classes. See the topic FoxPro class syntax for more information about what works and what doesn't work.

- We have added support for DIMENSION and DECLARE statements (which create a MEMVAR initialized with an array)

Changes in 2.0.7 (Bandol GA 2.07)

Possible breaking change

- We have removed the #define CRLF from the standard header file. There is a DEFINE CRLF in XSharp.Core now. If you are compiling against Vulcan and you are seeing an error about a missing CRLF then you may want to add the following to your code:
  ```
  DEFINE CRLF := e"\r\n"
  ```

Compiler

- UDCs that were resulting in an empty list of tokens were triggering a compiler error in the preprocessor. This has been fixed.
- Calling a method on an array would be translated to a ASend() with the method name as parameter when the method does not exist in the underlying array class. The compiler will generate a warning now when this happens,.
- The compiler was producing incorrect code for (USUAL) casts. This has been fixed. In rare cases this may produce a compilation error. If that happens to you then simply create a usual by calling the USUAL constructor: USUAL{somevalue}
- Fixed several problems with methods declared outside of a CLASS .. END CLASS
- In the FoxPro dialect NOT, AND, OR and XOR are now allowed as alternate syntax for .NOT.,.AND., .OR. and .XOR.
- In the FoxPro dialect you can now include statements before the first entity in the file. The compiler will recognize these and will automatically create a function with the name of the source file and will add the code in these statements a body of this function.
- The compiler now allows to cast an integer expression to logic when /vo7 is enabled. The LOGIC(_CAST is always supported for expressions of type integer
- Incorrect use of language features (such as using a VOSTRUCT in the Core or FoxPro dialect) is now detected earlier by the compiler leading to somewhat faster compile times for incorrect code.
- The compiler now also initialized multi dimensional string arrays with an empty string when /vo2 is enabled, like in the code below:
  ```
  CLASS TestClass
      EXPORT DIM aDim[3,3] AS STRING
  END CLASS
  ```
- In previous builds you could not set breakpoints on the source code line with a SELF() or SUPER() call if this line was immediately after the CONSTRUCTOR(). This has been fixed.
- When a project contains "_DLL METHOD", "_DLL ASSIGN" or "_DLL ACCESS" (after exporting from VO) then the compiler will now generate a more meaningful errormessage.

- The compiler will no longer produce hundreds of the same error messages when a source file contains many of the same error. After 10 errors per source file the compiler will only report unique error numbers. So if your source code has 20 different error messages then you will still see 20 errors reported, but if your source contains the same error type 100 times then the list will be truncated after 10 errors.
- The compiler no longer allows code behind end tokens such as ENDIF or NEXT. The standard header file 'XSharpDefs.xh' now includes rules that will eliminate these tokens.

Runtime

- The ++ and -- operators for the usualtype were not working for Date and Datetime values
- FErase() and FRename() now set FError() to 2 when the source file does not exist
- The File() function was throwing an exception for paths with invalid characters. It now returns FALSE and sets the Ferror()
- Several specific numbers were producing incorrect Str() results. This has been fixed.
- The case of the name of the Value property for several types was changed from Value to VALUE. This caused problems for people that were interfacing with X# code from C# code. The original case has been restored. This change has been reversed.
- Under certain situations the error stack would not contain the complete list of frames. This has been fixed.
- The size of the Close and Copy buttons of the Error Dialog has been enlarged so there is more space for translated strings
- The Pad..() functions were returning a padded version of "NIL" for NIL values. This was not compatible with Xbase++. They now return a string with all spaces. Btw: VO throw an exception when you call Pad..() with a NIL value.
- Fixed a problem with the PadC() function for values > 1 character.
- We have changed the Val() function to be more compatible with Visual Objects
- The runtime contained a second overload for the Space() function that accepted an Int parameter. This was causing problems in the macro compiler. This overload has been removed. You may have to change your code because of that.
- Fixed a problem in EnforceType() and EmptyUsual() with the STRING type
- AEval and AEvalOld() now both pass the array index as second parameter to the codeblock that is evaluated

RDD System

- Fixed a problem that EOF and BOF were not both set to true when opening an empty DBF with an index
- Fixed a problem with DbSeek() and Found() for DBFNTX and DBFCDX
- The DBF class was not properly decoding field names and/or index expressions that contain Ascii characters > 127 (field names like STRAßE)
- File dates were updated when a dbf was closed even when nothing was changed. This has been fixed.
- The runtime now contains code that closes all open workareas at shutdown. This should help to prevent DBF or index corruption.

- The Advantage RDD was automatically doing a GoTop after the index order was changed. This no longer happens.
- The Advantage RDD now retries opening DBF and Index files a couple of times before failing.
- Fixed a small incompatibility between DBFCDX and AXDBFCDX

VS Integration

- The Core Classlibrary template had a typo in a file name which caused it not to be loaded correctly
- The code generator for the Windows Forms editor was duplicating USING statements. This has been fixed. Duplicate using statements will be deleted when a form is opened and saved in the designer.
- The compilation messages on the output window for the compile time and the number of warnings and errors is now only shown for the build verbosity normal and higher. The warnings and errors message is also shown for lower build verbosity if there are compiler errors.
- The project system will no longer update the version number in the project file if the project file was created with build 2.0.1 or later.
- Fixed a problem with setting and clearing the "Specific version" property for Assembly References.
- The default templates for the VO compatible editors are now installed in the XSharp\Templates folder and the editor uses this location as 'fallback' when you don't have templates in your project
- The Properties folder is now placed as first child in the tree of a Project, and the VO Binaries items are placed before resource items in the list of children of a source item in the tree.

VOXporter

- VOXPorter now prefixes Debug tokens with @@
- VOXPorter now removes INSTANCE declaration for properties that are also declared as ACCESS/ASSIGN
- VOXPorter now adds spaces between variable names that are delimited with .AND. or .OR.. So `"a.and.b"` becomes `"a .and. b"`

Documentation

- We have "lifted" some of the documentation of the Visual Objects runtime functions and added these to our runtime documentation. This is 'work in progress', some topics will need some extra work.

Changes in 2.0.6.0 (Bandol GA 2.06)

General

- We received a request to keep the version numbering simpler. For that reason this new build is called Bandol 2.06 and the file versions for this build are also 2.06. The assembly versions for the runtime assemblies are all 2.0, and we intend to keep those stable as long as possible, so you will not be forced to recompile code that depends on the runtime assemblies.
- Several fixes that were meant to be included in 2.0.5.0 were not included in that build. This has been corrected in 2.0 6.0

Compiler

- A missing ENDTEXT keyword now produces an error XS9086
- Unbalanced textmerge delimiters produce a warning XS9085
- The TEXT keyword in the FoxPro dialect is now only recognized when it is the first non whitespace token on a line. As a result of this you can use tokens like <text> in Preprocessor commands again.
- The VO cast operations on literal strings no longer produce a compiler warning about possible memory leaks.

Runtime

- Runtime errors in late bound code were always shown as TargetInvocationException. The true cause of the error was hidden that way. We are now unpacking the error and rethrowing the original error, including the callstack that was leading to that error
- Some texts in the string resources were updated
- Calling the Str() function with a -1 value for length and/or decimals produced results that were not compatible with VO. This was fixed.
- Fixed a problem with DBZap() and files with a DBT memo.
- In some situations EOF and BOF were not set to TRUE when opening an empty DBF file. This has been fixed.
- PSZ values with an incorrect internal pointer are now displayed as "<Invalid PSZ>(..)"

RDD System

- The code to read and write to columns in an Advantage workarea now uses separate column objects, just like the code for the DBF RDD. This makes the code a bit easier to understand and should make the code a bit faster.

VS Integration

- The text block between TEXT and ENDTEXT is now displayed in the same color as literal strings

- The VO compatible Project Item templates no longer automatically add references to your project
- Project files from version 2.01.0 and later will no longer be "touched" when opening with this version of the X# project system, since there have been no changes to the project file format since that build.

VOXporter

- The CATCH block in the generated Start function now calls ErrorDialog() to show the errors. This uses the new language resources to display the full error with VO compatible error information (Gencode, Subcode etc)

Changes in 2.0.5.0 (Bandol GA 2.01)

Compiler

- Blank lines after an END PROPERTY could confuse the compiler. This has been fixed
- The TEXT .. ENDTEXT command has been implemented in the compiler (FoxPro dialect only)
- The \ and \\ commands have been implemented (FoxPro dialect only)
- Procedures in the FoxPro dialect may now return values. Also the /vo9 options is now enabled by default in the FoxPro dialect. The default return value for a FUNCTION and PROCEDURE is now TRUE in the foxpro dialect and NIL in the other dialects.
- Error messages no longer refer to Xbase types by their internal names (XSharp.__Usual) but by their normal name (USUAL).

MacroCompiler

- Creating classes with a namespace prefix was not working. This has been fixed.

Runtime

- Fixed a problem with ArrayNew() and multiple dimensions
- When calling constructor of the Array class with a number the elements were already initialized. This was not compatible with Vulcan.NET. There is now an extra constructor whtich takes a logical parameter IFill which can be used to automatically fill the array
- The text for the ERROR_STACK language resource has been updated
- Calling Str() with integer numbers was returning a slightly different result from VO. This has been fixed.
- Added support functions for TEXT .. ENDTEXT and TextMerge and an output text file.
- Fixed a problem in the DTOC() function
- You can now add multiple ImplicitNamespace attributes to an assembly

- We have added several FoxPro system variables (only _TEXT does something at this moment)

RDDs

- Zap and Pack operations were not properly setting the DBF file size
- An Append() in shared mode was not properly setting the RecCount
- Opening a file with one of the Advantage SQL RDDs was not working. This has been fixed.
- Writing DateTime.Minvalue to a DBF would not write an empty date but the date 1.1.1 This has been fixed.

VO SDK

- Fixed a problem in ListView:EnsureVisible().
- Some questionable casts (such as the one that cause the previous problem) have been cleaned up

Visual Studio Integration

- Parameter tips for constructor calls were off by one parameter. This has been fixed.
- When looking for types, the XSharp namespace is now the first namespace that is searched.

Changes in 2.0.4.0 (Bandol GA)

Compiler

- Fix a problem in assignment expressions where the Left side is an aliased expression with a workarea in parentheses:
  `(nArea)->LastName := AnotherArea->LastName`
- Multiline statements, such as FOR blocks, no longer generate Multiline breakpoints in the debugger.
- Fixed a problem where blank lines or lines with 'inactive' preprocessor comments after a class definition would generate a compiler error.
- Errors for implicit conversions between INT/DWORD and PTR now produce a better error message when they are not supported.
- USUAL.ToObject() could not be called with the latebinding compiler option was enabled. This has been fixed.
- Fixed an internal compiler error with untyped STATIC LOCALs.
- Fixed a problem with aliased expressions.
- Indexing PSZ values is no longer affected by the /az compiler option

MacroCompiler

- Fixed a problem with some aliased expressions
- The macro compiler now detects that you are overriding a built-in function in your own code and will no longer throw an "ambigous method" exception but will choose function from your code over functions defined in the X# runtime

Runtime

- FIxed several problems in the Directory() function
- Fixed problem with indexing PSZ values
- Added StackTrace property on the Error object so also errors caught in a BEGIN SEQUENCE will have stack information.
- Fixed problems with "special" float values and ToString(), such as NaN, PositiveInfinity
- Fixed a problem with RddSetDefault() with a null parameter
- DbInfo(DBI_RDD_LIST) was not returning a value. This has been fixed.
- We have updated many of the language resources, Also the Error:ToString() now uses the language resources for captions like 'Arguments' and 'Description'.
- Low level file errors now include the callstack
- Fixed some problems in AsHexString()
- The DosErrString() no longer gets its messages from the language string tables. The messages have been removed and also the related members in the XSharp.VOErrors enum.
- Added a Turkish language resource.

RDD System

- Fix locking problem in FPT files
- Fixed several problems with OrdKeyCount() and filters, scopes and SetDeleted() setting
- Some DBF files have a value in the Decimals byte for field definitions for field types that do not support decimals. This was causing problems. These decimals are now ignored.
- Opening and closing a DBF without making changes was updating the time stamp. This has been fixed.
- Fixed problems in Pack() and Zap()
- Fixed a problem where custom indexes were accidentally updated.
- Fixed several problems with OrdKeyCount() in combination with Filters, SetDeleted() and scopes.

VO SDK Classes

- Most of the libraries now compile with "Late Binding" disabled for better performance.
  To help in doing this some typed properties have been added such as SqlStatement:__Connection which is typed as SQLConnection.

Visual Studio integration

- Fixed a problem in the Brace matching code
- Improved Brace matching for keywords. Several BEGIN .. END constructs have now been included as well as CASE statements inside DO CASE and SWITCH, RECOVER, FINALLY, ELSE, ELSEIF and OTHERWISE
- Fix a problem with adding and deleting references when unloaded or unavailable references existed.

VOXPorter

- The program is now able to comment, uncomment and delete source code lines from the VO code when exporting to XSharp.
  You have to add comments at the end of the line. The following comments are supported:

```
// VXP-COM : comments the line when exporting it
// VXP-UNC : uncomments the line
// VXP-DEL : deletes the line contents

example:
// METHOD ThisMethodDoesNotGetDefinedInVOcode() // VXP-UNC
// RETURN NIL // VXP-UNC
```

Changes in 2.0.3.0 (Bandol RC3)

Compiler

- Code generation for STATIC LOCALs of type STRING was not initializing the variables to an empty string when /vo2 was selected. We have also improved code generation for STATIC LOCALs when they are initialized with a compile time constant
- In preparation for the support for variables passed by reference to functions/methods with clipper calling convention we are now assigning back the locals variables to the parameter array at the end of a function/method with clipper calling convention.
- The compiler would not complain if you were assigning a value of one enum to a variable of another enum. This has been fixed.
- Added support for the FoxPro '=' assignment operators. Other dialects also allow the assignment operator but a warning is generated in the other dialects.
- Xbase++ classes inside BEGIN NAMESPACE .. END NAMESPACE were not recognized. This has been fixed.
- Statements inside WITH blocks are no longer constrained to assignment expressions and method calls. You can now use the WITH syntax for expressions anywhere inside a WITH block. If the compiler can't find the WITH variable then it will output a new error message (XS9082)

- Updated the Aliased Expression rules to make sure that compound expressions properly respect the parentheses.
- The __DEBUG__ macro was not always set correctly. We have changed the algorithm that sets this macro. When the DEBUG define is set then this macro gets defined. When the NDEBUG define is set then this macro is not defined. When both defines are absent then __DEBUG__ is NOT set.
- The compiler was allowing you to use the '+' operator between variables/ expressions of type string and logic. This is now flagged as an error.

MacroCompiler

- Fixed a problem with resolving Field names that were identical to keywords or keyword abbreviations (for example DATE and CODE) and for Field names that are equal to built-in function names (such as SET)
- Fixed a problem where a complicated expression evaluated with an alias prefix was not evaluated correctly.
- The macro compiler initializes itself from the Dialect option in the runtime to enable/disable certain behavior.
- The macro compiler now recognizes the "." operator for workarea access and memvar access when running in the FoxPro dialect.

Runtime

- Added functions FieldPutBytes() and FieldGetBytes()
- Added function ShowArray()
- Added several defines that were missing, such as MAX_ALLOC and ASC_A.
- Added Crypt() overloads that accept BYTE[] arguments
- The ClassDescribe() method for DataObject classes (XPP dialect) now includes properties and methods that were dynamically added.
- Fixed a problem with the RELEASE command for MemVars. This was also releasing variables defined outside the current function / method.
- There is now also a difference between the FoxPro dialect and other dialects in the behavior of the RELEASE command.
  FoxPro completely deletes the variables, the other dialect set the value of the variables to NIL.
- New PRIVATE memvars are initialized to FALSE in the FoxPro dialect. In the other dialects they are initialized to NIL.
- Some numeric properties in the RuntimeState were giving a problem when a numeric of one type was written and another numeric type was expected when reading. This has been fixed.
- Fixed a problem with return NIL values from Macro compiled codeblocks.
- The parameter to DbClearScope() is now optional
- The USUAL type now allows to compare between values of type PTR and LONG/INT64 The PTR value is converted to the appropriate Integral type and then an Integral comparison is done.
- The USUAL type now also allows comparisons between any type and NIL.
- Casts from USUAL values to SHORT, WORD, BYTE and SBYTE are no longer checked to be compatible with VO.

RDD System

- Added support for different block sizes in DBFFPT.
- DBFFPT now allows to override the block size (when creating) from the users code. Please note that block sizes < 32 bytes prevent the FPT from opening in Visual FoxPro.
- Added support for reading various Flexfile memo field types, including arrays.
- Added support for writing to FPT files
- When creating FPT files we now also write the FlexFile header. Please note that our FPT driver does not support "record recycling" for deleted blocks like FlexFile does. We also only support writing STRING values to FPT files and Byte[] values.
- Added support for Visual FoxPro created CDX files that were created with the COLLATE option. The RDD dll now contains collation tables for all possible combinations of collation and CodePage.
- Added support for USUALs with a NIL value and the comparison operators (>, >=, <, <=). These operators return FALSE, except the >= and <= operators which return TRUE when both sides of the comparison are NIL.
- We exposed several Advantage related function and types. Also the function AdsConnect60() was defined. We have not created functions for all available functions in Ace32 and Ace64, but only the ones needed in the RDD.
- If you are missing a function in the ACE class, please let us know. All functions should be available and accessible now in the Ace32 and Ace64 classes or in the ACEUNPUB32 or ACEUNPUB64 classes.
- The ADS RDD was returning incorrect values for LOGIC fields.
- Fixed some problems with skipping in CDX indexes and scopes and filters.
- Executing DbGoTop() twice or DbGoBottom() twice for DBFCDX would confuse the RDD. This has been fixed.
- Fixed a problem with Seeking() in an empty DBF file
- FieldPut for STRING fields in the Advantage RDD now truncates the fields to the maximum length of the field before assigning the value
- Fixed a problem with UNIQUE CDX Indexes.
- You can now create VFP compatible DBF files with DBCreate(). To do so use the following field types (apart from the normal CDLMN):

W	Blob
Y	Currency
B	Double
T	DateTime
F	Float
G	General
I	Integer
P	Picture
Q	Varbinary
V	Varchar

  Special field flags can be indicated by adding a suffix to the type:
  "0" = Nullable
  "B" = Binary
  "+" = AutoIncrement
  So this creates a nullable date: "D0" and this creates an autoincremental integer "I+".

Auto increment columns are initialized with a counter that starts with 1 and a step size of 1. You can change that by calling DbFieldInfo:

```
DbFieldInfo(DBS_COUNTER, 1, 100) // sets the counter
for field 1 to 100
DbFieldInfo(DBS_STEP, 1, 2)      // sets the step
size for field 1 to 2
```

- Fixed a locking problem with FPT files opened in shared mode
- Fixed several problems related to OrderKeyCount() and various settings of Scopes and SetDeleted() in the DBFCDX RDD.

VO SDK Classes

- Fixed a problem in the DateTimePicker class when assigning only a time value.
- System classes and RDD classes have been cleaned up somewhat and now compile in AnyCPU mode. So this means that you can use the DbServer class in a 64 bit program !
  The projects for these two libraries also no longer have the "Late Binding" compiler option enabled. There is still some late bound code in these libraries but this code now uses explicit late bound calls such as Send(), IVarGet() and IVarPut().
- Because of the change in the handling of __DEBUG__ some SDK assemblies are not better optimized.

Visual Studio integration

- Added support for WITH .. END WITH blocks in the editor
- When generating Native Resources (RC files) the BuildSystem now sets a #define __VERSION__. This will have the fileversion number of the XSharp.Build.DLL without the dots. (2.1.0.0 will be written as "2100")
- The XSharp help item in the VS Help menu now opens the local Help (CHM) file
- Fixed a problem in the WCF service template
- Correction to the multi line indenting for code that uses attributes
- Code generation for new Event handlers now includes a RETURN statement, even when VS does not add one to the statement list
- The intellisense option "Show completionlist after every character" has been disabled since it was having a negative impact on performance and would also insert keywords with @@ characters in front of them.
- Several changes to the code parsing for the Windows Forms editor. Comments and Regions should now be saved and regenerated as well as attributes on classes. Also code generation for images from project resources has been fixed as well as parsing of static fields and enumerators declared in the same assembly.
  Please note. If you are using values from types defined in the same assembly as the form then the assembly needs to be (re)compiled first before the form can be successfully opened in the Windows Forms Editor.
- New methods generated from the Windows forms editors will now be generated with a closing RETURN statement.
- We have made some improvements to the presentation of QuickInfo in the source code editor.

Tools

- VOXporter now also exports VERSIONINFO resources

Changes in 2.0.2.0 (Bandol RC 2)

Compiler

- File wide PUBLIC declarations (for MEMVARs) were incorrectly parsed as GLOBALs. Therefore they were initialized with NIL and not with FALSE. They are now generated correctly as public Memvars. The creation of the memvars and the initialization is done in after the Init3 procedures in the assembly have run.
- Instance variable initializers now can refer other fields and are allowed to use the SELF keyword. This is still not recommended. The order in which fields are initialized is the order in which they are found in the source code. So make sure the field initializers are defined in the right order in your code.
- AUTO properties are now also initialized with an empty string when /vo2 is enabled.
- The compiler was allowing you to define instance variables for Interfaces. They were ignored during code generation. Now an error message is produced when the compiler detects fields on interfaces.
- When the compiler detects 2 ambiguous symbols with different types (for example a LOCAL and a CLASS with the same name) then the error message now clearly indicates the type for each of these symbols.
- Fixed an exception in the Preprocessor
- Added support for the FoxPro runtime DLL.
- The ANY keyword (an alias for USUAL) is no longer supported.
- Keywords that appear after a COLON (":") DOT (".") or ALIAS (->) operator are no longer parsed as keyword but as identifier. This should solve issues with parsing code that for example accesses the Date property of a DateTime class.
- We have added support for the **WITH .. END WITH** statement block:

```
LOCAL oPerson as Person
oPerson := Person{}
WITH oPerson
    :FirstName := "John"
    :LastName := "Doe"
    :Speak()
END WITH
```
You can also use the DOT (.) as prefix for the names. The only expressions allowed inside WITH .. ENDWITH are assignments and method calls (like you can see above)
- Added support for the FoxPro LPARAMETERS statement. Please not that a function or procedure can only have a PARAMETERS keyword OR a

LPARAMETERS keyword OR declared parameters (names between parentheses on the FUNCTION/PROCEDURE line)

- Added support for the FoxPro THIS keyword and .NULL. keyword
- We have added support for the FoxPro Date Literal format {^2019-06-21} and FoxPro DateTime Literals {^2019-06-21 23:59:59}.
- Date literals and DateTime literals are now also supported in the Core dialect. Date Literals will be represented as DateTime values in the Core dialect.
- The standard header file xsharpdefs.xh now conditionally includes header files for the Xbase++ dialect and FoxPro dialect. These header files do not have much content at this moment, but that will change in the coming months.
- When the compiler detects that some header files are included but that the defines in these header files are also available as constants in references assemblies then a warning will be generated and the include file will be skipped (XS9081)
- The compiler now supports an implicit function _ARGS(). This will be resolved to the arguments array that is passed to functions/methods with clipper calling convention. This can be used to pass all the arguments of a function/method to another function/method.
- We have added the TEXT ... ENDTEXT command for the FoxPro dialect. The string inbetween the TEXT and ENDTEXT lines is passed to a special runtime function __TextSupport that will receive 5 parameters: the string, the merge, NoShow, Flags and Pretext arguments. You will have to define this function yourself for now. it will be included in the XSharp Foxpro runtime in a future version.
- We have added support for END keywords for all entity types that did not have one yet. The new end keywords are **optional**. They are listed in the table below. The FoxPro ENDPROC and ENDFUNC keywords will be mapped to END PROCEDURE and END FUNCTION with a UDC.

Start	End
PROCEDURE	END PROCEDURE
PROC	END PROC
FUNCTION	END FUNCTION
FUNC	END FUNC
METHOD	END METHOD
ASSIGN	END ASSIGN
ACCESS	END ACCESS
VOSTRUCT	END VOSTRUCT
UNION	END UNION

- The compiler now registers the Dialect of the main in the Dialect property of the RuntimeState (Non Core dialects only)

MacroCompiler

- Fixed a problem with escaped literal strings
- Fixed a problem with implicit narrowing conversions
- Fixed a problem with macro compiled alias operations (Customer)->&fieldName

Runtime

- Fixed a problem in the Round() function.
- Fixed a problem in the ExecName() function.
- Added FoxPro runtime DLL.
- Added XML support functions in the Xbase++ dialect runtime
- Added support for dynamic class creation in the Xbase++ dialect runtime.
- Fixed a problem in the Push-Pop workarea code for aliased expressions.
- converting a NULL to a symbol would cause an exception. This has been fixed.

RDD system

- Fixed several problems in the ADS RDD
- The DBFCDX RDD is now included
- The DBFVFP RDD is now included. This RDD can be used to access files with DBF/FPT/CDX extension and support the Visual Foxpro field types, such as Integer, Double, DateTime and VarChar. Reading files should be fully supported. Writing should also work with the exception of the Picture and General formats and with the exception of the AutoIncremental Integer fields. You can also use the RDD to open the various "definition" files from VFP such as projects, forms and reports. The RDD 'knows' about the different extensions for indexes and memos. You can also open DBC files as normal tables. In a future version we will support the VFP database functionality.

Visual Studio Integration

- You can now specify that multi line statements should indent on the 2nd and subsequent lines.
- Type lookup for functions inside a BEGIN NAMESPACE .. END NAMESPACE did not include the types in this namespace.
- Started intellisense for INLINE methods in the Xbase++ dialect
- Fixed several problems in intellisense
- Improved intellisense for VAR keywords declared in a FOREACH loop
- Several other (smaller) improvements.

Tools

- VOXporter now writes DEFINES in the RC files and no longer literal values.
- VOXporter: fix for module names with invalid chars for filenames

Changes in 2.0.1.0 (Bandol RC 1)

Compiler

- Added support for the so called IF Pattern Expression syntax, which consists of an IS test and an assignment to a variable, prefixed with the VAR keyword:

```
IF x is Foo VAR oFoo
    ? oFoo:DoSomething()
ENDIF
```

  The variable oFoo introduced in the expression will only be visible inside the IF statement.
  Of course you can also use the pattern on other places, such as ELSEIF blocks, CASE statements, WHILE expressions etc:

```
IF x is Foo VAR oFoo
    ? oFoo:DoSomething()
ELSEIF x is Bar VAR oBar
    ? oBar:DoSomethingElse()
ENDIF
```

- Fixed a problem with method modifiers and generic methods
- Fixed a problem with partial classes with different casing and destructors
- Fixed a problem with Interfaces and methods with CLIPPER calling convention
- The compiler now generates an error (9077) when an ACCESS or ASSIGN method has Type Parameters and/or Constraint clauses
- Fixed a problem with DEFINEs with specific binary numeric values. Also overflow checking is now always of when calculating the result of numeric operations for the values of a DEFINE.
- When a constant value was added or subtracted to a numeric value < 32 bits then the result was seen as 32 bits by the compiler. This sometimes forced you to use casts in your code. With this change that cast is no longer necessary.
- The compiler allowed you to concatenate non string values and strings and was automatically calling ToString() on the non strings. This is no longer possible. The compiler now generates an error (9078)when it detects this.
- We have added error trapping code to the compiler that should route internal errors to compiler error XS9999. If you see such an error, please let us know.
- DIM arrays of literal strings are now initialized properly.
- There was a problem when switching between dialects when using the shared compiler. It would sometimes no longer detect dialect specific keywords. This has been fixed.
- Fixed a problem where incorrect code was producing an error "Failure to emit assembly"
- Fixed a problem in code that uses _CAST to cast a 32 bits value to 16 bits

- Fixed a problem with overloaded indexed properties where the index parameter in a subclass has a different type than the index parameter in the super class.
- Changed implementation of several aliased operations (ALIAS->FIELD and (ALIAS)->(Expression))
- Changed preprocessor handling of extended strings ((<token>))
- The Roslyn code was not marking some variables as 'assigned but not read' to be compatible with the old C# compiler. We are now flagging these assignments with a warning. This may produce a lot of warnings in your code that were not detected before.
To support this we have received some requests to "open up" the support for 1 based indexes in the compiler. In the past the compiler would only allow 1 based indexing for variables of type System.Array or of the XBase ARRAY Type.
We have now added a couple of interfaces to the runtime. If your type implements one of these interfaces then the compiler will recognize this and allow you to use 1 based indexes in your code and then the compiler will automatically subtract 1 from the numeric index parameter. The XSharp ARRAY type and ARRAY OF type now also implement (one of) these interfaces/
The interfaces are:

```
INTERFACE IIndexer
    PUBLIC PROPERTY SELF[index PARAMS INT[]] AS USUAL GET SET
END INTERFACE


INTERFACE IIndexedProperties
    PROPERTY SELF[index AS INT   ] AS USUAL GET SET
    PROPERTY SELF[name  AS STRING] AS USUAL GET SET
END INTERFACE
INTERFACE INamedIndexer
    PUBLIC PROPERTY SELF[index AS INT, name AS STRING] AS USUAL
GET SET
    END INTERFACE
```

Runtime

- Fixed some problems in the OrderInfo() function
- Fixed several problems with DB..() functions in the runtime
- Fixed several problems with the macro compiler
- Fixed a problem with the handling of default parameters in late bound calls to methods
- Improved error messages for missing methods and/or properties in late bound code.
- The Select() function was changing the current workarea. This has been fixed.
- Converting a USUAL to a STRING was not throwing the same exceptions as VO. It was always calling ToString() on the USUAL. Now the behavior is the same as in VO.
- F_ERROR has been defined as a PTR now and no longer as numeric
- CreateInstance can now also find classes defined in namespaces
- Fix problems with missing parameters in late bound code. Also added (limited) support for calling overloaded methods and constructors in late bound code.
- Fixed problems with TransForm(), and several of the Str() functions.
- XSharp.Core is now fully compiled as Safe code.

- Fixed a problem with late bound assigns and access
- NIL<-> STRING comparisons are now compatible with Visual Objects
- Fixed problem with AEval() and missing parameters
- Added Set() function. Please be careful when using header files for _SET defines. There are subtle differences between the definitions in Harbour, Xbase++ and VO/Vulcan.
  We recommend NOT to use the defines from the header file but to use the defines that are defined inside the X# runtime DLLs
- Changed implementation of the functions used by the compiler for Aliased operations

RDD system

- Added support for DBF character fields up to 64K.
- Implemented the DBFCDX RDD
- Fixed several problems related to the DBFNTX RDD
- The DBF RDD was using the incorrect locking scheme for Ansi DBF files. It now uses the same scheme as VO and Vulcan.
- Macro compiled index expressions are not of the type _CodeBlock and not of the type RuntimeCodeBlock (the RuntimeCodeblock is encapsulated inside the _CodeBlock object).
  That prevents problems when storing these expressions inside a USUAL

Visual Studio integration

- Fixed an exception that could occur when typing a VAR expression
- When the project system makes a backup of a project file, we are now making sure that Readonly flags are cleared before writing to or deleting existing files.
- Reading intellisense data from C++ projects could send the intellisense engine into an infinite loop. This has been fixed.
- The changes to the Form.Designer.prg are now written to disk immediately, to make sure that changes to the form are recompiled if you press 'Run' or 'Debug' from the window of the form editor
- Improved support for intellisense for the VAR keyword.
- Added support for FoxPro on the Project Properties page to prepare for the Compiler and Runtime changes for FoxPro.
- .CH files are now also recognized as "X#" files in Visual Studio.
- You can now control the characters that select an entry from a Completion List. For example the DOT and COLON now also select the current selected element. The complete list can be found on the Tools-Options-TextEditor-XSharp-Intellisense page.
- Assemblies added to a project would not be properly resolved until the next time the project was loaded. This has been fixed.
- Fixed a problem in the codedom parser which feeds the windows form editor. You can now inherit a form from another form in the same assembly. You will have to compile the project first (of course).
- The .CH extension is now also registered as relevant for the X# project system.
- Changed auto indentation for #ifdef commands
- Fixed an exception that could occur during loading of project files with COM references.

- Added templates for class libraries in XPP and VO Dialect
- Sometimes a type lookup for intellisense was triggered inside a comments region. This has been fixed.

Tools

- VOXPorter was not removing calling conventions when creating delegates. This has been fixed
- VOXporter was sometimes generating project files with many duplicates of resource items. This has been fixed.
- VOXporter now marks prefix identifiers that conflict with one of the new keywords with "@@"
- The delay for the VOXporter welcome screen has been shortened.

Changes in 2.0.0.9 (Bandol Beta 9)

Compiler

- The Lexer (the part of the compiler that recognizes keywords, literals etc) has been rewritten and is slightly faster.
- The compiler now supports digit separators for numeric literals. So you can now write 1 million as:
  1_000_000
- Fixed problem where static local variables were not initialized with "" even when compiler option -vo2 was selected
- #ifdef commands using preprocessor macros such as __XSHARP_RT__ were not working correctly.
- The Xbase++ dialect now also supports the 'normal' class syntax.
- We had changed the 'Entrypoint' algorithm in Beta 8. This has been restored now and the -main command line option now works again as well. In stead the "body" of the Start method is now encapsulated in an anonymous function.
- Duplicate include files no longer produce an error but a warning
- Fix for problem with default parameter values with 'L' or 'U' suffix
- Added compiler error when specifying default parameter values for methods/functions with clipper calling convention
- DIM arrays of STRING were not initialized with "" when /vo2 was specified. This has been fixed.
- Added support for Dbase style memory variables (MEMVAR, PUBLIC, PRIVATE, PARAMETERS). See the MEMVAR topic in the help file for more information. This is only available for certain dialects and also requires the /memvar commandline option
- Added support for undeclared variables (this is NOT recommended!). This is only available for certain dialects and requires the /memvar AND the /undeclared commandline options
- Fixed a problem for comparisons between USUAL variables and STRING variables

- Fixed a problem with partial classes where the classname had different casing in the various declarations
- Fixed a problem with numeric default parameters with L or U suffixes
- Fixed a problem with line continuation semi colons followed by a single line comment with the multiline comments style.
- Fixed a problem with methods containing YIELD statements in combination with compiler option /vo9
- When a visibility modifier was missing on a generic method then this method was created as a private method. This has been fixed.
- When choosing between overloaded functions in XSharp.RT and XSharp.Core the function in the XSharp.RT assembly would sometimes be chosen although the overload in XSharp.Core was better
- CASE statements without CASE block but only a OTHERWISE block would crash the compiler. This has been fixed and an warning about an empty CASE statement has been added.

Runtime

- Several changes to the Macro compiler, such as the parsing of Hex literals, case sensitivity of parameters (they are no longer case sensitive) and limited support for function overloading.
- Several missing functions have been added, such as _Quit(),
- The return value of several Ord..() functions was incorrect. This has been fixed.
- Fixed a problem with CurDir() for the root directory of a drive
- Fixed a problem with calling Send() with a single parameter with the value NULL_OBJECT.
- Solved problem with incorrect parameters for DiskFree() and DiskSpace()
- MemoRead() and MemoWrit() and FRead..() and FWrite..() now respect the SetAnsi() setting like the functions in the VO Runtime.
- We have added 2 new functions to read/write binary files: MemoReadBinary() and MemoWritBinary()
- Not all DBOI_ enum values had the same value as in Vulcan. This has been solved.
- SetDecimalSep() and SetThousandSep() now also set the numeric separators in the current culture.
- The USUAL -> STRING conversion now calls AsString()
- Added support for Dbase style dynamic memory variables (MEMVAR, PUBLIC, PRIVATE, PARAMETERS). See the Memory Variables topic in the help file for more information.
- The IsDate() function now also returns TRUE for USUALs of type DateTIme. There is also a separate IsDateTime() function. We have also added IsFractional() (FLOAT or DECIMAL) and IsInteger (LONG or INT64) and IsInt64()
- Added missing Cargo slot to the Error class. Also improved Error:ToString()
- Fix for problem in W2String()
- And many more small changes.

Visual Studio Integration

- We have added a new tab page in the Project Properties dialog: Dialect. This contains dialect specific language options.

- 2 options from the Build options page (which is configuration dependent) have been moved to the Language page (which is build INdepedent), because that makes more sense:
  - Include Path
  - NoStdDef
- We have also added a project property on the Language page to specify an alternative standard header file (in stead of XSharpDefs.xh)
- The XSharp.__Array type was shown in the intellisense with the wrong name
- We have added entries on the Project Properties dialog pages to enable MEMVAR support and to enable Undeclared variables
- Fixed a problem in the CodeDom provider (used by the Windows Form editor) where fields with array types were losing their array brackets when writing back to the source.
- When writing changes from the windows form editor we are no longer writing to disk but to the opened (sometimes invisible) windows of the .designer.prg. This should prevent warning messages about the .designer.prg file that was changed outside Visual Studio
- Fixed a problem parsing source code where identifier names were starting with '@@'
- The Debugger was showing UINT64 as typename for ARRAYs. This has been fixed.
- Renaming forms in the Windows Forms editor was not working for forms with a separate .designer.prg. This has been fixed.
- Fixed a (very old) problem where the OutPutPath property in the xsproj file was sometimes set to $(OutputPath).
- Fixed an exception in the editor for empty source files or header files.
- Fixed an exception when the error list was created for errors without errorcode
- Commenting a single line in the editor will now always use the // comment format

Tools

- No changes in this release.

Changes in 2.0.0.8 (Bandol Beta 8)

Compiler

- The compiler source code has been upgraded to Roslyn 2.10 (C# 7.3). As a result of that there are some new compiler options, such as /refout and we also support the combination of the "PRIVATE PROTECTED" modifier that defines a type member as accessible for subclasses in the same assembly but not for subclasses in other assemblies
- We have added support for Xbase++ class declarations. See the Xbase++ class declaration topic for more information about the syntax and what is supported and what not.

- We have added support for simple macros with the &Identifier syntax
- We have added support for late bound property access:
  - The <Expression>:&<Identifier> syntax.
    This translates to IVarGet(<Expression>, <Identifier>).
  - The <Expression>:&(<Expression2>) syntax.
    This translates to IVarGet(<Expression>, <Expression2>).
  - Both of these can also be used for assignments and will be translated to IVarPut:
    <Expression>:&<Identifier> := <Value>
    This becomes IVarPut(<Expression>, <Identifier>, <Value>)
  - All of these will work even when Late Binding is not enabled.
- We have added a new compiler options /stddefs that allows you to change the standard header file (which defaults to XSharpDefs.xh)
- We have added a new preprocessor Match marker <#idMarker> which matches a single token (all characters until the first whitespace character)
- When you select a dialect now, then the compiler will automatically add some compiler macros. The VO dialect declares the macro __VO__, the Vulcan dialect declares the macro __VULCAN__ the harbour dialect declares the macro __HARBOUR__ and the Xbase++ dialect declares the macro __XPP__.
- When compiling against the X# runtime then also the macro __XSHARP_RT__ will be defined.
- We have added a new warning when you pass a parameter without 'ref' modifier (or @ prefix) to a method or function that expects a parameter by reference or an out parameter.
- We have also added a warning that will be shown when you assign a value from a larger integral type into a smaller integral type to warn you about possible overflow problems.

Runtime

- This build includes a **new faster macro compiler**. It should be fully compatible with the VO macro compiler. Some of the .Net features are not available yet in the macro compiler.
- We moved most of the generic XBase code to XSharp.RT.DLL. XSharp.VO.DLL now only has VO specific code. We have also added XSharp.XPP.DLL for XPP
- Fix Ansi2OEM problem with FRead3(), FWrite3() and FReadStr
- Added missing functions EnableLBOptimizations() and property Array:Count
- Fixed problem with latebound assign with CodeBlock values
- Fixed problem with AScan() and AEval() with missing parameters
- Changed error return codes for DirChange(), DirMake() and DirRemove()
- Send() was "swallowing" errors. This has been fixed
- Fixed a problem with assigning to multi dimensional arrays
- Fixed a problem with creating objects with CreateInstance() where objects are not in the "global" namespace
- Fixed several problems in the RDD system and support functions.
- Fixed several problems in the late binding support, such as IsMethod, IsAccess, IVarPut, IVarPutSelf etc.
- Fixed several problems with TransForm()
- Integer divisions for usuals containing integers now return either integers or else fractional numbers depending on the compiler setting of the main app.

- We fixed several conversions problems during late bound calls
- We have fixed several problems with the Val() and Str() functions.
- The internal type names for DATE and FLOAT have been changed to __Date and __Float. If you rely on these type names please check your code !
- DebOut32 was not outputting data to the debug terminal if the runtime was compiled in release mode. This has been fixed.

Visual Studio Integration

- Fixed filtering on 'current project' in the error list
- Type lookup for local variables was sometimes failing. This has been fixed
- Fixed a problem with Brace Matching that could cause an exception in VS
- Fixed a problem with Tooltips that could cause an exception in VS
- Fixed a problem with uncommenting that could cause an exception in VS
- New references added in VS would not always be included in the type search in the editor. This has been fixed.
- Member prototypes for constructors now include the type name and curly braces
- We have started work on improved code completion for variables declared with VAR
- We have started with support for code completion for members of Generic types. This is not finished yet.
- PRG files that are not part of a X# project and not part of a Vulcan project are now also colorized in the editor.

Tools

- VulcanXPorter was always adjusting the referenced VO libraries and was ignoring the "Use X# Runtime" checkbox
- VOXPorter now has an option to copy the resources referenced in the AEF files to the Resources subfolder in the project
- VOXPorter now also copies the cavowed, cavofed and cavoded template files to the properties folders in your project.

Changes in 2.0.0.7 (Bandol Beta 7)

Compiler

- When calling a runtime function with a USUAL parameter the compiler now automatically prefers methods or functions with "traditional" VO types over the ones with enhanced .Net types. For example when there are 2 overloads, one that takes a byte[] and another that takes a string, then the overload that takes a string will get preference over the overload that takes a byte[].
- Resolved a problem with .NOT. expressions inside IIF() expressions

- Improved debugger break point generation for Invoke expressions (like String.Compare())
- Fixed a pre-processor error for parameters for macros defined in a #define. These parameters must have the right case now. Parameters with a different case will not be resolved any longer.
- Fixed a pre-processor error where optional match patterns in pre-processor rules were repeated. This is too complicated to explain here in detail <g>.
- The code generated by the compiler for Array operations now uses the new interfaces declared in the X# runtime (see below).

Runtime

- We have added several missing functions, such as _GetCmdLine, Oem2AnsiA() and XSharpLoadLibrary
- Fixed problems in CreateInstance, IVarGet, IVarPut(), CtoDAnsi() and more.
- Added VO Compatible overload for FRead4()
- No longer (cathed) exceptions are produced for empty dates
- Ferror() was not always return the error of a file operation. This has been fixed
- We have added a new FException() function that returns the last exception that occurred for a low level file operation
- Casting a usual containing a PTR to a LONG or DWORD is now supported
- Some new interfaces have been added related to array handling. The compiler no longer inserts a cast to Array inside the code, but inserts a cast to one of these interfaces depending on the type of the index parameter. The USUAL type implements IIndexer and IIndexProperties and dispatches the call to the objects inside the usual when this objects exposes the interface. This is used for indexed access of properties when using AEval or AScan on an ARRAY OF <type>
  - XSharp.IIndexer
  - XSharp.INamedIndexer
  - XSharp.IIndexedProperties

SDK Classes

- We have added the Hybrid UI classes from Paul Piko (with permission from Paul)

Tools

- The Vulcan XPorter now also has an option to replace the runtime and SDK references with references to the X# runtime

Changes in 2.0.0.6 (Bandol Beta 6)

Compiler

- The compiler was sometimes still generating warnings for unused variables generated by the compiler. This has been fixed.
- The compiler will now produce a warning that #pragmas are not supported yet (9006)
- Added compiler macro __FUNCTION__ that returns the current function/method name in original casing.
- Literal sub arrays for multidimensional arrays no longer need a type prefix when compiling in the Core dialect
- Fixed problem with the Global class name that would happen when building the runtime assemblies (these have a special convention for the global class names)
- When the calling convention of a method in an interface is different from the calling convention of the implementation (CLIPPER vs Not CLIPPER) then a new error (9067) will be generated by the compiler.
- The calling convention for _DLL functions and procedures is now optional and defaults to PASCAL (stdcall)
- The namespace alias for using statements was not working in all cases.
- The compiler will now generate an error for code that incorrectly uses the VIRTUAL and OVERRIDE modifiers.
- The compiler was throwing an exception for a specific kind of incorrect local variable initializer with generic arguments. This has been fixed.
- Visibility modifiers on GET or SET accessors for properties were not working correctly (INTERNAL, PRIVATE etc). This has been fixed.
- The compiler now handles PSZ(_CAST,...) and PSZ(..) differently. When the argument is a literal string, then the PSZ will only be allocated once and stored in a "PSZ Table" in your assembly. The lifetime of this PSZ is then the lifetime of your app. When this happens then the new compiler warning XS9068 will be shown. When the argument is a string stored in a local or global (or define) then the compiler can't know the lifetime of the PSZ. It will therefore allocate the memory for the PSZ with the StringAlloc() function. This ensures that the PSZ will not go out of scope and be freed. If you use this a lot in your application then you may be repeatedly allocating memory. We recommend that you avoid the use of the cast and conversion operators for PSZs and take control of the lifetime of the PSZ variables by allocating and freeing the PSZ manually. PSZ casts on non strings (numerics or pointers) simply call the PSZ constructor that takes an intptr (this is used on several spots in the Win32API library for 'special' PSZ values).
- Named arguments are now also supported in the Vulcan dialect. This may lead to compiler errors if your code looks like the code below, because the compiler will think that aValue is a named argument of the Empty() function.

```
IF Empty(aValue := SomeExpression())
```

- If you were inheriting a static class from another class then you would get a compiler warning before. This is now a compiler error, because this had a side effect where the resulting assembly contained a corrupted reference.
- The overload resolution code now chooses a type method/function over a method/function with clipper calling convention.
- The Xbase++ dialect is now recognized by the compiler. For the time being it behaves the same as Harbour. We have also added the compiler macro **__DIALECT_XBASEPP__** that will be automatically define to TRUE when compiling in Xbase++ mode.
- Fixed a problem in the PDB line number generation that would cause incorrect line numbers in the debugger

Visual Studio integration

- The source code editor was not always showing the correct 'active' region for #defines defined in #include files.
- Opening a source file without entities (e.g. a header file) could result in an error message inside VS.
- Fixed a null reference exception in the editor
- Fixed a problem when un-commenting code in the editor
- Improved load time performance for large solutions with many dependencies.
- Fixed a problem where the intellisense engine could lock a DLL that was used by a project reference or assembly reference.
- Fixed a problem where missing references (for example COM references that were not installed on the developers machine) could cause problems with the type lookup when opening forms in the windows forms editor.
- Added an option to select the Harbour dialect on the project properties page.

The Build System

- The Build system did not recognize that begin NAMESPACE lines in source code were commented out. This has been fixed.

VOXporter

- We have added an option to sort the entities in alphabetical order in the output file.
- We have added an option so you can choose to add the X# Runtime as reference to your application (otherwise the Vulcan runtime is used)

Runtime

- The SetCentury setting was incorrect after calling SetInternational(#Windows). This has been fixed.
- The Descend function for dates now returns a number just like in VO
- The functions ChrA and AscA have been renamed to Chr() and Asc() and the original functions Chr() and Asc() have been removed. The original functions were using the DOS (Oem) codepage and this is not compatible with Visual Objects.
- On several places in the runtime characters were converted from 8 bit to 16 bit using the System.Encoding.Default codepage. This has been changed. We use the codepage that matches the WinCodePage in the Runtimestate now. So by

setting the Windows codepage in the runtime state you now also control the conversions from Unicode to Ansi and back

- The Oem2Ansi conversion was incorrect for some low level file functions.
- We have changed several things in the Late Binding support
- All String - PSZ routines (String2PSz(), StringAlloc() etc) now use the Windows Codepage to convert the unicode strings to ansi.
- If you library is compiled with 'Compatible String comparisons' but the main app isn't, then the string comparisons in the library will follow the same rules as the main app because the main app registers the /vo13 setting with the runtime. The "compatible" stringcomparison routines in the runtime now detect that the main app does not want to do VO compatible string comparisons and will simply call the normal .Net comparison routines.
  We therefore recommend that 3rd party products always use the Compatible String comparisons in their code.
- Preliminary documentation for the runtime was generated from source code comments and has been included as chapter in this documentation.

The VO SDK

- This build includes the first version of the VO SDK compiled against the X# runtime. We have included the following class libraries
  - Win32API
  - System Classes
  - RDD Classes
  - SQL Classes
  - GUI Classes
  - Internet Classes
  - Console Classes
  - Report Classes
- All assemblies are named VO<Name>.DLL and the classes in these assemblies are in the VO namespace.
- This SDK is based on the VO 2.8 SP3 source code. The differences between VO 2.8 SP3 and VO 2.8 SP4 will be merged in the source later,
- The Libraries for OLE, OleServer and Internet Server are not included. The OleAutoObject class and its support classes is included in the XSharp.VO library. OleControl and OleObject are not included.
- Preliminary documentation for these classes was generated from source code comments and has been included as chapter in this documentation.

The RDD system

- This build includes the first version of the RDD system. DBF-DBT is ready now. Other RDDs will follow in the next builds. Also most of the RDD related functions are working in this build.
- This build also includes the first version of the Advantage RDD. With this RDD you can access DBF/DBT/NTX files , DBF/FPT/CDX files and ADT/ADM/ADI files. The RDD names are the same as the RDD names for Vulcan. (AXDBFCDX, AXDBFNTX, ADSADT). We also support the AXDBFVFP format and the AXSQLCDX, AXSQLNTX, AXSQLVFP. For more information about the differences and possibilities of these RDD look in the Advantage documentation.
  We have coded the Advantage RDD on top of the Advantage Client Engine. Our

RDD system detects if you are running in x86 or x64 mode and calls functions in Ace32 or Ace64 accordingly.

To use Advantage you copy the support DLLs from an Advantage Vulcan RDD to the folder of your application. Look at the Advantage docs for Vulcan to see the list of the DLLs. The Advantage RDD is part of the standard XSharp.RDD.DLL which therefore replaces the AdvantageRDD.Dll for Vulcan.

- The XSharp.Core DLL now also has RDD support. We have chosen NOT to implement this in functions, but as static methods inside the CoreDb class. Old code that uses the VoDb..() functions can be simply ported by changing "VoDb" to "CoreDb."

  The parameters and return values that are USUAL in VO and Vulcan are implemented as OBJECT in the CoreDb class.

  The ..Info() methods have 2 overloads. One that takes an Object and one that takes a reference to an object.

  The methods inside CoreDb return success or failure with a logical value like the VODB..() functions in VO. If you want to know what the error was during the last operation then you can access that with the method CoreDb._ErrInfoPtr() . This returns the last exception that occurred in a RDD operation.

- At this moment the CoreDb class only has a FieldGet() that returns an object. We will add some extra methods that return values in a specified type in the next build (such as FieldGetString(), FieldGetBytes() etc). We will also add overloads for FieldPut() that take different parameter types.

- The XSharp.VO DLL has the VoDb..() functions and the higher level functions such as DbAppend(), EOF(), DbSkip() etc.

  The VoDb..() functions return success or failure with a logical value. If you want to know what the error was during the last operation then you can access that with the method _VoDbErrInfoPtr() . This returns the last exception that occurred in a RDD operation.

- You can mix calls to the VoDb..() functions and CoreDb...() methods. Under the hood the VoDb..() functions also call the CoreDb methods.

- The higher level functions may throw an exception just like in VO. For example when you call them on a workarea where no table is opened. Some functions simply return an empty value (like Dbf(), Recno()). Others will throw an exception. When you have registered an error handler with ErrorBlock() then this error handler will be called with the error object. Otherwise the system will throw an exception.

- Date values are returned by the RDD system in a DbDate structure, Float values are returned in a DbFloat structure. These structures have no implicit conversion methods. They do however implement IDate and IFloat and they can and will be converted to the Date and Float types when they are stored in a USUAL inside the XSharp.VO DLL. The DbDate structure is simply a combination of a year, month and date. The DbFloat structure holds the value of fields in a Real8, combined with length and the number of decimals.

- More documentation about the RDD system will follow later. Of course you can also look at the help file and source code on GitHub.

Changes in 2.0.0.5 (Bandol Beta 5)

Compiler

- The strong named key for assemblies with native resources was invalid. This has been fixed
- When an include file was included twice for the same source (PRG) file then a large number of compiler warnings for duplicate #defines would be generated. Especially when the Vulcan VOWin32APILibrary.vh was included twice then over 15000 compiler warnings would be generated per source file where this happened. This large number of warnings could lead to excessive memory usage by the compiler. We are now outputting a compilation error when we detect that the same file was included twice. We have also added a limit of 500 preprocessor errors per source (PRG) file.
- A change in Beta 4 could result in compiler warnings about unused variables that were introduced automatically by the X# compiler. This warning will no longer be generated.
- The compiler now correctly stores some compiler options in the runtime state of XSharp.

Runtime

- Fixed a problem in the Ansi2OEM and OEM2Ansi functions.
- Fixed a problem in the sorting for SetCollation(#Windows)
- Fixed a problem with string comparisons in runtime functions like ASort(). This now also respects the new runtime property CompilerOptionVO13 to control the sorting

Visual Studio integration

- The sorting of the members in the editor dropdown for members was on methodname and propertyname and did not include the typename. When a source file contained more than one type then the members would be mixed in the members dropdown

Build System

- The default value for VO15 has been changed back from false to undefined.

Changes in 2.0.0.4 (Bandol Beta 4)

Compiler

- **POSSIBLY BREAKING CHANGE: Functions now always take precedence over same named methods**. If you want to call a method inside the same class you need to either prefix it with the typename (for static methods) or with the SELF: prefix. If there is no conflicting function name then you can still call the method with just its name. We recommend to prefix the method calls to make your code easier to read.
- The compiler was accepting just an identifier without a INSTANCE, EXPORT or other prefix and without a type inside a class declaration. It would create a public field of type USUAL. That is no longer possible.
- Improved the positional keyword detection algorithm (this also affects the source code editor)
- The || operator now maps to the logical or (a .OR. b) and not to the binary or (_OR(a,b))
- The VAR statement now also correctly parses

    ```
    VAR x = SomeFunction()
    ```

    And will compile this with a warning that you should use the assignment operator (:=).
    We have added this because many people (including we) copy examples from VB and C# where the operator is a single equals token.
- Error messages about conflicting types now include the fully qualified type name.
- The compiler no longer includes the width for literal Floats. This is compatible with VO.
- A Default parameter of type Enum is now allowed.

Runtime

- Added several functions that were missing, such as __Str() and DoEvents()
- Fixed a problem in the macro compiler with non-english culctures.
- Added several overloads for Is..() functions that take a PSZ instead of a string, such as IsAlpha() and IsUpper().
- Added some missing error defines, such as E_DEFAULT and E_RETRY.
- Fix for a problem with SubStr() and a negative argument
- Fix for a problem with IsInstanceOf()
- Fix for a problem with Val() and a hex value with an embedded 'E' character
- Added implicit conversions from ARRAY to OBJECT[] and back.
- Several changes to the code for Transform() and Unformat() to cover several exotic picture formats

- Changes to the code for SetCentury() to automatically also adjust the date format (SetDateFormat())
- Fixes for the Str() family of functions in combination with SetFixed() and SetDigitFixed().

Visual Studio integration

- Fixed a problem when building projects in the latest build of Visual Studio
- Several 'keywords' were not case synchronized before, such as TRUE, FALSE, NULL_STRING etc,
- Keywords are not case synchronized on the current line as long as the user has the cursor on them or immediately after them. That means that when you type String and want to continue to change it to StringComparer then the formatter will no longer kick in and change "String" to the keyword case before you have the chance to complete the word.
- The Control Order dialog inside the form editor was not saving its changes.
- Added an option to include all entities from the editor, or just the members from the current selected type in the right dropdown of the editor
- The editor was also matching braces inside literal strings and comments. This has been fixed.
- Fixed a problem with the CodeDom parser where extended strings (strings containing CRLF tokens or other special tokens) were parsed incorrectly. This resulted in problems in the windows forms editor.
- The member resolution code in the editor was not following the same logic as the compiler: When a function and a method with the same name exist it was resolving to the method in stead of the function. This has been fixed.
- Fixed a problem when debugging in X64 mode.
- Fixed an exception when comparing source code files with SCC integration.
- Fixed several problems w.r.t. the XAML editor:
  o Code is now generated with STRICT calling convention to avoid problems when compiler option "Impliciting CLIPPER calling convention" is enabled
  o WPF and other templates now include STRICT calling convention for the same reason
  o The XAML editor could not properly load the current DLL or EXE and had therefore problems resolving namespaces and adding user controls to the tool palette. This has been fixed.
- We have added an option to the Tools/Editor/XSharp/Intellisense options that allow you to control how the member combobox in the editor works. You can choose to only show methods & properties of the current type or all entities in the right combobox. The left combobox always shows all types in the file.
- Some of the project and item templates have been updated. Methods and constructors without parameters now have a STRICT calling convention. Also the compiler option /vo15 has been explicitly disabled in templates for the Core dialect.

Changes in 2.0.0.3 (Bandol Beta 3)

Compiler

- When 2 method overloads have matching prototypes the compiler now prefers the non generic one over the generic one
- Fixed an exception that could occur when compiling a single line of source code with a preprocessor command in it.

Runtime

- Added Mod() function
- Added ArrayNew() overload with no parameters
- Fixed problem in __StringNotEquals() when length(RHS) > length(LHS) and SetExact() == FALSE
- Added missing string resource for USUAL overflow errors

Visual Studio integration

- Improved keyword case synchronization and indenting. Also a source file is 'Keyword Case' synchronized when opened.
- Opening a source file by double clicking the find results window no longer opens a new window for the same source file
- Improved type lookup speed for intellisense
- Fixed a problem that would prevent type lookup for types in the same namespace
- Fix for QuickInfo problem introduced in the latest Visual Studio 2017 builds
- QuickInfo tips are no longer shown in the debugger where they were overlapping with debugger tooltips
- The comboboxes with methods and functions in the editor window no longer shows parameter names and full type names. Now it shows the shortened type names for the parameters
- These same comboboxes now show the file name for methods and properties defined in another source file
- Fixed problem in the window editor with generating code for tab pages

Vulcan XPorter

- Project dependencies defined in the solution file were not properly converted

VO XPorter

- Fixed a problem where resource names were replaced with the value of a define

Changes in 2.0.0.2 (Bandol Beta 2)

Compiler

- The compiler now transparently accepts both Int and Dword parameters for XBase Array indices
- When the compiler finds a weakly typed function in XSharp.VO and a strongly typed version in XSharp.Core then it will choose the strongly typed version in XSharp.Core now.
- In the VO and Vulcan dialect sometimes an (incorrect) warning 'duplicate usings' was displayed. This is now suppressed.
- The debugger information for the Start function has been improved to avoid unnecessary step back to line 1 at the end of the code
- The debugger break point information for BEGIN LOCK and BEGIN SCOPE has been improved
- The debugger break point information for multi line properties has been improved
- /vo6, /vo7 and /vo11 are now only supported in the VO/Vulcan dialect

Runtime

- Removed DWORD overloads for Array indexers
- Fixed overload problem for ErrString()
- Fixed overload problem for _DebOut()
- Fixed problems in DTOC() and Date:ToString()
- Fixed ASort() incompatibilities with VO
- Fixed memory blocks now get filled with 0xFF when they are released to help detect problems

Visual Studio

- Fix 'Hang' in VS2017 when building
- Fix 'Hang' in VS2017 when a tooltip (QuickInfo) was displayed
- Fixed problem with debugging x64 apps
- You can no longer rename or delete the Properties folder
- Selecting 'Open' from the context menu on the the Properties folder now opens the project properties screen
- Updated several icons in the Project Tree
- Enhancements in the Goto Definition

Build System

- Fix problem with CRLF in embedded resource commandline option

Changes in 2.0.0.1 (Bandol Beta 1)

Compiler

New features

- Added support for **ARRAY OF** language construct. See the [Runtime chapter](#) for more information about this.
- Added support for the X# Runtime assemblies when compiling in the VO or Vulcan dialects.
- Added support for the "Pseudo" function ARGCOUNT() that returns the # of declared parameters in a function/method compiled with clipper calling convention.
- Added a new warning number for assigning values to a foreach local variable. Assigning to USING and FIXED locals will generate an error.

Optimizations

- Optimized the code generation for Clipper calling convention functions/methods
- The /cf and /norun compiler options are no longer supported
- The preprocessor no longer strips white space. This should result in better error messages when compiling code that uses the preprocessor.
- Some parser errors are now more descriptive
- Changed the method that is used to determine if we compile against CLR2 or CLR4. The compiler checks at the location either system.dll or mscorlib.dll. When this location is in a path that contains "v2", "2.", "v3" or "3." then we assume we are compiling for CLR2. A path that contains "V4" or "4." is considered CLR4. The /clr commandline option for the compiler is NOT supported.
- The preprocessor now generates an error when it detects recursive #include files.

Bug fixes

- Fixed a problem when using the [CallerMemberAttribute] on parameters when compiling in Vulcan or VO Dialect
- Abstract properties should no longer generate a warning about a body
- You can now correctly use ENUM values as array indexes.
- Fixed a problem for Properties with PUBLIC GET and PRIVATE SET accessors.
- Fixed an issue where assigning an Interface to a USUAL required a cast to Object
- Fixed an issue where IIF expressions with literal types were returning the wrong type (the L or U suffix was ignored)
- Fixed an issue where the declaration LOCAL x[10] was not compiled correctly. This now compiles into a local VO Array with 10 elements.

Visual Studio Integration

- Build 1.2.1 introduced a problem that could cause output files to be locked by the intellisense engine. This has been fixed

- The editor parser had problems with nested types. This has been fixed
- Enum members were not included in code completion for enums inside X# projects
- Some improvements in the code reformatting
- Added option on the Tools/Options for the editor to include keywords in the "All tokens" completion list
- Fixed a problem where assemblies that could not be loaded to retrieve meta information would be retried 'for ever'
- Fixed a problem with retrieving type information from assemblies that contained both managed and unmanaged code.
- Added some properties for referenced assemblies to the IDE Properties window
- Fixed a problem with assembly references and the Windows Forms editor, introduced in one of the latest Visual Studio 2017 updates
- When enabling XML output on the Project Properties window an incorrect filename was shown for assemblies that contain a '.'in the assembly name.
- The editor parser now has better support for parameters of type REF and OUT
- Added support for 'Embed Interop Types' in the property windows for Assembly References and COM references
- Fixed a problem where the codemodel was sometimes locking output DLLs for Project references

Build System

- Fixed a problem with the naming of the XML documentation file.

Runtime

- Added XSharp.Core.DLL, XSharp.VO.DLL and XSharp.Macrocompiler.DLL. Most runtime functions are implemented and supported. See the X# Runtime chapter for more information

VO XPorter

- SDK related options have been removed. They will be moved to a new tool later.

Changes in 1.2.1

Compiler

- Fixed a problem where a compilation error resulted in the message "Failed to emit module" without further information
- Fixed a problem with ++, -- += and similar operations in aliased expressions (like CUSTOMER->CUSTNO++)

- Constructor initializers and Collection initializers were not working after a constructor with parameters. That has been fixed.
- Fixed an issue with negative literal values stored in a USUAL when overflow checking was enabled.
- For the CATCH clause now both the ID and the TypeName are optional. This means that there are 4 variations.
  You can only have one catch clause without type, since this defaults to the System.Exception type. However, you can have many catch clauses without ID.

```
CATCH ID AS ExceptionType
CATCH ID                   // defaults to Exception type
CATCH AS ExceptionType
CATCH                      // defaults to Exception type
```

Visual Studio Integration

- Improved the speed of the background code scanning
- Improved the speed of the background parser inside the editor
- Fixed a problem in the codedom provider that is used by the windows forms editor

Changes in 1.2.0

Compiler

- You can now pass NULL for parameters declared by reference for compatibility with VO & Vulcan.
  We STRONGLY advise not to do this, unless you make sure that the function expects this and does not assign to the reference parameter without checking for a NULL reference first. This will only work when the /vo7 compiler option is enabled.
- We have made some optimizations in the Lexer. The compiler should be a little faster because of that
- We fixed a problem with the automatic constructor generation (/vo16) for classes that inherit from classes defined in an external DLL
- When compiling with /vo2 any string fields assigned in a child class before the super constructor was called would be overwritten with an empty string. The generated code will now only assign an empty string when the string is NULL.
  Note: we do not recommend to assign parent fields in the child constructor before calling the super constructor. Manually coded default values for parent fields will still overwrite values assigned in the child constructor before the SUPER call
- Fixed a problem with CHECKED() and UNCHECKED() syntax in the VO dialect
- Fixed a problem with choosing overloads for methods where an overload exists with a single object parameter and also an overload with an object[] parameter.
- Added support to the parser for LOCAL STATIC syntax

- Fixed a problem with compiler option /vo9 (Allow missing return values) and procedures or methods that return VOID
- Improved debugger sequence point generation. The compiler no longer generates 'hidden' breakpoint information for startup and closedown code in the VO/Vulcan dialects, and for expression statements no longer a double step is necessary.
- ACCESS and ASSIGN for partial classes could generate error messages without source file name. This has been solved.
  The compiler now generates slightly different code for these "partial" properties. The Access and Assign are implemented as compiler generated methods and the property getter and property setter now call these methods.
- The compiler was not recognizing the _WINCALL calling convention. This has been fixed.
- The compiler now generates a warning when the #pragma command is used

Visual Studio Integration

- More performance improvements in the editor. Especially large source files with incorrect code could slow down the editor.
- The editor parser no longer tries to parse include files repeatedly when these files contain #defines only (like the Vulcan header files)
- The source code editor tried to show intellisense for words in a comment region. That has been fixed.
- We have started work on Object Browser and Class Browser.
- Opening and closing of projects should be slightly faster
- The internal code model used by the editors now disposes its loaded information when projects are closed and no projects need this information anymore. This should reduce the memory usage of the X# project system
- Matching keywords, such as IF .. ENDIF and FOR .. NEXT should now be highlighted in the editor
- If you select an identifier in the editor then that identifier will be highlighted in the current method/function on all places where it is used
- We have added several features that you need to enable/disable on the **Tools/Options/Text Editor/XSharp/Intellisens**e dialog:
  - The code completion in the editor also supports instance member completion when a dot is pressed.
    Please note that the compiler ONLY accepts this in the Core language, not in the VO & Vulcan dialect. So the option has no effect inside projects with other dialects.
  - We have added some options to control the sorting of the DropDown comboboxes in the editor, as well as if fields/instance variables should be included in these comboboxes. When you do not sort, then the entries in the dropdown box will be shown in the order in which they are found in the source file.
  - We have added the option to autocomplete identifiers when typing. This includes locals, parameters, class fields, namespaces, types etc.
- Overridden methods in subclasses with the same signature as the parent methods they override are no longer counted as overloads in completionlists
- A missing reference DLL could "kill" the intellisense engine. This no longer happens. Of course the type info from a missing referenced DLL is not included.

- Properties and methods in the generated source files for XAML code (the .g.prg files in the OBJ folder) are now also parsed and included in the completion lists in intellisense and in the Class Browser and Object Browser windows.

VOXPorter

- The installer now includes the correct version of VOXPorter <g>
- VOXporter now supports the following commandline options:

```
/s:<source folder or aef>
/d:<destination folder>
/r:<runtime folder>
/nowarning
```

- Some code corrections were added for issues found in the GUI classes
- The template files can now also be found when VOXPorter is run from a different working directory

Changes in 1.1.2

Compiler

- Added compiler warning for code that contains a #pragma
- Fixed a problem with iif() functions and negative literal values

Visual Studio Integration

- Fixed a slowness in the editor after typing a send (:) operator
- Enum values are now properly decoded in the debugger
- Fixed the CodeDom provider for handling literal FALSE values and negative numbers. As a result, more (Vulcan created) winforms should open without problems
- Some positional keywords (such as ADD and REMOVE) are no longer colored as keyword in the editor for incomplete code when they appear after a colon ':' or dot '.'

VOXPorter

- Fixes for exporting the VO RDD Classes from the SDK

Changes in 1.1.1

Compiler

- Fixed a problem with Debugger Breakpoints for DO CASE and OTHERWISE
- Fixed a problem with Debugger Breakpoints for sourcecode that is defined in #included files
- Added support for the Harbour Global Syntax where the GLOBAL keyword is optional
- Fixed a problem with FOR.. NEXT loops with negative step values
- In some situations the @@ prefix to avoid keyword conflicts was not removed from workarea names or field names. This has been fixed
- In the VO/Vulcan dialect a warning (XS9015) was generated when a default parameterless SUPER constructor call was automatically generated. This error message is now suppressed. However a generated SUPER constructor call with parameters still generates a warning.
- Prepared the compiler for Xbase type names and function names in the XSharp Runtime

Preprocessor

- Fixed a crash in the preprocessor
- The preprocessor was generating an error "`Optional block does not contain a match marker`" for blocks without match marker. This is now allowed.
  (for example for the ALL clause in some of the Database UDCs)
- When the same include files was used by multiple source files, and different sections of this file were included because of different #ifdef conditions, then the preprocessor would get "confused". This has been fixed.
- Debugger file/line number information from source code imported from #include files is not processed correctly.

Visual Studio Integration

- Fixed several issues with the Windows Form Editor
- The class declaration generated by the VO compatible editors now included the PARTIAL modifier.

Changes in 1.1

Compiler

- Fixed a problem with Codeblocks used in late bound code after the release of X# 1.0.3
- Fixed a problem with overriding properties in a subclass that inherit from a class where only the Assign (Set) or Access (Get) are defined.
- The compiler option /vo16: automatically generate VO Clipper constructors has been implemented.
- Fixed a crash in the compiler for compiler errors that occur on line 1, column 1 of the source file
- Fixed a problem where overflow checking was not following the /ovf compiler option
- Fixed problem with public modifier for interface methods
- Added proper error message with unreachable fields in external DLLs
- Fixed a problem with debugger sequence points (debugger stepping)
- X# generated pdb files are now marked with the X# language GUID so they are recognized as X# in the VS debugger
- DATETIME (26) and DECIMAL (27) are added as UsualType to the compiler in preparation of the X# runtime that allows usuals of these types
- Compiler options /VO15 and /VO16 now produce an error message when used outside the VO/Vulcan dialect
- Methods declared outside a class (VO style code) would declare a private class and not a public class
- ASTYPE has been changed to a positional keyword
- Fixed a problem with the Chr() and _Chr() functions for literal numbers > 127
- Added support for the __CLR2__ and __CLR4__ compiler macros. The version is derived from the folder name of mscorlib.dll and/or system.dll
- The Codeblock syntax was not working in the Core dialect.
- Some new keywords, such as REPEAT, UNTIL, CATCH, FINALLY,VAR, IMPLIED, NAMESPACE, LOCK, SCOPE, YIELD, SWITCH etc are now also positional and will only be recognized as keyword when at the start of a line or after a matching other keyword.
  This should help prevent cryptic error messages when these keywords are used as function names.

Visual Studio

Source code editor

- Added Goto Definition for Functions and Procedures
- Improved Info tips for Functions and Procedures
- Improved case synchronization

- Added first version of smart indenting
- Fixed lookup problems in the intellisense engine that could lock up VS
- Compiler Generated types are now suppressed from the completion lists.
- Added partial support for intellisense for LOCAL IMPLIED and VAR variables
- Added support for Format Document. This also sets the case for identifiers according to the tools defined in the Tools/Options menu
- Performance improvements for the background file scanner. This scanner is also paused during the build process to improve the compilation speed.

Project System and MsBuild

- Fixed a problem in the project files with conditioned property groups. Existing projects will be updated automatically
- Added support for the /vo16 compiler option in MsBuild and the VS Project Property pages.
- Fixed a problem with the /nostddef compiler option which was not working as expected.
- Fixed a problem which would occur when entering resources or settings in the project property dialog
- Fixed a problem with the /nostdlib compiler option
- License.Licx files are now added as "Embedded Resource"
- Fixed a problem with the automatic adding of License files
- When a project has a "broken reference" and a new reference is added with the correct location, then the broken reference will be deleted and the new references will be added instead.
- The MSBuild support DLL was unable to find the location of the compiler and native resource compiler when running inside a 64 bit process

Form Editor

- Improved Windows Form Editor support for types defined in project references. We will now detect the location of the output files for these projects, like the C# and VB project systems.
- The Code parser for the Form Editor was having problems with untyped methods. This has been fixed.

VO Window and Menu Editor

- The code generator for the Window and Menu editor will delete old unused defines.
- Changed the item template for VO windows to fix a problem when adding an event handler to a window that has not been saved yet
- The code generator for the Window editor was not outputting a style for WS_VISIBLE. This has been fixed.

Debugger

This build introduces a first version of the XSharp debugger support
- The Visual Studio debugger now shows the language X# in the callstack window and other places
- Functions, Methods and procedures are now displayed in the X# language style in the callstack window
- Compiler generated variables are no longer shown in the locals list
- The locals list now shows SELF in stead of this

- X# predefined types such as WORD, LOGIC etc are shown with their X# type names in the locals window

Testing

- Added support for the Test Explorer window
- Added templates for unit testing with XUnit, NUnit and Microsoft Test

Other

- Added warning when Vulcan was (re)installed after XSharp, which could cause a problem in the Visual Studio integration
- The VS Parser was marking Interfaces as structure in stead of interface. This has been fixed.
- The tool XPorter tools have better names in the VS Tools Menu
- The VS Background parser gets suspended while looking up type information to improve the intellisense speed
- Several changes were made to the templates that come with X#

XPorter

- Fix problem in propertygroup conditions.

VOXPorter

- Generate clipper constructors is now disabled by default
- Fixed a problem in the VS Template files.

Changes in 1.0.3

Compiler

- Fixed a problem with the index calculation for Vulcan Arrays indexed with a usual argument
- Fixed a problem with the generation of automatic return values for code that ends with a begin sequence statement or a try statement
- Optimized the runtime performance for literal symbols.
  The compiler now generates a symbol table for the literal symbols and each literal symbol used in your app is only created once.
  You may experience a little delay at startup if your app uses a LOT (thousands) of literal symbols. But the runtime performance should be better.
- Added a compiler error for code where numerics are casted to an OBJECT with the VO compatible _CAST operator.
  This is no longer allowed:
  ```
  LOCAL nValue as LONG
  LOCAL oObject as OBJECT
  ```

```
nValue := 123
oObject := OBJECT(_CAST, nValue)
```

Changes in 1.0.2

Compiler

- Added support for XML doc generation. We support the same tags that the C# compiler and other .Net compiler support.
- Improved some parser errors.
- Created separate projects for portable and non portable (.Net framework 4.6) for the compiler and scripting
- Fixed the code generation for conversion from USUAL to a known type. Now the same error is generated that Vulcan produces when the object type in the usual does not match the type of the target variable
- When declaring a type with the same name as the assembly now a compiler error is generated with a suggested work around.
- Fixed a strange compiler message when using a PTR() operation on a method call
- Indexed access to bytes in a PSZ is now 1 based like in VO when the VO dialect is used. The Vulcan dialect needs 0 based index access like Vulcan.
- The error message for compound assignments of FLOAT and USUAL has been removed. The compiler now contains a workaround for the problem in the Vulcan Runtime
- For ambiguous code where the compiler has to choose between a Function call and a static method call in any other class, the compiler now chooses the function call over the method call (Vo and Vulcan dialect). The warning will still be generated.
- When passing a variable by reference with the @ sign the compiler will now check to see if the declared type of the function/method parameter matches the type of the local variable.
- Some compiler warnings for unused variables were being suppressed in the VO/Vulcan dialect. They are activated again.

Scripting

- The scripting was not working in release 1.01

Visual Studio Integration

- QuickInfo could generate a 'hang' in the VS editor. This has been fixed
- Added quickinfo for globals and defines
- Added completionlists for globals and defines
- Added VO Form editor to edit vnfrm/xsfrm files and generate the code and resources

- Added VO Menu editor to edit vnmnu/xsmnu files and generate the code and resources
- Added VO DbServer editor and VO Fieldspec editor to edit vndbs/xsdbs and vnfs/xsfs files and generate the code and resources
- Added keyword and identifier case synchronization.
- Fixed a problem where typing SUPER(in the editor could throw an exception
- Prebuild and Postbuild entries in the project file are now configuration specific
- Added support for XML Doc generation in the project system
- Fixed a 'hang' that could occur with Visual Studio 2017 version 15.3 and later

VO Xporter

- Fixed a problem when importing certain VO 2.7 AEF files
- Fixed a problem with acceptable characters in the solution folder name
- VO Form and menu entities are also included in the xsproj file
- Added an option to the INI files to specify the Vulcan Runtime files location (   )

Changes in General Release (1.0.1.1)

Compiler

- Fixed a problem with VERY old versions of the Vulcan Runtime
- Variables declared as DIM Byte[] and similar are now Pinned by the compiler
- [Return] attribute was not properly handled by the compiler. This has been fixed
- Compound Assignment (u+= f or -=) from USUAL and FLOAT were causing a stackoverflow at runtime caused by a problem in the Vulcan Runtime. These expressions now generate a compiler error with the suggestion to change to a simple assignment (u := u + f)

Visual Studio Integration

- Project References between XSharp Projects were also loaded as assemblyreference when resolving types. This could lead to speed problems and unnecessary memory usage
- Improved the speed of the construction of Completion Lists (such as methods and fields for a type).
- We have also added Completion List Tabs, where you can see fields, properties, methods etc. on separate tabs. You can enable/disable this in the Tools/Options/Text Editor/XSharp/Intellisense options page.

VO XPorter

- We have added a check to make sure that the default namespace for a X# project cannot contain a whitespace character

Changes in General Release (1.0.1)

New Features

- We have added support for ENUM Basetypes (ENUM Foo AS WORD)
- We have added a separate syntax for Lambda Expressions
- We have added support for Anonymous Method Expressions
- Typed local variables can now also be used for PCALL() calls
- Methods with the ExtensionAttribute and Parameters with the ParamArrayAttribute attributes now compile correctly, but with a warning

Compiler

- Fixed a problem with a late bound assign of a literal codeblock
- Resolved several name conflicts
- Improved several of the error messages
- Fixed compilation problem for Properties with only a SET accessor
- Fixed a crash in a switch block with an if .. endif statement
- Fix problem with virtual instance methods and structures
- Fixed name conflict foreach variables used in Array Literals
- Changed resolution for Functions and static methods with the same name.
  In the VO/Vulcan dialect functions take precedence over static methods. If you want to call the static method then then you need to prefix the method call with the classname.
- There is a separate topic in this documentation now that describes the syntax differences and similarities between Codeblocks, Lambda Expressions and Anonymous Method Expressions.
- Fixed incorrect error message for Start() function with wrong prototype.
- When an ambiguity is detected between a function and a static method then a warning is displayed

Visual Studio

- Added parameter tips for Functions and methods from "Using Static" classes
- Added parameter tips for Clipper Calling Convention functions and methods
- Added support for generics in Intellisense
- Intellisense will show keywords in stead of native type names (WORD in stead of System.UInt16 for example)

- Parameter tips are now shown when a opening **'('** or **'{'** is typed as well as when the user types a comma **','**.
- Parameter tips will show REF OUT and AS modifiers
- Added intellisense for COM references, both for normal COM references as well as for Primary Interop Assemblies. Also members from implemented interfaces are now included in intellisense code completion (this is very common for COM).
- Improved intellisense for Project References from other languages, such as C# and VB.
- Added intellisense for Functions in referenced projects and referenced Vulcan and/or X# assemblies
- Suppress "special type names" in intellisense lists
- Added support for "VulcanClassLibrary" attribute to help find types and functions
- Errors from the Native Resource compiler are now also included in the error list
- Fixed problem with parameter tips for Constructors
- Added memberlist support for X# type keywords such as STRING, REAL4 etc.
- Fixed several issues with the Windows Form editor in relation to ActiveX controls
- Added a menu option to start the VO Xporter tool
- Added background scanning for dependent items, to make sure that newly generated code is scanned and available for intellisense.
- Several changes to the Windows Forms editor and project system:
  - Added support for adding ActiveX controls
  - Added support for Form Inheritance. Our forms are also visible in the C# Inherited form wizard
  - Added support to add our Windows Forms Custom Controls to the ToolBox in Visual Studio
  - Some performance enhancements in the Codedom Parser that is used by the Windows Form Editor. You should notice this for larger forms.
- Fixed several crashes reported by users
- Native Resource files (.rc) now open in the Source code editor
- Improved background parsing speed
- Improved keyword colorization speed
- Improved handling of Type and Member dropdowns in the editor

Tools

- Added a first version of the VO Xporter tool
- The installer now registers .xsproj files, .prg, .ppo. .vh, .xh and .xs files so they will be opened with Visual Studio

Documentation

- We have added some chapters on how to convert your VO AEF and/or PRG files to a XIDE project and/or a Visual Studio solution.

Changes in 0.2.12

Scripting

- We have added the ability to use X# scripts. Some documentation about how this works can be found here. You can also find scripting examples in the c:\Users\Public\Documents\XSharp\Scripting folder

Compiler

All dialects

- The compiler is now based on the Roslyn source code for C# 7.
- Accesses and Assigns with the same name for the same (partial) class in separate source files are now merged into one property. This will slow down the compiler somewhat. We recommend that you define the ACCESS and ASSIGN in the same source file.
- Added support for repeated result markers in the preprocessor
- We have added the compiler macro __DIALECT_HARBOUR__
- Fixed the name resolution between types, namespaces, fields, properties, methods, globals etc. The core dialect is very close to the C# rules, the other dialect follows the VO rules.
- Some warnings for ambiguous code have been added
- _Chr() with untyped numeric values would crash. This has been fixed.
- We made some changes to the character literal rules. For the VO and Harbour dialect there are now other rules then for Core and Vulcan. See the help topic for more information

VO and Vulcan Dialect

- Several VO compatibility issues have been fixed
- The QUIT, ACCEPT, WAIT, DEFAULT TO and STORE command are now removed from the compiler and defined in our standard header file "XSharpDefs.xh" which is located in the \Program Files(x86)\XSharp\Include folder. These commands are not compiled in the core dialect
- Added support for CONSTRUCTOR() CLASS MyClass and DESTRUCTOR CLASS MyClass (in other words, outside the CLASS .. ENDCLASS construct
- The # (not equal) operator is now recognized when used without space before the keywords NIL, NULL_STRING, NULL_OBJECT etc. so #NIL is not seen as the symbol NIL but as Not Equal To NIL
- SizeOf and _TypeOf were special tokens in VO and could not be abbreviated. We have changed the X# behavior to match this. This prevents name conflicts with variables such as _type.

- We have added support for DLL entrypoints with embedded @ signs, such as "CAVOADAM.AdamCleanupProtoType@12"
- (DWORD) (-1) would require the unchecked operator. This is now compatible with Vulcan and generates a DWORD with the value System.Uint32.MaxValue.
- STATIC VOSTRUCT now gets compiled as INTERNAL VOSTRUCT. This means that you cannot have the same structure twice in your app. Why would you want to do that ?
- Fixed several cases of "incorrect" code that would be compiled by VO, such as a codeblock that looks like:
  cb := { |x|, x[1] == 1 }
  Note the extra comma.
  This now compiled into the same codeblock as:
  cb := { |x| x[1] == 1 }
- The /vo16 compiler option has been disabled for now (does not do anything) because it had too many side effects.

Visual Studio Integration

- Deleted files and folders are moved them to the Trash can.
- Fixed an intellisense problem in the XAML editor
- Added support for Code Completion between different X# projects
- Added support for Code Completion and other intellisense features for source code in VB and C# projects
- Added support for parameter info

Documentation

- We have added (generated) topics for all undocumented compiler errors. Some topics only contain the text that is shown by the compiler. More documentation will follow. Also some documentation for the X# Scripting has been added.

Changes in 0.2.11

Compiler

All dialects

- Improved some error messages, such as for unterminated strings
- Added support for the /s (Syntax Check only) command line option
- Added support for the /parseonly command line option which is used by the intellisense parser
- Added some compiler errors and warnings for invalid code
- The preprocessor did not properly handle 4 letter abbreviations for #command and #translate. This has been fixed
- Fixed some problems found with the preprocessor

- We switched to a new Antlr parser runtime. This should result in slightly better performance.
- Changed the way literal characters and strings are defined:
  - o In the Vulcan dialect a literal string that is enclosed with single quotes is a char literal. Double quotes are string literals
  - o In the Core and VO dialect a literal string that is enclosed with single quotes is a string literal. Double quotes are also string literals.
    To specify a char literal in Core and VO you need to prefix the literal with a 'c':

```
LOCAL cChar as CHAR
cChar := c'A'
```

- Changed the way literal characters and strings are defined:
- **sizeof**() and **_sizeof**() no longer generate a warning that they require 'unsafe' code, when compiling for x86 or x64. When compiling for AnyCpu the warning is still produced.
- When the includedir environment variable was not set then the XSharp\Include folder would also not be found automatically.

VO/Vulcan Compatibility

- Added /vo16 compiler option to automatically generate constructors with Clipper calling convention for classes without constructor

Harbour Compatibilty

- Started work on the Harbour dialect. This is identical with the VO/Vulcan dialect. The only difference so far is that the IIF() expressions are optional

Visual Studio

New features / changed behavior:

- Added Brace Matching
- Added Peek definition (Alt-F12)
- All keywords are not automatically part of the Completionlist
- Fixed a member lookup problem with Functions and Procedures inside a Namespace
- Increased background parser speed for large projects
- Fixed type lookup for fields and properties from parent classes
- Fixed problem where CSharp projects could not find the output of a XSharp project reference
- The Intellisense parser now properly used all current projects compiler options.
- Prevent crashes when the X# language buffer is fed with "garbage" such as C# code

Installer

- The local template cache and components cache for VS2017 was not cleared properly this has been fixed.
- Added code to properly unregister an existing CodeDomProvider when installing

Documentation

- Several empty chapters are now hidden.
- Added description of the templates

Changes in 0.2.10

This build focuses on the last remaining issues in the VO and Vulcan compatibility and adds a lot of new features to the Visual Studio integration.

Compiler

VO/Vulcan Compatibility

- We have completed support for the **DEFINE** keyword. The type clause is now optional. The compiler will figure out the type of the define when no type is specified.
  The DEFINEs will be compiled to a Constant field of the Functions class, or a Readonly Static field, when the expression cannot be determined at compile time (such as for Literal dates or symbols).
- We have extended the **preprocessor** . It now has support for #command, #translate, #xcommand and #xtranslate. Also "Pseudo function" defines are supported, such as :

  ```
  #define MAX(x,y) IIF((x) > (y), (x), (y))
  ```

  This works just like a #xtranslate, with the exception that the define is case sensitive (unless you have enabled the "VO compatible preprocessor" option (/vo8).
  The only thing that is *not working* in the preprocessor is the *repeated result marker*.
- In VO/Vulcan mode the compiler now accepts "**garbage**" between keywords such as ENDIF and NEXT and the end of the statement, just like the VO compiler.
  So you no longer have to remove "comment" tokens after a NEXT or ENDIF. This will compile without changes in the VO and Vulcan dialect:

  ```
  IF X == Y
      DoSomething()
  ENDIF X == Y
  ```

or

```
    FOR I := 1 to 10
        DoSomething()
    NEXT I
```

We do not recommend this coding style, but this kind of code is very common...

- Fixed an issue with recognition of single quoted strings. These were always recognized as CHAR_CONST when the length of the string was 1. Now they are treated as STRING_CONST and the compiler backend has been adjusted to convert the STRING literals to CHAR literals when needed.
- In VO and Vulcan dialect when the compiler option /vo1 is used then RETURN statements without value or with a return value of SELF are allowed for the Init() and Axit() methods. Other return values will trigger a compiler warning and will be ignored.

New features / changed behavior:

- The compiler now produces an error when a source file ends with an **unterminated multi line comment**
- Added **ASTYPE** expression, similar to the AS construct in other languages. This will assign a value of the correct type or NULL when the expression is not of the correct type:

```
VAR someVariable := <AnExpression> ASTYPE <SomeType>
```

- The **Chr()** and **_Chr()** functions are now converted to a string or character literal when the parameter is a compile time constant
- **Compilation speed** for assemblies with larger numbers of functions, procedures, defines, globals or _dll functions has been improved.
- **_DLL FUNCTIONS** now automatically are marked with CharSet.Auto
- Fixed some inconsistencies between Colon (:) and Point (.) interoperability and the super keyword
- Fixed several compiler issues reported by FOX subscribers and other users.

Visual Studio

New features / changed behavior:

- Tested and works with the release version of **Visual Studio 2017**
- We have added support for **regions** inside the VS editor. At this moment most "entities" are collapsible as well as statement blocks, regions and lists of usings, #includes and comments.
- We have added support for **member and type drop downs** in the VS Editor
- We have added support for **Code completion** in the VS editor
- We have added support for **Goto definition** in the VS Editor
- Errors detected by the intellisense scanner are now also included in the **VS error list.**
- We have added **help links** to the errors in the VS error list. The help links will bring you to the appropriate page on the X# website. Not all the help pages are complete yet, but at least the infrastructure is working.
- We have added support for **snippets** and included several code snippets in the installer

- We have made several changes to the **project properties dialogs**
  - The **pre** and **post build** events are now on a separate page for the Project Properties. These are now also <u>not</u> defined <u>per configuration</u> but are shared between the various configurations.
    If you want to copy output results to different folders for different configurations you should use the $(Configuration) and $(Platform) variables
  - We have moved the **Platform** and **Prefer32Bits** properties to the Build page to make them configuration dependent
  - Fixed a problem with casing of the **AnyCPU** platform which would result in duplicate items in the VS Platform combobox
  - Added support for **ARM** and **Itanium** platform types
  - Some properties were saved in project file groups without a **platform identifier**. This has been fixed
  - We have added a project property to control how Managed file resources are included: **Use Vulcan Compatible Managed Resources**
    When 'True' then resources files are included in the assembly without namespace prefix. When 'False' then the resource files are prefixed with the namespace of the app, just like in other .Net languages, such as C#
- We have fixed some **code generation** problems
- The parser that is used in the Windows Forms editor now also properly handles **background images**. Both images in the resx for the form and also background images in the shared project resources
- We have added **Nuget** support for our project system.
- We have made several changes to fix problems in project files
  - The project system now silently fixes problems with **duplicate items**
  - Fixed a problem with **dependencies** between xaml files and their dependent designer.prg files and other dependent files
  - Fixed a problem with **dependent items** in **sub folders** or in a folder tree that includes a dot in a folder name.
  - Fixed a problem in the **WPF template**
- Fixed a refresh problem when **deleting** a **references** node
- Added implementation of the OAProject.Imports property, which is used by **JetBrains**

XPorter

- Fixed a problem converting **WPF style projects**

Changes in 0.2.9

Compiler

With this build you can compile the Vulcan SDK without changes, except for some obvious errors in the Vulcan SDK that Vulcan did not find!

We consider the Vulcan Compatibility of the compiler finished with the current state of the compiler. All Vulcan code should compile without proble now.

VO/Vulcan Compatibility

New features / changed behavior:

- All Init procedures are now properly called at startup. So not only the init procedures in the VOSDK libraries but also init procedures in other libraries and the main exe
- Changed the method and type resolution code:
  - A method with a single object parameter is now preferred over a method with an Object[] parameter
  - When both a function (static method) exists and an instance method we will now call the static method in code inside methods that does not have a SELF: or SUPER: prefix.
  - In situations where the @ operator is used to pass variables by reference.
  - To make it more compatible with Vulcan for overloads with different numeric types.
  - To prefer a method with specific parameters over a method with usual parameters
  - To avoid problems with Types and Namespaces with the same name.
  - To prefer a method with an OBJECT parameter over the one with OBJECT[] parameters when only 1 argument is passed
  - When 2 identical functions or types are detected in referenced assemblies we now choose the one in the first referenced assembly like Vulcan does, and generate warning 9043
- The **sizeof** operator now returns a DWORD to be compatible with VO and Vulcan.
- Added support for **EXIT PROCEDURES** (PROCEDURE MyProcedure EXIT). These procedures will automatically be called during program shutdown, just before all the global variables are cleared.
The compiler now generates an $Exit function for each assembly in which the exit procedures will be called and the reference globals in an assembly will be cleared. In the main app a $AppExit() function is created that will call the $Exit functions in all references X# assemblies. When a Vulcan compiled assembly is referenced, then all the public reference globals will be cleared from the $AppExit() function.
- Added support for **PCALL** and **PCALLNATIVE**
- Added support for several Vulcan compatible compiler options:
  - **/vo1** Allow Init() and Axit() for constructor and destruction
  - **/vo6** Allow (global) function pointers. DotNet does not "know" these. They are compiled to IntPtr. The function information is preserved so you can use these pointer in a PCALL()
  - **/ppo**. Save the preprocessed compiler output to a file
  - **/Showdefs** Show a list of the defines and their values on the console
  - **/showincludes** Show a list of the included header files on the console
  - **/verbose** Shows includes, source file names, defines and more. on the console
  - **DEFAULT TO** command
  - **ACCEPT** command
  - **WAIT** command
- Several code generation changes:

- o Changed the code generation for DIM elements inside VOStruct arrays because the Vulcan compiler depends on a specific naming scheme and did not recognize our names.
- o Improved the code generation inside methods with CLIPPER calling convention.

Bug fixes

- Implicit namespaces are now only used when the /ins compiler option is enabled. In Vulcan dialect the namespace Vulcan is always included.
- Fixed several problems with the @ operator and VOSTRUCT types
- Fixed a problem with DIM arrays of VOSTRUCT types
- Fixed a problem with LOGIC values inside VOSTRUCT and UNION types
- Fixed several problems with the VOStyle _CAST and Conversion operators.
- Fixed several numeric conversion problems
- Fixed several problems when mixing NULL, NULL_PTR and NULL_PSZ
- Fixed several problems with the _CAST operator
- Fixed several problems with PSZ Comparisons. X# now works just like Vulcan and VO and produces the same (sometimes useless) results
- Fixed a problem with USUAL array indexes for multi dimensional XBase Arrays
- Fixed codeblock problems for codeblocks where the last expression was VOID
- Changed code generation for NULL_SYMBOL
- Preprocessor #defines were sometimes conflicting with class or namespace names. For example when /vo8 was selected the method System.Diagnostics.Debug.WriteLine() could not be called because the DEBUG define was removing the classname. We have changed the preprocessor so it will no longer replace words immediately before or after a DOT or COLON operator.
- Fixed a compiler crash when calling static methods in the System.Object class when Late Binding was enabled
- Fixed String2Psz() problem inside PROPERTY GET and PROPERTY SET
- And many more changes.

All dialects

New features / changed behavior:

- Several code generation changes:
  - o The code generation for ACCESS and ASSIGN has changed. There are no longer separate methods in the class, but the content of these methods is now inlined in the generated Get and Set methods for the generated property.
  - o Optimized the code generation for **IIF** statements.
  - o The **debugger/step** information has been improved. The debugger should now also stop on IF statements, FOR statements, CASE statements etc.
- Indexed access to properties defined with the SELF keyword can now also use the "**Index**" property name
- Functions and Procedures inside classes are not allowed (for now)
- RETURN <LiteralValue> inside an ASSIGN method will no longer allocate a variable and produce an warning
- Several keywords are now also allowed as Identifier (and will no longer have to be prefixed with @@):

Delegate, Enum, Event, Field, Func, Instance, Interface, Operator, Proc, Property, Structure, Union, VOStruct and many more

As a result the following is now valid code (but not recommended):

```
FUNCTION Start AS VOID
   LOCAL INTERFACE AS STRING
   LOCAL OPERATOR AS LONG
   ? INTERFACE, OPERATOR
   RETURN
```

You can see that the Visual Studio language support also recognizes that INTERFACE and OPERATOR are not used as keywords in this context

Bug fixes

- Fixed a problem with the REPEAT UNTIL statement
- Fixed a crash for code with a DO CASE without a matching END CASE
- Fixed several issues for the code generation for _DLL FUNCTIONs and _DLL PROCEDUREs
- Fixed a problem (in the Roslyn code) with embedding Native Resources in the Assembly.
- Fixed a problem with the _OR() and _AND() operators with more than 2 arguments.
- Added support for Pointer dereferencing using the VO/Vulcan Syntax : DWORD(p) => p[1]
- Fixed several problems with the @ operator
- When two partial classes had the same name and a different casing the compiler would not properly merge the class definitions.
- Fixed a crash when a #define in code was the same as a define passed on the commandline
- Indexed pointer access was not respecting the /AZ compiler option (and always assumed 0 based arrays). This has been fixed
- Fixed a problem with the caching of preprocessed files, especially files that contain #ifdef constructs.
- Fixed a problem which could occur when 2 partial classes had the same name but a different case
- Fixed a compiler crash when a referenced assembly had duplicate namespaces that were only different in Case
- Fixed problems with Functions that have a [DllImport] attribute.
- Error messages for ACCESS/ASSIGN methods would sometimes point to a strange location in the source file. This has been fixed.
- Fixed a problem with Init Procedures that had a STATIC modifier
- Fixed a problem in the preprocessor when detecting the codepage for a header file. This could cause problems reading header files with special characters (such as the copyright sign ©)
- And many more changes.

Visual Studio Integration

- Added support for all compiler options in the UI and the build system
- Fixed problems with dependent file items in subfolders
- The Optimize compiler option was not working

- The 'Clean' build option now also cleans the error list
- Under certain conditions the error list would remain empty even though there were messages in the output pane. This has been fixed.
- The <Documentationfile> property inside the xsproj file would cause a rebuild from the project even when the source was not changed
- Earlier versions of XPorter could create xsproj files that would not build properly. The project system now fixes this automatically
- Fixed a problem with the build system and certain kind of embedded managed resources

Documentation

- We have added many descriptions to the commandline options
- We have added a list of the most common compiler errors and warnings.

Changes in 0.2.8

Compiler

VO/Vulcan Compatibility

- Default Parameters are now handled like VO and Vulcan do. This means that you can also have date constants, symbolic constants etc as default parameter
- String Single Equals rules are now 100% identical with Visual Objects. We found one case where Vulcan does not return the same result as Visual Objects. We have chosen to be compatible with VO.
- When compiling in VO/Vulcan mode then the init procedures in the VO SDK libraries are automatically called. You do not have to call these in your code anymore
  Also Init procedures in the main assembly are called at startup.
- The /vo7 compiler option (Implicit casts and conversions) has been implemented. This also includes support to use the @ sign for REF parameters
- You can now use the DOT operator to access members in VOSTRUCT variables
- We have fixed several USUAL - Other type conversion problems that required casts in previous builds
- The compiler now correctly parses VO code that contains DECLARE METHOD, DECLARE ACCESS and DECLARE ASSIGN statements and ignores these
- The compiler now parses "VO Style" compiler pragma's (~"keyword" as white-space and ignores these.
- Fixed a problem where arrays declared with the "LOCAL aSomething[10] AS ARRAY" syntax would not be initialized with the proper number of elements
- Fixed a problem when calling Clipper Calling Convention constructors with a single USUAL parameter

- Attributes on _DLL entities would not be properly compiled. These are recognized for now but ignored.
- Fixed several numeric conversion problems

New features

- We have added support for Collection Initializers and Object Initializers
- Anonymous type members no longer have to be named. If you select a property as an anonymous type member then the same property name will be used for the anonymous type as well.
- Missing closing keywords (such as NEXT, ENDIF, ENDCASE and ENDDO) now produce better error messages
- IIF() Expressions are now also allowed as Expression statement. The generated code will be the same as if an IF statement was used

```
FUNCTION IsEven(nValue as LONG) AS LOGIC
  LOCAL lEven as LOGIC
  IIF( nValue %2 == 0, lEven := TRUE, lEven := FALSE)
RETURN lEven
```

We really do not encourage to hide assignments like this, but in case you have used this coding style,it works now <g>.
- AS VOID is now allowed as (the only) type specification for PROCEDUREs
- We have added a .config file to the exe for the shared compiler that should make it faster
- The XSharpStdDefs.xh file in the XSharp is now automatically included when compiling. This file declares the CRLF constant for now.
- Include files are now cached by the compiler. This should increase the compilation speed for projects that depend on large included files, such as the Win32APILibrary header file from Vulcan
- When a function is found in an external assembly and a function with the same name and arguments is found in the current assembly, then the function in the current assembly is used by the compiler
- Compiler error messages for missing closing symbols should have been improved
- Compiler error messages for unexpected tokens have been improved

Bug fixes

- Several command-line options with a minus sign were not properly handled by the compiler
- Fixed several crashes related to assigning NULL_OBJECT or NULL to late bound properties have been fixed
- Partial class no longer are required to specify the parent type on every source code location. When specified, the parent type must be the same of course. Parent interfaces implemented by a class can also be spread over multiple locations
- We have fixed a crash that could happen with errors/warnings in large include files
- Abstract methods no longer get a Virtual Modifier with /vo3
- Fixed a problem with virtual methods in child classes that would hide parent class methods

- Automatic return value generation was also generating return values for ASSIGN methods. This has been fixed.
- We fixed a problem with the Join Clauses for LINQ Expressions that would cause a compiler exception
- The /vo10 (compatible iif) compiler option no longer adds casts in the Core dialect. It only does that for the VO/Vulcan dialect

Visual Studio Integration

We have changed the way the error list and output window are updated. In previous version some lines could be missing on the output window, and the error code column was empty. This should work as expected now.

- We have merged some code from some other MPF based project systems, such as WIX (called Votive), NodeJS and Python (PTVS) to help extend our project system. As a result:
  o Our project system now has support for Linked files
  o Our project system now has support for 'Show All Files' and you can now Include and Exclude files. This setting is persisted in a .user file, so you can exclude this from SCC if you want.
  o We have made some changes to support better 'Drag and Drop'
- We have fixed several issues with regard to dependent items
- When you include a file that contains a form or user control, this is now recognized and the appropriate subtype is set in the project file, so you can open the windows forms editor
- We are now supporting source code generation for code behind files for .Settings and .Resx files
- The combobox in the Managed Resource editor and Managed Settings tool to choose between internal code and public code is now enabled. Selecting a different value in the combobox will change the tool in the files properties.
- The last response file for the compiler and native resource compiler are now saved in the users Temp folder to aid in debugging problems.
- The response file now has each compiler option to a new line to make it easier to read and debug when this is necessary.
- The code generation now preserves comments between entities (methods)
- We fixed several minor issues in the templates
- When the # of errors and warnings is larger than the built-in limit of 500, then a message will be shown that the error list was truncated
- At the end of the build process a line will be written to the output window with the total # of warnings and errors found
- The colors in the Source Code Editor are now shared with the source code editors for standard languages such as C# and VB
- When you have an inactive code section in your source code, embedded in an #ifdef that evaluates to FALSE then that section will be visibly colored gray and there will be no keyword highlighting. The source code parser for the editor picks up the include files and respects the path settings. Defines in the application properties dialog and the active configuration are not respected yet. That will follow in the next build.

Changes in 0.2.7.1

Compiler

- The compiler was not accepting wildcard strings for the AssemblyFileVersion Attribute and the AssemblyInformationVersion attribute. This has been fixed
- The #Pragma commands #Pragma Warnings(Push) and #Pragma Warnings(Pop) were not recognized. This has been fixed.
- The compiler was not recognizing expressions like global::System.String.Compare(..). This has been fixed

Visual Studio Integration

- Dependent items in subfolders of a project were not recognized properly and could produce an error when opening a project
- Fixed a problem in the VulcanApp Template
- The Windows Forms Editor would not open forms in a file without begin namespace .. end namespace. This has been fixed
- Source code comments between 'entities' in a source file is now properly saved and restored when the source is regenerated by the form editor
- Unnecessary blank lines in the generate source code are being suppressed
- The XPorter tool is now part of the Installation
- Comments after a line continuation character were not properly colored
- Changed the XSharp VS Editor Color scheme to make certain items easier to read
- New managed resource files would not be marked with the proper item type. As a result the resources would not be available at runtime. This has been fixed.
- Added 'Copy to Output Directory' property to the properties window

Setup

- The installer, exe files and documentation are now signed with a certificate

Changes in 0.2.7

Compiler

New features:

- Added support for the VOSTRUCT and UNION types

- Added support for Types as Numeric values, such as in the construct
  ```
  IF UsualType(uValue) == LONG
  ```
- Added a FIXED statement and FIXED modifier for variables
- Added support for Interpolated Strings
- Empty switch labels inside SWITCH statements are now allowed. They can share the implementation with the next label.
  Error 9024 (EXIT inside SWITCH statement not allowed) has been added and will be thrown if you try to exit out of a loop around the switch statement.
  This is not allowed.
- Added support for several /vo compiler options:
  - vo8 (Compatible preprocessor behavior). This makes the preprocessor defines case insensitive. Also a define with the value FALSE or 0 is seen as 'undefined'
  - vo9 (Allow missing return statements) compiler option. Missing return values are also allowed when /vo9 is used.
  Warnings 9025 (Missing RETURN statement) and 9026 (Missing RETURN value) have been added.
  - vo12 (Clipper Integer divisions)
- The preprocessor now automatically defines the macros __VO1__ until __VO15__ with a value of TRUE or FALSE depending on the setting of the compiler option
- The FOX version of the compiler is now distributed in Release mode and much faster. A debug version of the compiler is also installed in case it is needed to aid in finding compiler problems.

Changed behavior

- The compiler generated Globals class for the Core dialect is now called **Functions** and no longer Xs$Globals.
- Overriding functions in VulcanRTFuncs can now be done without specifying the namespace:
  When the compiler finds two candidate functions and one of them is inside VulcanRTFuncs then the function that is not in VulcanRTFuncs is chosen.
- Warning 9001 (unsafe modifier implied) is now suppressed for the VO/Vulcan dialect. You MUST pass the /unsafe compiler option if you are compiling unsafe code though!
- Improved the error messages for the Release mode of the compiler

Bug fixes

- RETURN and THROW statements inside a Switch statement would generate an 'unreachable code' warning. This has been fixed
- Fixed several problems with mixing signed and unsigned Array Indexes
- Fixed several problems with the FOR .. NEXT statement. The "To" expression will now be evaluated for every iteration of the loop, just like in VO and Vulcan.
- Fixed several compiler crashes
- Fixed a problem with implicit code generation for constructors
- Fixed a visibility problem with static variables inside static functions

Visual Studio Integration

- Fixed a problem that the wrong Language Service was selected when XSharp and Vulcan.NET were used in the same Visual Studio and when files were opened from the output window or the Find Results window

- Fixed some problems with 'abnormal' line endings in generated code
- Fixed a problem in the Class Library template
- Fixed a problem with non standard command lines to Start the debugger

Changes in 0.2.6

Compiler

- Added alternative syntax for event definition. See <u>EVENT</u> keyword in the documentation
- Added Code Block Support
- Implemented /vo13 (VO compatible string comparisons)
- Added support for /vo4 (VO compatible implicit numeric conversions)
- Aliased expressions are now fully supported
- Fixed a problem with the &= operator
- Fixed several crashes for incorrect source code.
- Fixed several problems related to implicit conversions from/to usual, float and date
- Indexed properties (such as String:Chars) can now be used by name
- Indexed properties can now have overloads with different parameter types
- Added support for indexed ACCESS and ASSIGN
- Fixed a problem when calling Clipper Calling Convention functions and/or methods with a single parameter
- Fixed a crash with defines in the preprocessor
- _CODEBLOCK is now an alias for the CODEBLOCK type
- Fixed a crash for properties defined with parentheses or square brackets, but without actual parameters

Visual Studio Integration

- Completed support for .designer.prg for Windows.Forms
- Fixed an issue in the CodeDom generator for generating wrappers for Services
- The XSharp Language service will no longer be used for Vulcan PRG files in a Side by Side installation
- Editor performance for large source files has been improved.
- All generated files are now stored in UTF, to make sure that special characters are stored correctly. If you are seeing warnings about code page conversions when generating code, then save files as UTF by choosing "File - Advanced Save Options", and select a Unicode file format, from the Visual Studio Menu.

Changes in 0.2.51

Visual Studio Integration & Build System

- The Native Resource compiler now "finds" header files, such as "VOWin32APILibrary.vh" in the Vulcan.NET include folder. Also the output of the resource compiler is now less verbose when running in "normal" message mode. When running in "detailed" or "diagnostics" mode the output now also includes the verbose output of the resource compiler.

Compiler

- Fixed a problem that would make PDB files unusable
- The error "Duplicate define with different value" (9012) has been changed to warning, because our preprocessor does a textual comparison and does not "see" that "10" and "(10)" are equal as well as "0xA" and "0xa". It is your responsibility of course to make sure that the values are indeed the same.
- Exponential REAL constants were only working with a lower case 'e'. This is now case insensitive
- Made several changes to the _DLL FUNCTION and _DLL PROCEDURE rules for the parser. Now we correctly recognize the "DLL Hints " (#123) and also allow extensions in these definitions. Ordinals are parsed correctly as well, but produce an error (9018) because the .Net runtime does not support these anymore. Also the Calling convention is now mandatory and the generated IL code includes SetLastError = true and ExactSpelling = true.
- Fixed a problem with the ~ operator. VO and Vulcan (and therefore X#) use this operator as unary operator and as binary operator.
  The unary operator does a bitwise negation (Ones complement), and the binary operator does an XOR.
  This is different than C# where the ~ operator is Bitwise Negation and the ^ operator is an XOR (and our Roslyn backend uses the C# syntax of course).

Changes in 0.2.5

Visual Studio Integration

- Fixed a problem where the output file name would contain a pipe symbol when building for WPF
- Fixed a problem with the Item type for WPF forms, pages and user controls
- The installer now has an option to not take away the association for PRG, VH and PPO items from an installed Vulcan project system.

- Added support for several new item types in the projects
- Added support for nested items
- Added several item templates for WPF, RC, ResX, Settings, Bitmap, Cursor etc.

Build System

- Added support for the new /vo15 command line switch.
- Added support for compiling native resources.

Compiler

- A reference to VulcanRT and VulcanRTFuncs is now mandatory when compiling in VO/Vulcan dialect
- Added support for indexed access for VO/Vulcan Arrays
- Added support for VO/Vulcan style Constructor chaining (where SUPER() or SELF() call is not the first call inside the constructor body)
- Added support for the &() macro operator in the VO/Vulcan dialect
- Added support for the FIELD statement in the VO/Vulcan dialect
  - The statement is recognized by the compiler
  - Fields listed in the FIELD statement now take precedence over local variables or instance variables with the same name
- Added support for the ALIAS operator (->) in the VO/Vulcan dialect, with the exception of the aliased expressions (AREA->(<Expression>))
- Added support for Late bound code (in the VO/Vulcan dialect)
  - Late bound method calls
  - Late bound property get
  - Late bound property set
  - Late bound delegate invocation
- Added a new /vo15 command line option (Allow untyped Locals and return types): By default in the VO/Vulcan dialect missing types are allowed and replaced with the USUAL type.
When you specify /vo15- then untyped locals and return types are not allowed and you must specify them.
Of course you can also specify them as USUAL
- The ? and ?? statement are now directly mapped to the appropriate VO/Vulcan runtime function when compiling for the VO/Vulcan dialect
- We now also support the VulcanClassLibrary attribute and VulcanCompilerVersion attribute for the VO & Vulcan dialect.
With this support the Vulcan macro compiler and Vulcan Runtime should be able to find our functions and classes
- The generated static class name is now more in par with the class name that Vulcan generates in the VO & Vulcan dialect.
- Added several implicit conversion operations for the USUAL type.
- When accessing certain features in the VO & Vulcan dialect (such as the USUAL type) the compiler now checks to see if VulcanRTFuncs.DLL and/or VulcanRT.DLL are included.
When not then a meaningful error message is shown.
- Added support for the intrinsic function _GetInst()
- Fixed a problem with case sensitive namespace comparisons
- Fixed a problem with operator methods

- Added preprocessor [macros](#) __DIALECT__, __DIALECT_CORE__, __DIALECT_VO__ and __DIALECT_VULCAN__
- The _Chr() pseudo function will now be mapped to the Chr() function
- Added support for missing arguments in arguments lists (VO & Vulcan dialect only)
- Fixed a crash when calculating the position of tokens in header files
- The installer now offers to copy the Vulcan Header files to the XSharp Include folder
- Added support for skipping arguments in (VO) literal array constructors

Documentation

- Added the XSharp documentation to the Visual Studio Help collection
- Added reference documentation for the Vulcan Runtime

Changes in 0.2.4

Visual Studio Integration

- Double clicking errors in the error browser now correctly opens the source file and positions the cursor
- Fixed several problems in the project and item templates
- The installer now also detects Visual Studio 15 Preview and installs our project system in this environment.

Build System

- Fixed a problem with the /unsafe compiler option
- Fixed a problem with the /doc compiler option
- Treat warnings as error was always enabled. This has been fixed.

Compiler

- Added support for Lambda expressions with an expression list
```
LOCAL dfunc AS System.Func<Double,Double>
dfunc := {|x| x := x + 10, x^2}
? dfunc(2)
```
- Added support for Lambda expressions with a statement list
```
LOCAL dfunc AS System.Func<Double,Double>
dfunc :=       {|x|
                    ? 'square of', x
                    RETURN x^2
               }
```
- Added support for the NAMEOF intrinsic function
FUNCTION Test(cFirstName AS STRING) AS VOID
```
FUNCTION Test(cFirstName AS STRING) AS VOID
```

```
        IF String.IsNullOrEmpty(cFirstName)
                THROW ArgumentException{"Empty argument", nameof(cFirstName)}
        ENDIF
```

- Added support for creating methods and functions with Clipper calling convention (VO and Vulcan dialect only)
- Using Statement now can contain a Variable declaration:
  Instead of:
  ```
  VAR ms := System.IO.MemoryStream{}
      BEGIN USING ms
    // do the work

  END USING
  ```
  You can now write
  ```
  BEGIN USING VAR ms := System.IO.MemoryStream{}
        // do the work
  END USING
  ```
- Added support for /vo10 (Compatible IIF behavior). In the VO and Vulcan dialect the expressions are cast to USUAL. In the core dialect the expressions are cast to OBJECT.

New language features for the VO and Vulcan dialect

- Calling the SELF() or SUPER() constructor is now allowed anywhere inside a constructor (VO and Vulcan dialect only). The Core dialect still requires constructor chaining as the first expression inside the constructor body
- Added support for the PCOUNT, _GETFPARAM and _GETMPARAM intrinsic functions
- Added support for String2Psz() and Cast2Psz()
- Added support for BEGIN SEQUENCE … END
- Added support for BREAK

Fixed several problems:

- Nested array initializers
- Crash for BREAK statements
- Assertion error for generic arguments
- Assertion on const implicit reference
- Allow ClipperCallingConvention Attribute on Constructors, even when it is marked as 'for methods only'
- Fixed a problem with Global Const declarations
- __ENTITY__ preprocessor macro inside indexed properties

Changes in 0.2.3

Visual Studio Integration

- We have changed to use the MPF style of Visual Studio Integration.

- We have added support for the Windows Forms Editor
- We have added support for the WPF Editor
- We have added support for the Codedom Provider, which means a parser and code generator that are used by the two editors above
- The project property pages have been elaborated. Many more features are available now.
- We have added several templates

Build System

- Added support for several new commandline options, such as /dialect
- The commandline options were not reset properly when running the shared compiler. This has been fixed.
- The build system will limit the # of errors passed to Visual Studio to max. 500 per project. The commandline compiler will still show all errors.

Compiler

- We have started work on the Bring Your Own Runtime support for Vulcan. See separate heading below.
- The __SIG__ and __ENTITY__ macros are now also supported, as well as the __WINDIR__, __SYSDIR__ and __WINDRIVE__ macros
- The debugger instructions have been improved. You should have a much better debugging experience with this build
- Several errors that indicated that there are visibility differences between types and method arguments, return types or property types have been changed into warnings. Of course you should consider to fix these problems in your code.
- The #Error and #warning preprocessor command no longer require the argument to be a string
- The SLen() function call is now inlined by the compiler (just like in Vulcan)
- The AltD() function will insert a call to "System.Diagnostics.Debugger.Break" within a IF System.Diagnostics.Debugger.IsAttached check
- Several compiler crashes have been fixed
- Added support for the PARAMS keyword for method and function parameters.
- Fixed a problem for the DYNAMIC type.

BYOR

- XBase type names are resolved properly (ARRAY, DATE, SYMBOL, USUAL etc)
- Literal values are now resolved properly (ARRAY, DATE, SYMBOL)
- NULL_ literals are resolved properly (NULL_STRING follows the /vo2 compiler option, NULL_DATE, NULL_SYMBOL)
- The /vo14 compiler option (Float literals) has been implemented
- The compiler automatically inserts a "Using Vulcan" and "using static VulcanRtFuncs.Functions" in each program
- You MUST add a reference to the VulcanRTFuncs and VulcanRT assembly to your project. This may be a Vulcan 3 and also a Vulcan 4 version of the Runtime. Maybe Vulcan 2 works as well, we have not tested it.
- Calling methods with Clipper calling convention works as expected.

- Methods/Functions without return type are seen as methods that return a USUAL
- If a method/function contains typed and typed parameters then the untyped parameters are seen as USUAL parameters
- Methods with only untyped parameters (Clipper calling convention) are not supported yet
- The ? command will call AsString() on the arguments

Changes in 0.2.2

Visual Studio Integration

- Added more project properties. One new property is the "Use Shared Compiler" option. This will improve compilation speed, but may have a side effect that some compiler (parser) errors are not shown in details.
  If you experience this, then please disable this option.
- Added more properties to the Build System. All C# properties should now also be supported for X#, even though some of them are not visible in the property dialogs inside VS.
- Added a CreateManifest task to the Build System so you will not get an error anymore for projects that contain managed resources
- The performance of the editor should be better with this release.
- Marking and unmarking text blocks as comment would not always be reflected in the editor colors. This has been fixed.

Compiler

- We have added a first version of the **preprocessor**. This preprocessor supports the #define command, #ifdef, #ifndef, #else, #endif, #include, #error and #warning. #command and #translate (to add user defined commands) are not supported yet.
- Missing types (in parameter lists, field definitions etc) were sometimes producing unclear error messages. We have changed the compiler to produce a "Missing Type" error message.
- We rolled the underlying Roslyn code forward to VS 2015 Update 1. Not that you see much of this from the outside <g>, but several fixes and enhancements have made their way into the compiler.
- Added a **YIELD EXIT** statement (You can also use **YIELD BREAK).**
- Added an (optional) **OVERRIDE** keyword which can be used as modifier on virtual methods which are overridden in a subclass.
- Added a **NOP** keyword which you can use in code which is intentionally empty (for example the otherwise branch of a case statement. The compiler will no longer warn about an empty block when you insert a NOP keyword there.
- The **On** and **Off** keywords could cause problems, because they were not positional (these are part of the pragma statement). This has been fixed.
- **_AND()** and **_OR()** expressions with one argument now throw a compiler error.
- The compiler now recognizes the /VO14 (store literals as float) compiler switch (it has not been implemented yet).

- Added a ****** operator as alias for the **^** (Exponent) operator.
- Added an "unsupported" error when using the Minus operator on strings.
- Fixed a "Stack overflow" error in the compiler that could occur for very long expressions.
- The right shift operator no longer conflicts with two Greater Than operators, which allows you to declare or create generics without having to put a space between them.
  (var x := List<Tuple<int,int>>{}

Changes in 0.2.1

Visual Studio Integration

- Added and improved several project properties
- Fix a problem with the "Additional Compiler Options"
- Improved coloring in the editor for Keywords, Comments etc. You can set the colors from the Tools/Options dialog under General/Fonts & Colors. Look for the entries with the name "XSharp Keyword" etc.
- Added Windows Forms Template

Compiler

- Several errors have been demoted to warnings to be more compatible with VO/Vulcan
- Added support for Comment lines that start with an asterisk
- Added support for the DEFINE statement. For now the DEFINE statement MUST have a type
  DEFINE WM_USER := 0x0400 AS DWORD
- Fixed problem with Single Line Properties with GET and SET reversed
- Several fixes for Virtual and Non virtual methods in combination with the /VO3 compatibility option

Changes in 0.1.7

- The "ns" (add default namespace to classes without namespace) has been implemented
- The "vo3" compiler option (to make all methods virtual) has been implemented
- Fixed an issue where the send operator on an expression between parentheses was not compiling properly
- Relational operators for strings (>, >=, <, <=) are now supported. They are implemented using the String.Compare() method.

- Fixed a problem with local variables declared on the start line from FOR .. NEXT statements
- Added first version of the documentation in CHM & PDF format
- Added several properties to the Visual Studio Project properties dialog to allow setting the new compiler options
- Fixed a problem in the Targets files used by MsBuild because some standard macros such as $(TargetPath) were not working properly
- XIDE 0.1.7 is included. This version of XIDE is completely compiled with XSharp !
- The name of some of the MsBuild support files have changed. This may lead to problems loading a VS project if you have used the VS support from the previous build. If that is the case then please edit the xsproj file inside Visual Studio and replace all references of "XSharpProject" with "XSharp" . Then safe the xsproj file and try to reload the project again
- The WHILE.. ENDDO (a DO WHILE without the leading DO) is now recognized properly

Changes in 0.1.6

- This version now comes with an installer
- This version includes a first version of the Visual Studio Integration. You can edit, build, run and debug inside Visual Studio. There is no "intellisense" available.
- The compiler now uses 1-based arrays and the "az" compiler option has been implemented to switch the compiler to use 0-based arrays.
- The "vo2" compiler option (to initialize string variables with String.Empty) has been implemented
- Please note that there is no option in the VS project properties dialog yet for the az and vo2 compiler options. You can use the "additional compiler options" option to specify these compiler options.
- The text "this" and "base" in error messages has been changed to "SELF" and "SUPER"
- Error of type "visibility" (for example public properties that expose private or internal types) have been changed to warnings
- Fixed a problem with TRY … ENDTRY statements without CATCH clause
- The compiler now has a better resolution for functions that reside in other (X#) assemblies
- Fixed a problem which could lead to an "ambiguous operator" message when mixing different numeric types.

Changes in 0.1.5

- When an error occurs in the parsing stage, X# no longer enters the following stages of the compiler to prevent crashes. In addition to the errors from the parser also an error 9002 is displayed.

- Parser errors now also include the source file name in the error message and have the same format as other error messages. Please note that we are not finished yet with handling these error messages. There will be improvements in the format of these error messages in the upcoming builds.
- The compiler will display a "feature not available" (8022) error when a program uses one of the Xbase types (ARRAY, DATE, FLOAT, PSZ, SYMBOL, USUAL).
- Fixed an error with VOSTRUCT and UNION types
- Fixed a problem with the exclamation mark (!) NOT operator

Changes in 0.1.4

- Several changes to allow calculations with integers and enums
- Several changes to allow VO compatible _OR, _AND, _NOT an _XOR operations
- Fix interface/abstract VO properties
- Insert an implicit "USING System" only if not explicitly declared
- Error 542 turned to warning (members cannot have the same name as their enclosing type)
- Changes in the .XOR. expression definition
- Fix double quote in CHAR_CONST lexer rule
- Allow namespace declaration in class/struct/etc. name (CLASS Foo.Bar)
- Fix access/assign crash where identifier name was a (positional) keword: ACCESS Value
- Preprocessor keywords were not recognized after spaces, but only at the start of the line. This has been fixed.
- Prevent property GET SET from being parsed as expression body
- Fix default visibility for interface event
- Unsafe errors become warnings with /unsafe option, PTR is void*
- Fix dim array field declaration
- Initial support of VO cast and VO Conversion rules (TYPE(_CAST, Expression) and TYPE(Expression)). _CAST is always unchecked (LONG(_CAST, dwValue)) and convert follows the checked/unchecked rules (LONG(dwValue))
- Fixed problem with codeblock with empty parameter list
- Fixed problems with GlobalAttributes.
- An AUTO property without GET SET now automatically adds a GET and SET block
- Allow implicit constant double-to-single conversion

Changes in 0.1.3

- Change inconsistent field accessibility error to warning and other similar errors
- Added commandline support for Vulcan arguments. These arguments no longer result in an error message, but are not really implemented, unless an equivalent argument exists for the Roslyn (C#) compiler. For example: /ovf and /fovf are both

mapped to /checked, /wx is mapped to /warnaserror. /w should not be used because that has a different meaning /warning level). /nowarn:nnnn should be used in stead

- Fixed problem where the PUBLIC modifier was assigned to Interface Members or Destructors
- Prevent expression statements from starting with CONSTRUCTOR() or DESTRUCTOR()
- Added support for ? statement without parameters
- The default return type for assigns is now VOID when not specified
- Added support for "old Style" delegate instantiation
- Added support for Enum addition
- Added an implicit empty catch block for TRY .. END TRY without catch and finally
- Added support for the DESCENDING keyword in LINQ statements
- Added support for VIRTUAL and OVERRIDE for Properties and Events
- Prevent implied override insertion for abstract interface members
- Fixed a problem where System.Void could not be resolved
- Fixed problem with Property Generation for ACCESS/ASSIGN
- Fixed problem with Abstract method handling

Changes in 0.1.2.1

- Added default expression
- Fixed problem with events
- Fixed some small lexer problems
- Fixed problem with _DLL FUNCTION and _DLL PROCEDURE

Changes in 0.1.2

- Fixed problem with handling escape sequences in extended strings
- Fixed issue in FOR.. NEXT statements
- Fixed a problem with SWITCH statements
- Fixed a problem with the sizeof() operator
- Fixed a problem in the REPEAT .. UNTIL statement
- Fixed a problem in TRY .. CATCH .. FINALLY .. END TRY statements.
- Fixed issue in Conditional Access Expression (Expr ? Expr)
- Allow bound member access of name with type args
- Fixed problem in LOCAL statement with multiple locals
- Fixed a problem when compiling with debug info for methods without a body
- Optimized the Lexer. This should increase the compile speed a lot
- Fixed a problem in the code that reports that a feature is not supported yet
- Fixed a problem when defining Generic types with a STRUCTURE constraint

- Compiler macros (__ENTITY__, __LINE__ etc) were causing a crash. For now the compiler inserts a literal string with the name of the macro.
- Build 0.1.1 did not contain XSC.RSP
- Fixed a problem where identifiers were not recognized when they were matching a (new) keyword

1.3   Migrating apps from VO to X#

In this topic we will discuss how to migrate applications from VO to X# with the aid of the VO XPorter tool from X#.

When you want to do this we strongly recommend you to start small. We will therefore start with a few VO example programs and describe the steps to take and the changes that need to be made after the conversion.

The examples

The examples on the following pages show you a step by step introduction to migrating your apps.

For each example there will be an example supplied with the installer of X#. The examples come with a "before" and "after" folder.

1.3.1   Example 1: The VO Explorer Example

Please note: This example assumes that you have installed X# with the default installation options, and that the X# runtime is installed in the GAC.
If you have not done that then the example won't run. Please see the chapter about this in this documentation

As first example of XPorting VO apps we take the VO Explorer example.
Start the VO Xporter program and point it to the VO Explorer example:
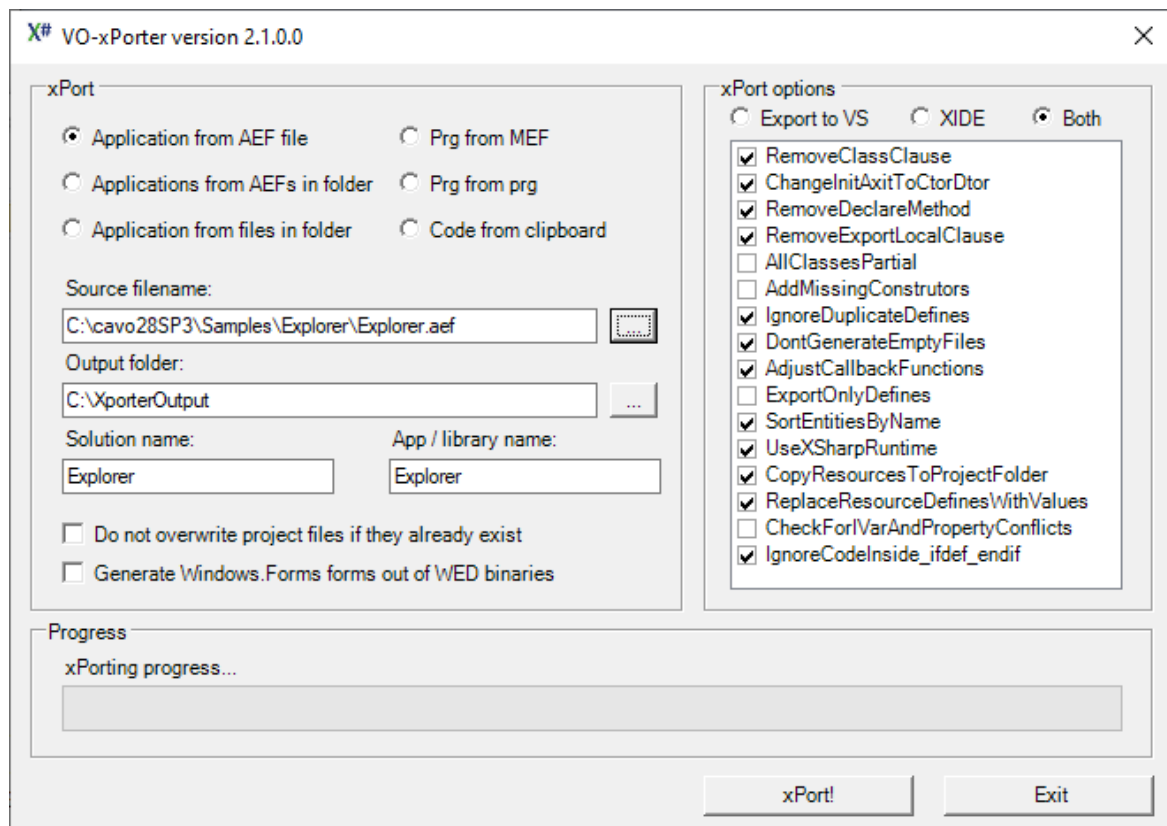
On my machine the VO installation is in the C:\Cavo28SP3 folder. That may be different on your machine of course.
I have not changed any or the xPort options. The options are:

Option	Description
RemoveClassClause	Remove the "Class Clause" for methods and accesses/assigns. This will group all methods inside a CLASS .. END CLASS block and then the CLASS clause at the method level is no longer required
ChangeInitAxitToCtorDtor	Init and Axit are reserved method names in VO. In DotNet these names are often used by classes. Vulcan and X# have introduced new keywords CONSTRUCTOR and DESTRUCTOR. This option automatically renames the Init and Axit Methods. Example:

```
METHOD Init(oOwner)  CLASS MyClass
```

becomes

```
CONSTRUCTOR(oOwner)
```

RemoveDeclareMethod	If you used strong typing in VO you needed to add DECLARE METHOD statements and/or DECLARE ACCESS and DECLARE ASSIGN. The X# compiler still recognizes these statements but no longer requires them
RemoveExportLocalClause	This option removes the EXPORT LOCAL clause that may be included in method declarations in VO code, but is ignored in X#
AllClassesArePartial	Partial classes is a mechanism in Vulcan and X# that allows you to split a class over multiple source files. That may be convenient if your classes have become very big or if you want to separate generated code from hand written code (like what the Windows Forms Editor in Visual Studio does). You can choose to make all Xporter classes partial. We have disabled this because there is a (small) performance penalty when you use this. Especially if you have split the Accesses and Assigns from your class over separate files.
AddMissingConstructors	VO allows you to declare a class without Init method / Constructor. In .Net this is also allowed., but then the compiler assumes you have a class with a constructor without parameters. This option will automatically generate missing constructors. Note that also the X# compiler supports emitting missing constructors even if they are not specified in the code, when the /vo16 compatibility option is enable. So this Xporter option is not necessary anymore and has been depreciated.
IgnoreDuplicateDefines	If you select this then duplicate defines will simply be ignored and not written to the output files.
Don'tGenerateEmptyFiles	Each module from your VO application will become a source file in X#. This option will suppress the generation of empty source files
AdjustCallbackFunctions	Some (advanced) VO code contains callback functions where the address of a function is passed to another function and will get called. The Windows API uses that a lot, for example for enumerating windows or printers. That mechanism will not work in .Net. This option will change your code and will help you to get a working solution.
ExportOnlyDefines	This option allows you to generate a DLL with only the defines from the AEF(s) or files you have selected. That may be

useful if you have ported your code before with the Vulcan Transporter and want to get rid of the header files with the defines.

Sort Entities by name	This will sort all entities on name, Methods will be sorted inside the CLASS.. ENDCLASS and functions will be sorted as well
Use XSharp Runtime	When you select this (the default) then your app will be compiled against the X# runtime. Otherwise we will use the Vulcan runtime (which is not included with our product)
Copy Resources to Project folder	When you select this option then all BMP, ICO, CUR etc resources that are used in resource entities in your app will be copied to a Resources subfolder inside your project.
Replace Resource Defines with Values	This controls the way that the numeric identifiers associated with menu options and control identifiers on forms are exported to the external resource files
Check For IVar and Property Conflicts	In Visual Objects, it is allowed to have ACCESS/ASSIGN methods that have the same name with IVars (usually INSTANCE vars) of the same class. This is not allowed in .Net, so if this option is enabled, the Xporter will detect this and will prefix the IVar names with an underscore ("_").
Ignore code inside #ifdef...#endif	As a means of preparing Visual Objects code to be ported to X#, it is common practice to provide a VO version and a X# version of the same code in the VO version of the source code, surrounded by #ifdef __XSHARP__ ... #else ... #endif preprocessor directives. When this option is set, Xporter does not touch any code that is surrounded by those directives.

The XPorter will allow you to select a destination folder and names for your Visual Studio solution and project files.
After pressing xPort you will see the following folder s
C:\XporterOutput\Examples\Explorer
C:\XporterOutput\Examples\Explorer\Explorer
C:\XporterOutput\Examples\Explorer\Explorer\Resources
C:\XporterOutput\Examples\Explorer\Explorer\Properties
The first folder is the so called "Solution" folder
The subfolder contains the project and its files.
If you had selected multiple AEF files then each AEF would have its own subfolder under the Solution Folder
The solution folder contains the files:

File	Contains
Explorer.sln	VS Solution file
Explorer.viproj	XIDE project file

The project folder contains the files

File	Contains
Customer Explorer.prg	Module source code
Data Servers.prg	Module source code
Explorer Shell.prg	Module source code
Explorer.viapp	XIDE Application file
Explorer.xsproj	VS Project file
Help About.HelpAboutDialog.rc	Form resource used by VS
Help About.HelpAboutDialog.xsfrm	Form "binary" used by VS
Help About.prg	Module source code
Help About.rc	Form resource used by XIDE
Icons.ICONONE.rc	Icon resource used by VS
Icons.ICONTWO.rc	Icon resource used by VS
Icons.prg	Module source code
Icons.rc	Icon resources used by XIDE
Menus.CustomerExplorerMenu.rc	Menu resource used by VS
Menus.CustomerExplorerMenu.xsmnu	Menu "binary"
Menus.CustomerExplorerMenu_Accelerator.rc	Menu accelerators resource used by VS
Menus.EMPTYSHELLMENU.rc	Menu resource used by VS
Menus.EMPTYSHELLMENU.xsmnu	Menu binary
Menus.EMPTYSHELLMENU_Accelerator.rc	Menu accelerator resource used by VS
Menus.OrdersListViewMenu.rc	Menu resource used by VS
Menus.prg	Module source code
Menus.rc	Menu resources for XIDE
Start.prg	Module source code

To compile and build the project we open the file Explorer.SLN in Visual Studio. Inside Visual Studio it looks like:
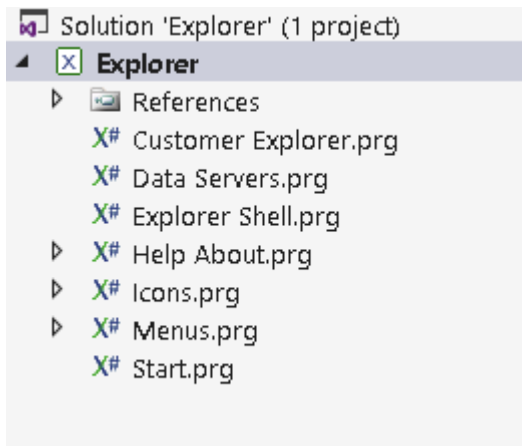
The arrows in front of several modules show that there are subitems in these modules. Those subitems contain the Form, Menu and Icon resources.

Now let's try to build the solution inside Visual Studio. (Menu option Build – Build Solution) This compiles the application and builds a debuggable version in the c:\XporterOutput\Explorer\Debug folder. You will see that this folder contains an EXE, PDB (debug symbols) and copies of the VO runtime libraries:

Explorer.exe

Explorer.pdb

SDK_Defines.dll

VOGUIClasses.dll

VORDDClasses.dll

VOSystemClasses.dll

VOWin32APILibrary.dll

XSharp.VO.dll

XSharp.RT.dll

XSharp.Core.dll

Now can try to run the app. This works, but as soon as we select the File/Open menu option we will get a runtime error. After some resizing the screen looks similar to this
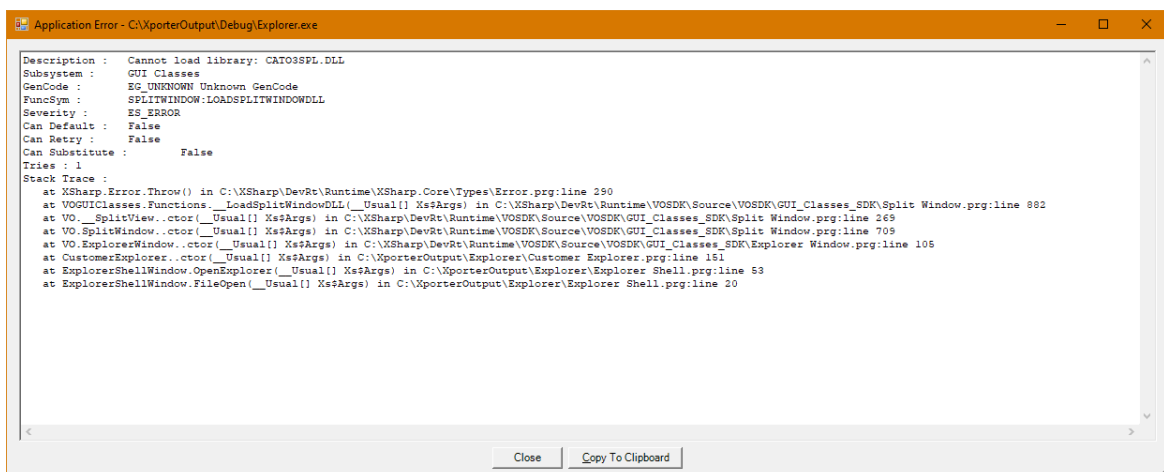
The error message is clear: the app uses the splitwindow control and this requires the splitwindow support DLL.

This DLL is located in the Redist folder of your VO installation. Simply copy the "Cato3spl.dll" file from your VO Redist folder to the output folder as well as the MSVCRT.DLL and try again.

You can also add the DLLs to your project (Add Existing Item, and point to the DLLs in the

Redist folder). This will copy them to the project folder. Then set the build action for the 2 DLLs to None and the "Copy to Output Directory" property to "Preserve Newest". When you now build your app the DLL will be copied to the right folder.

After that the code will run just like it did in VO with no code changes!

Let's have a closer look at what the xPorter has done:

- Each module in VO has become a PRG file in the X# project
- Source code has been grouped by class and put in a CLASS .. END CLASS construct
- Init methods have been renamed to CONSTRUCTOR
- Compiler macros such as "%CavoSamplesRootDir%\Explorer\" have been replaced with literal values: "C:\cavo28SP3\Samples\Explorer\"
- Recourses have become "children" of the source file where they were defined
- Locations for resources have also been changed %CavoSamplesRootDir% \Explorer\TWO.ICO has been changed to ICONTWO Icon C: \cavo28SP3\Samples\Explorer\TWO.ICO.
  Also the RESOURCE keyword has been removed.
- The binaries for the forms and menus have not been included in this version of the XPorter (Beta 11). That will change.
- The Start file has the biggest change. In VO there was a method in there App:Start(). However the App class is not defined in this application but part of the GUI Classes. There was also no Start function in the App. VO would call the Start function in the GUI classes and would merge the App:Start() method in the application with the rest of the App class in the GUI Classes.
  This is not allowed in .Net. Classes cannot be spread over more than one assembly (the .Net name for of a Library, EXE or DLL).
  Therefore the XPorter has created a new class (xApp) and moved the Start method to this Xapp class. It has also added a Start() function which instantiates the App class and calls the Start method inside XApp. You can also see in the example below that a TRY .. CATCH .. END TRY construct is added which captures the errors that are not catched anywhere else and shows an error message in an ErrorBox object. This is the errorbox that was shown before because the CATO3SPL.DLL was missing.
- Finally the Start function is marked with an attribute (The word between the square brackets). In this case the attribute is [STAThread] which means that the main function (and therefore the whole app) needs to run in **S**ingle **T**hreaded **A**ppartment mode. This is important if your app uses external code such as OCX/ActiveX controls.

```
[STAThread];
FUNCTION Start() AS INT
    LOCAL oXApp AS XApp
    TRY
        oXApp := XApp{}
        oXApp:Start()
    CATCH oException AS Exception
        ErrorDialog(oException)
    END TRY
RETURN 0
```

- If you look closer to the solution explorer in Visual Studio and click on the References node in the tree you will see that the example has references to the Vulcan runtime and the VO Class libraries as well as the SDK defines library that was included with the XPorter.
- If you open the project properties you will see on the General page that the selected dialect is "Visual Object" , on the Dialect page are most VO options are set and on the

Build page the platform target is x86 (because the GUI classes are designed to run in x86 mode) and several warnings have been suppressed. These warnings are:

Number	Meaning
162	Suppresses a warning about Unreachable code
618	Suppresses a warning about the use of obsolete functions, such as CollectForced
649	Suppresses a warning about unused private and/or internal fields.
9025	Suppresses a warning about missing return statements
9032	Suppresses a warning about return values inside ASSIGN methods and/or constructors/destructors
9043	Suppresses a warning about ambiguous code, which could happen if two assemblies define a function with the same name.

In this particular example only warning 618 is generated by the compiler because the CustomerExplorer:Close() method calls CollectForced. After commenting out that line you can remove all the "suppressed" warnings from the app and it will compile without any warnings.

You will find the "code before" and "code after" in the XSharp Examples folder

1.3.2 Example 2: The VOPAD Example

This example converts the VOPAD AEF file from the C:
\cavo28SP3\Samples\Controls\Richedit folder.
After converting it with the same options as the 1st example we will have a folder structure like
C:\XporterOutput\Examples\VoPad
C:\XporterOutput\Examples\VoPad\VoPad

File	Contains
VoPad.sln	VS Solution file
VoPad.viproj	XIDE project file

The project folder contains the files

File	Contains
!Readme!.prg	Module source code
image.IMAGE.rc	Image resource for VS
image.prg	Module source code
image.rc	Image resource for XIDE
Manifest.CREATEPROCESS_MANIFEST_RESOURCE_ID.rc	Manifest resource for VS
Manifest.prg	Module source code
Manifest.rc	Manifest resource for XIDE
PadHelp About.HelpAbout.rc	Help about resource for VS
PadHelp About.HELPABOUT.xsfrm	Help about form Binary for VS
PadHelp About.POWVOBMP.rc	Splash resource for VS
PadHelp About.prg	Module source code
PadHelp About.rc	Help resources for XIDE
PadMenus.StandardPadMenu.rc	Menu resource for VS
PadMenus.StandardPadMenu.xsmnu	Menu binary for VS
PadMenus.StandardPadMenu_Accelerator.rc	Accelerator resource for VS

PadMenus.prg	Module source code
PadMenus.rc	Resources for XIDE
PadShell.IDI_VOPADICON.rc	App Icon resource for VS
PadShell.prg	Module source code
PadShell.rc	App Icon resource for XIDE
PadStart.prg	Module source code
PadWin.oMarginDialog.rc	Form resource for VS
PadWin.oMarginDialog.xsfrm	Form binary for VS
PadWin.prg	Module source code
PadWin.rc	Form resource for XIDE
Vopad.viapp	XIDE Application
Vopad.xsproj	VS Project

Open the project in VS and compile. You will see the following errors / warnings

```
PadWin.prg(473,1): warning XS1030: #warning: 'The following method did not include
a CLASS declaration'
PadWin.prg(480,1): warning XS1030: #warning: 'The following method did not include
a CLASS declaration'
PadHelp About.prg(78,3): error XS9046: Cannot assign to 'font' because it is a
'method'
PadHelp About.prg(81,8): error XS0119: 'TextControl.Font(params __Usual[])' is a
method, which is not valid in the given context
PadHelp About.prg(88,3): error XS9046: Cannot assign to 'font' because it is a
'method'
PadHelp About.prg(91,8): error XS0119: 'TextControl.Font(params __Usual[])' is a
method, which is not valid in the given context
PadWin.prg(166,19): error XS1061: 'RichEdit' does not contain a definition for
'RTFChangeFont' and no extension method 'RTFChangeFont' accepting a first argument
of type 'RichEdit' could be found (are you missing a using directive or an assembly
reference?)
```

Let's examine this by double clickig on the errors / warnings:
First the warnings. These are generated by a #warning preprocessor statement inserted by XPorter because it found problems in your code:
- There was an ACCESS FilterIndex CLASS SaveAsDialog in the source. This adds an access to a class in the GUI Classes. This is not allowed in .Net.
  The XPorter has created a special subclass of SaveAsDialog to hold this extra property. Fortunately this ACCESS is no longer needed because it has been added to the GUI Classes already.
  So we can completely de   lete that code.
- There was a method RTFChangeFont CLASS RichEdit. The same problem as the SaveAsDialog:FilterIndex. In this case the method is necessary. It has been moved to a subclass of RichEdit already. However since the original code still points to the RichEdit class there could be a runtime problem because this method is not part of the richedit

class.
There are 2 possible solutions here:
1. Change the method to an extension method
2. Change the code that uses the RichEdit class to use the changed class

Option 1 is the best here since we are not using any private or protected properties of the RichEdit class here. To achieve that change the code from

```
CLASS RichEdit_external_class INHERIT RichEdit
METHOD RTFChangeFont()
    etc
```

to

```
STATIC CLASS RichEditExtensions
STATIC METHOD RTFChangeFont( SELF oEdit as RichEdit) AS VOID
    LOCAL oFontDlg AS StandardFontDialog
    oFontDlg := StandardFontDialog{oEdit:Owner}
    oFontDlg:FontColor := oEdit:TextColor
    oFontDlg:Font       := oEdit:ControlFont
    oFontDlg:Show()
    oEdit:TextColor := oFontDlg:FontColor
    oEdit:ControlFont       := oFontDlg:Font
END CLASS
```

Note

*The **Extension Method** works great to enhance existing classes, but has one disadvantage: The Vulcan Runtime will not pick up these extension methods for late bound code. That is not a problem in this example but may be a problem elsewhere.*

Now we have added an extension method that can be used like a normal method.
This change also solves the error in PadWin.prg
Then look at the other problems: they all have to do with a Font property in the TextControl class. Unfortunately the VO class libraries have both a Font() method and a Font() Access/Assign for the TextControl class. Having a method and a property with the same name is not allowed in .Net.
When the VO Gui classes were ported to Vulcan the decision has been made to rename the Font property to Controlfont. So we need to make that change. Simply double click on the errors and change

```
:Font
```

to

```
:ControlFont
```

(4 times)
After that the code should compile and run. Clicking the Font button on the toolbar should show the font dialog (from the RTFChangeFont extension method)

Some of the changes that were made to the code (omitting the ones that were also in example 1):

- Manifest.prg is empty but included because it had a dependent resource.

You will find the "code before" and "code after" in the XSharp Examples folder

1.3.3   Example 3: The Pizza Example

Our 3rd migration example is the Pizza Example. You can find this in the Samples\Controls\Buttons folder in your VO Installation.
I have selected the same output folder (C:\XporterOutput\Examples\) and the XPorter will create a Pizza subfolder with the contents of the AEF and a Pizza solution in the Examples folder.
There is no real need to list the files again. You get the idea.
After opening Pizza.sln inside visual Studio and compiling you will find the following problems:
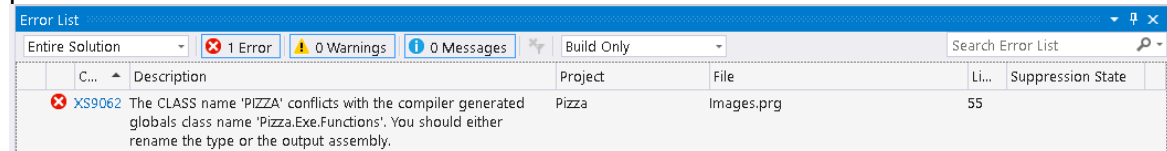
The problem is:
- The original VO program had an output name 'PIZZA.EXE'.
- The X# compiler uses the same logic as the Vulcan compiler and creates a static class called Pizza.Exe.Functions that will hold the functions in the application, the globals and (new for X#) the defines.
- The application also has an image called 'PIZZA'.
- When we compile this app there is now a conflict between the class named PIZZA and the namespace Pizza (from the class Pizza.Exe.Functions).
- This is something that is not recommended by the DotNet standard. Vulcan ignores this and allows both a class Pizza and a namespace Pizza. X# follows the .Net guidelines more closely.
- There are 2 solutions to this problem:
  - Rename the image class Pizza (inside Images.prg)
  - Rename the output filename. To rename the output filename open the Project Settings (from the Project menu) and set the Assembly Name on the "General" page. You could rename the assembly to **PizzaApp** and it will solve the problem
- In this case I choose to rename the PIZZA image. So open Images.PRG and rename

```
CLASS PIZZA INHERIT Bitmap
```

to

```
CLASS ImgPIZZA INHERIT Bitmap
```

- Then recompile again. Now one problem is left:
```
C:\XporterOutput\Examples\Pizza\Order Dialog.prg(165,25): error XS0246: The type
or namespace name 'PIZZA' could not be found (are you missing a using directive
or an assembly reference?)
```

- This is the place where the Pizza image is used in the example app. To change this, open the Order Dialog window (Order Dialog.ORDERDIALOG.xsfrm). Select the button on the bottom left and change the "Image" text in the property window from Pizza to ImgPizza. Save the form. The code will be regenerated as well.
- If you look at the order dialog rc file after the changes you will see that the file has been marked with a header that this code was generated. Also #defines have been added for

all windows styles used in the form. The resource is "self contained" so there is no dependency on any external file in the resource file.

- Recompile and the program compiles.
- When we run the program we see that it works as expected:
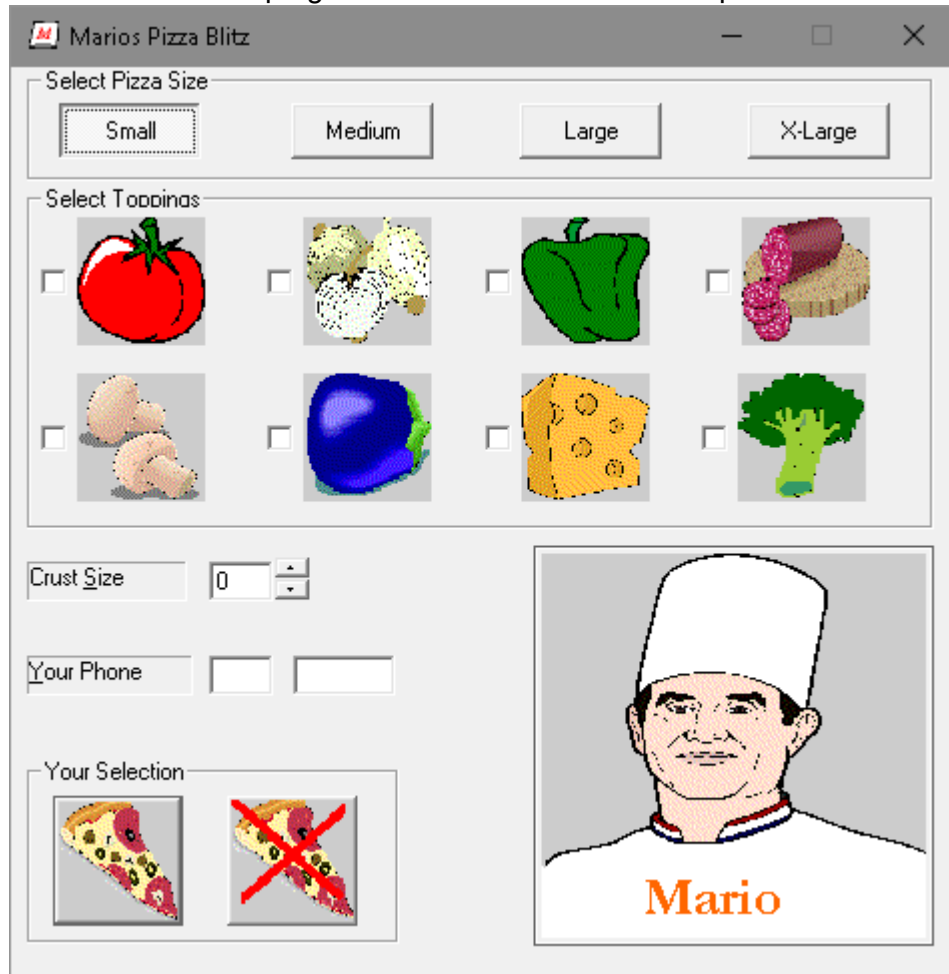
You will find the "code before" and "code after" in the XSharp Examples folder

1.3.4 Example 4: Ole Automation - Excel

For the 4th example we will not use a standard example from Visual Objects but will create a new sample app in Visual Objects to remote control Excel and to write data to an Excel Sheet.

I found the following example in the Comp.Lang.Clipper.Visual-Objects newsgroup (which I changed a little bit for this example).

- Create a new terminal application in Visual Objects and copy and paste the code below into the app.
- Open the application Properties and add the Ole Library.
- Also rename the app to "ExcelTest".
- Now open the start module and copy the code.
- Compile and run, and you will find an xls file in the C:\ExcelTest folder.

```
FUNCTION Start()
LOCAL oExcel AS OBJECT
LOCAL oWorkBooks AS OBJECT
LOCAL oWorksheet AS OBJECT
LOCAL oRange AS OBJECT
LOCAL cFile AS STRING
cFile := "C:\ExcelTest\example.xls"
DirMake("C:\ExcelTest")
oExcel:=OLEAutoObject{"Excel.Application"}
oExcel:Visible:=FALSE // Don't show the EXCEL execute
oExcel:DisplayAlerts:=FALSE // Don't show messages
oWorkBooks:=oExcel:Workbooks
oWorkBooks:add() //open a new worksheet
oWorkSheet:=oExcel:ActiveSheet // active the first sheet
oRange:=oWorkSheet:[Range,"A1","A1"] // A1 cell
oRange:SELECT()
oRange:FormulaR1C1:="Hello my text"
oExcel:ActiveWorkBook:SaveAs(cFile,56,"","",;
     FALSE,FALSE) //"56" save the file in work book of EXCEL 97-
2003 (Excel 8)

oWorkBooks:Close()
oExcel:Quit()
WAIT

RETURN NIL
```

- Export the AEF to "C:\ExcelTest\ExcelTest.AEF"
- Run the VOExporter and export the code.
- After opening the solution inside Visual Studio you will also get an application with one source file (Start.prg). The source is almost identical with one difference. The line :

```
oRange:=oWorkSheet:[Range,"A1","A1"] // A1 cell
```

has been changed to

```
oRange:=oWorkSheet:Range["A1","A1"] // A1 cell
```

- Note that the name Range is now in front of the Square brackets.
  Range is a so called "Indexed property" of the worksheet. Visual Objects uses a "funny" syntax for this.
  X# uses the same syntax that most other languages do. The property looks like an array (the square brackets) .
- Now compile the app in X#. You will get the following errors:

Code	Description	Project	File	Li...	Suppr
❌ XS9059	Cannot convert Array Index from 'string' to 'int'.	ExcelTest	Start.prg	15	
❌ XS9059	Cannot convert Array Index from 'string' to 'int'.	ExcelTest	Start.prg	15	
❌ XS0126	An object of a type convertible to '__Usual' is required	ExcelTest	Start.prg	1	
⚠ XS0028	'Functions.Start()' has the wrong signature to be an entry point	ExcelTest	Start.prg	1	
❌ XS1558	'Functions' does not have a suitable static Start method	ExcelTest	XSC	1	

(Error List — Entire Solution — ❌ 4 Errors ⚠ 1 Warning ℹ 0 Messages — Build Only — Search Error List)

Let's look at these errors:

- The last errors in the list indicate that the Start method is not correct. This is because the Start function in .Net has to either be a VOID function or a function that returns an INT. In this case no return type has been declared, so X# thinks you want to create a function that returns a USUAL. Change the prototype of the start function to:

```
FUNCTION Start() AS VOID
```

and remove the NIL return value from the RETURN statement.

- Now 2 errors remain. These indicate that X# does not know how to convert the array index from string to int. This is on the line with the range assignment.
  The code above uses 'Late Binding', so the types are not known at compile time. X# does not know if the Range property from the Worksheet object is an indexed property or if it returns an array. It assumes that it returns an array and wants to specify the array index which is numeric (and subtract one because Visual Objects uses 1 based arrays where .Net uses 0 based arrays).
- The best way to fix this is to use strong typing and use the Generated Class wrappers for Excel.
  In Visual Objects you would use the 'Automation Server' tool to generate such a class wrapper. In .Net there is a similar tool. The easiest way to achieve this is to add a reference to Excel to the references of the application:
- Right Click on "References" in the solution Explorer and Choose "Add Reference"
- In the 'Add Reference' dialog, choose the COM tabpage
- Locate the "Microsoft Excel nn.m Object Library". On my machine that is Excel 16.0.
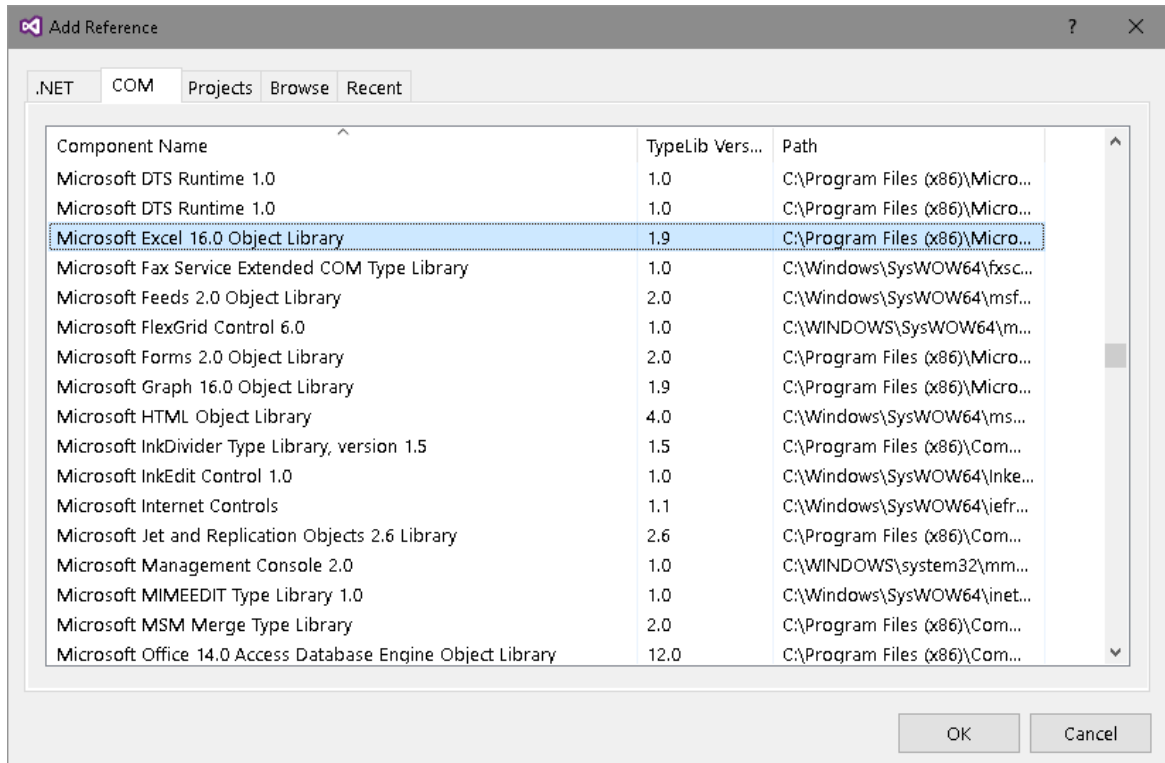
- Click Ok.
- This will add an entry to your References list with the name "Microsoft.Office.Interop.Excel". This is a generated type library that contains class definitions for all types inside Excel.
- Now go into your code, add a Using statement for the namespace of the Excel classes and change "AS OBJECT" to the right types:

```
USING Microsoft.Office.Interop.Excel
FUNCTION Start() AS VOID
LOCAL oExcel AS Application
LOCAL oWorkBooks AS Workbooks
LOCAL oWorksheet AS Worksheet
LOCAL oRange AS Range
```

- Also change the call to create the Main Excel object:
  ```
  oExcel:=ApplicationClass{}
  ```

- Your code is now strongly typed, so you should also get intellisense if you try to select a member from one of these objects, such as oExcel:Workbooks.
- Compile and run the code . It works as expected.
- You may want to change the value **56** in the **SaveAs** line to the appropriate enum value: `xlFileFormat.xlExcel8`
- Now run the app again and everything works as expected.
- If you look in the folder where the EXE is generated you will see both ExcelTest.Exe and Microsoft.Office.Interop.Excel.dll, the type library

Note 1

If you are wondering why we declare oExcel as Application but use ApplicationClass to instantiate the object, here is the reason:
Application is the interface, **ApplicationClass** is the actual class that implements the Application interface. This is a model that you will see for most Automation Servers.

Note 2

Many people have asked for a way to implement OLE Events. With the X# code and the generated type library this is very easy.
Add the following code to the start method to define a BeforeSave and AfterSave event

```
oExcel:WorkbookBeforeSave += OnBeforeSave
oExcel:WorkbookAfterSave += OnAfterSave
```

and add the following functions

```
FUNCTION OnBeforeSave (oWb AS Workbook, SaveAsUI AS LOGIC, Cancel
REF Logic) AS VOID
? "OnBeforeSave", oWb:Path, oWb:Name, SaveAsUI
RETURN

FUNCTION OnAfterSave (oWb AS Workbook, Success AS LOGIC) AS VOID
? "OnAfterSave", oWb:Path, oWb:Name, Success
RETURN
```

And run the example again. You will see that both functions are called. Before the save the path and name are not set properly, afterwards they are set to the values specified in the code
Some versions of Excel do not support the WorkbookAfterSave event. In that case you will get a compile time error

Note: If you run this code through the debugger and set a break point in the OnBeforeSaveAs you will see that the callstack in these events is a little bit weird:
there is no Start() function inside this callstack (I have disabled the 'Just My code option', otherwise you would only see the line with OnBeforeSave())

Call Stack

Name
⏺ ExcelTest.Exe.Functions.OnBeforeSave(Microsoft.Office.Interop.Excel.Workbook oWb, bool SaveAsUI, ref bool Cancel) Line 31
Microsoft.Office.Interop.Excel.AppEvents_SinkHelper.WorkbookBeforeSave() + 0x1e bytes
[Native to Managed Transition]
[Native to Managed Transition]
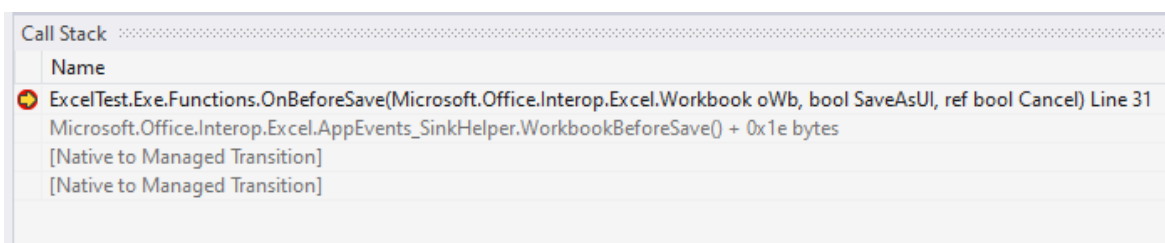
That is because these events are called on a separate thread. If you look at the Threads window in VS (Debug - Windows - Threads) you will see that:
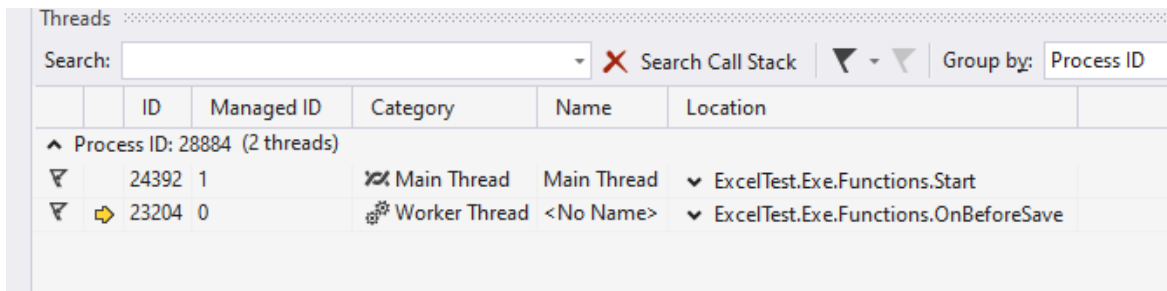
Note 3

If you look closely in the Add References dialog you may also find other occurrences of the Excel library (on the .Net tab). On my machine these are:

These are so called "Primary Interop Assemblies" (PIAs), pre-compiled assemblies, for different Excel versions. You can use these as well. These assemblies are installed with the Office Developers Tools for Visual Studio. On my machine they are located in subfolders of "c:\Program Files (x86)\Microsoft Visual Studio 14.0\Visual Studio Tools for Office\PIA" .

You will find the "code before" and "code after" in the XSharp Examples folder

1.3.5 Example 5: OCX - The Email Client Example

This example shows how to migrate an application that uses an ActiveX/OCX Control. We are using the Email example from Visual Objects that you can find in the Examples folder, subfolder Email.

The problem that we can expect here is that the X# Runtime (and also Vulcan rumtime) does not support ActiveX controls.

So lets try to solve this.

- First run VOXporter and create a Visual Studio solution from the AEF
- Compile and run in Visual Studio.
- We will get 2 messages:
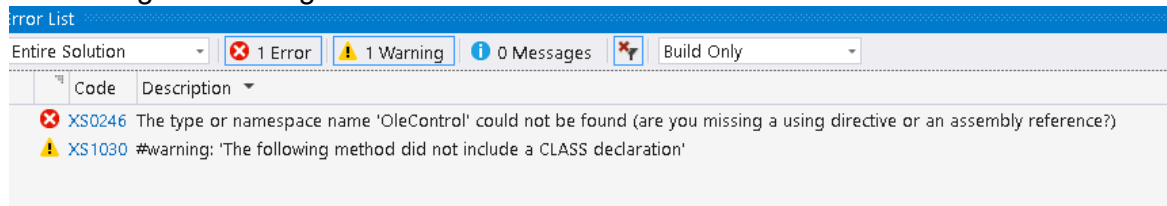
The first message shows the biggest problem in this example. The second message was inserted by the Xporter to warn us that the original code was adding a method to a class that exists inside the VO Gui classes.

Lets fix these problems quickly to be able to compile the app. We will add the OCX Later:

- Click on the warning. You will see that the XPorter has added a `CLASS` `ToolBar_external_class` that inherits from Toolbar. The original code was trying to add the ShowButtonmenu method to the existing Toolbar class.
- We can solve this problem, that we have also seen in the VOPAD example by either adding an Extension Method or by subclassing the Toolbar class.
- Just like in the VOPAD example I prefer the extension method.
- Change the class name and method declaration. We will create 2 overloads, because the symTb parameter is optional:

```
STATIC CLASS ToolBarExtensions
    STATIC METHOD ShowButtonMenu(SELF tbSelf as Toolbar, nButtonID
as LONG, oButtonMenu as Menu) AS VOID
        tbSelf:ShowButtonMenu(nButtonID, oButtonMenu, #MAINTOOLBAR)
        RETURN
STATIC METHOD ShowButtonMenu(SELF tbSelf as Toolbar, nButtonID as
LONG, oButtonMenu as Menu,symTb as Symbol) AS VOID
```

- Remove the Default() line and replace `SELF` in the body of the original ShowButtonMenu with `tbSelf`
- We will also have to make some changes to the code that calls this method. This is due to the fact that the code calls ShowButtonMenu on the Toolbar access from the window class. This Toolbar access is untyped and therefore returns a USUAL..
  So locate the 2 lines with `SELF:ToolBar:ShowButtonMenu` and change that to `((Toolbar) SELF:ToolBar):ShowButtonMenu.` You cannot use the oToolbar field of the Window class, because the DataWindow class will return the Toolbar from its framewindow in stead of its own toolbar.
  In the improved VO SDK that we will include with our X# runtime we will solve this problem by strongly typing properties such as Window:Toolbar.
  Maybe you would be tempted to add the Extension methods to the USUAL type, so you

would not have to add the casts to the code that calls ShowButtonMenu.
That will compile, but unfortunately will produce a problem at runtime. The X# compiler (like the Vulcan and VO Compilers) knows that the USUAL type is special and will not try to emit a method call, but will produce code that calls Send() on the usual to call its method. And the Vulcan Runtime does not handle extension methods inside the Send() function.

- We can confirm that this works later when we press the "Reply" button on the toolbar. That should bring up the menu with "Reply to Sender" and "Reply to All"

Now it is time to fix the ActiveX/OCX problem

- Click on the error about OleControl.

- As a quick workaround we will change the code and let webbrowser inherit from MultiLineEdit. That gives us a control that will certainly work. We will implement the OCX later. To do so go to the Class Webbrowser.PRG and change the INHERIT clause. It says **INHERIT OleControl** now. Change that to **INHERIT MultiLineEdit**.

- Compile again and now some other errors will be shown. 2 of these mention the type cOleMethod. Goto this code by double clicking the error.

- You will see the Quit method of the Webbrowser class. This code uses an internal class and internal methods from the VO OLE Classes. Comment out the contents of this method for now.

- Compile again and you will see that only a few errors are left. Some of these are the same as the error in the VOPAD Example and require that we change the Font property to ControlFont. Correct that.

- One error points to an incorrect '+' operator: in the line

```
cTemp +=  +"; "+ cEntry
```

- This is an obvious error in the original VO code that was never picked up by the VO Compiler. Remove the + before the double quote

- The last error comes from the constructor of the Webbrowser class. It is calling the CreateEmbedding method from the OleControl. This method does not exist in the MultiLineEdit class, so we comment it out for now. We will deal with the Webbrowser later.

- The rest of the code should compile without problems after commenting out the call to SELF:CreateEmbedding().

- You should be able to run the app now.

- There will be a runtime error if you try to open the Address Book because it uses the Databrowser control which depends on Cato3Cnt.dll. Fix this by copying the cato3*.dll and msvcrt.dll from the Cavo28\Redist folder to your output folder.

- Recompile and run the example. I twill now produce an error inside the Display method of the Webbrowser class (DisplayHtml if you have used the Email example from VO 2.8 SP4).
  This method takes the content of the email, writes it to disk and calls the Navigate method of the Webbrowser control (late bound, using the Send() function of VO). This will not work.
  Since we have changed the webbrowser control and made it a multi line edit we can change this behavior. Instead of writing the email text to disk we can simply assign it to the TextValue property of the MultiLineEdit. So comment out the body of the Display method (do not throw the code away we will need it later) and replace it with:

```
SELF:TextValue := cText
```

- After that the sample should run without problems. You can also display the emails. Of course it will not show HTML properly but that is the next step.

How to add the ActiveX to the code

The VO compatible GUI classes inside Vulcan do not support ActiveX Controls. Windows Forms however has great support for ActiveX Controls.
We will use the ActiveX support from Windows Forms to add the ActiveX control to the example.
There are 2 possibilities here:
1. Replace the whole Email Display window with a Windows Forms window
2. Use a trick to use Windows Forms to show the ActiveX control and merge that control into our VO GUI app

The first solution is by far the easiest to understand, but we will have to create a whole new window and we will have to change the calling code as well.
We leave it up to you to make the choice for your own apps.
In this example we will the choose the second approach.

Create a Windows Forms Window to display the email

For this method we use a Windows.Forms.Form window as "Host" for the ActiveX control. We will instantiate that window and will grab the windows handle to the control and link that windows handle with our VO GUI window.
To do this you must take the following steps:
- Right Click on the project icon in the solution explorer and select "Add New Item"
- This brings up a list of possible new items. Choose the icon for Windows Forms Form, give it a meaningful name, such as "EmailDisplayForm.prg" and click Add.
- This will open the Form Designer window.
- Open the ToolBox. The webbrowser control will not be in there.
- Right click on an empty area in the toolbox and select "Choose Items...". This will bring up a dialog where you can control the contents of the ToolBox.
- Select the "COM Components" Tab and scroll down until you see the Microsoft Web Browser control:

- Tick the checkbox in front of the control and press Ok. This should add the ActiveX to the toolbox:
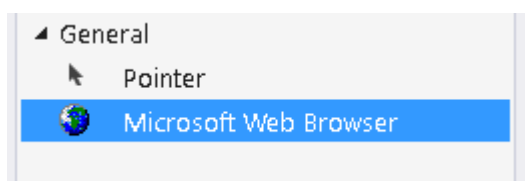
- You can drag the control to a different place in the Toolbox if you are not happy with where it landed.

- Now drag the Control from the ToolBox to the form. There is no need to size or move the control.
- Visual Studio will add two references to our project. These are:
  - **AxSHDocVw**, a type library that contains that code for the actual ActiveX control
  - **SHDocVw**, a type library that contains code for the supporting automation server interfaces and classes
- The form editor will add a field named **axWebBrowser1** to the form. This field is of the type AxSHDocVw.AxWebBrowser.
- Goto the property window and change the **Modifiers** Field to change it from **Private** to it **Public** (Export).
  That will make the field accessible outside of the webBrowserHost class
- Save the code, and close the form
- Now goto the Webbrowser class
- Add the following using clauses to the top of the file:

```
using Email
using AxShDocVw
using ShDocVw
```

The first using is the namespace where the webBrowserHost window is generated. The second namespace is the namespace of the generated ActiveX and the third namespace if that of the other types that we need, such as enums and events.

- Add the following 2 fields (no need to elaborate I think):

```
EXPORT oHost as webBrowserHost
EXPORT oWebBrowser as AxWebBrowser
```

- Goto the constructor of the Webbrowser class and add the following lines of code (in stead of the CreateEmbedding() that we commented out before)

```
SELF:oHost := webBrowserHost{}              // Create the host
window, do not show !
SELF:oWebBrowser := SELF:oHost:axWebBrowser1  // Get the ActiveX
on the form
SetParent(oWebBrowser:Handle, self:Handle())  // Using Windows
API "steal" its handle and link to the MLE
SELF:oWebBrowser:Visible := TRUE              // make the webbrowser
visible
SELF:Okay := TRUE
```

- And add the following methods to make sure that the ActiveX has the same with and height as the MultiLineEdit that is its owner and to make sure it is properly destroyed,

```
METHOD Resize(oEvent)
 LOCAL oDim as Dimension
```

```
    SUPER:Resize(oEvent)
    oDim := SELF:Size
    IF oDim:Width > 0
        SELF:oWebBrowser:SuspendLayout()
        SELF:oWebBrowser:Location := System.Drawing.Point{0,0}
        SELF:oWebBrowser:Size :=
System.Drawing.Size{oDim:Width,oDim:Height}
        SELF:oWebBrowser:ResumeLayout()
    ENDIF
    RETURN NIL


  METHOD Destroy()
    SUPER:Destroy()
    SELF:oWebBrowser:Dispose()
    SELF:oHost:Dispose()
    RETURN NIL
```

- And we need to "restore" the old behavior to display the HTML in the browser window, that we commented out before.
  So goto the WebBrowser:Display() method (DisplayHtml for VO 2.8 SP4) and restore the old code and change

```
  Send(SELF, #Navigate, cFileName)
```

into

```
  SELF:oWebBrowser:Navigate(cFileName)
```

so you change this into an early bound method call
- To finish our work, browse through the source of the webbrowser class and find lines that call Navigate, such as

```
  Send( SELF, #Navigate, "#top" )
```

and change these to early bound method calls:

```
  SELF:oWebBrowser:Navigate("#top" )
```

And look for lines like:

```
  Send( SELF, #ExecWB, OLECMDID_PRINT, OLECMDEXECOPT_DODEFAULT, NIL,
  NIL )
```

and change these to early bound method calls using enums in the type library. Also remove the NIL values:

```
SELF:oWebBrowser:ExecWB(OLECMDID.OLECMDID_PRINT,
OLECMDEXECOPT.OLECMDEXECOPT_DODEFAULT )
```

- That wraps it up. Everything works now, including the PrintPreview and Print functionality.
- Of course you can now also use the activeX events and respond to them.
  You have to do that the .Net way. Something like this:

```
SELF:oWebBrowser:NavigateComplete2 += NavigateComplete2
```

and then the implementation

```
METHOD NavigateComplete2(sender AS OBJECT, e AS
DWebBrowserEvents2_NavigateComplete2Event) AS VOID
        SELF:Owner:StatusBar:SetText("Showing file:"
+e:uRL:ToString())
```

You will find the "code before" and "code after" in the XSharp Examples folder

1.4   The X# Runtime

In X# version 2 - Bandol we have introduced the X# runtime.

In this chapter we would like to give you an overview of the design decisions that we made, what the runtime looks like, where you can find which types and functions etc. We will also list here the features that are not supported yet.

Introduction

When we designed the X# compile and X# Runtime we had a few focus points in mind:

- The language and runtime should be VO compatible whenever possible. We know that the Vulcan devteam made some decisions not to support certain features from VO, but we decided that we would like to be as compatible as technically possible.
- We want our runtime to be fully Unicode and AnyCPU. It should run on any platform and also both in x86 and x64 mode. That has caused some challenges because VO is Ansi (and not Unicode) and also X86. In VO you can cast a LONG to a PTR. That will not work in X64 mode because a LONG is 32 bits and a PTR 64 bits
- We want the code to compile in "Safe" mode. No unsafe code when not strictly needed. The biggest problem / challenge here is the PTR type. With a PTR you can access memory directly and read/write from memory, even if you don't "own" that memory. However the same PTR type is also used as "unique identifier" for example in the low level file i/o and in the GUI classes for Window and Control handles. These PTR values are never used to read/write memory but are like object references. We have decided to use the .Net IntPtr type for this kind of handles. Of course the compiler can transparently convert between PTR and IntPtr.
- We want to prove that the X# language is a first class .Net development language. That is why we decided to write the X# runtime in X#. By doing that we also create a large codebase to test the compiler. So that is a win - win situation.
- We want the runtime to be thread safe. Each thread has its own "global" state and its own list of open workareas. When a new thread is started it will inherit the state of the main thread but will not inherit the workareas from the main thread
- At this moment the X# Runtime is compiled against .Net Framework 4.6.

Assemblies in the X# Runtime

If you want to know in which Assembly a function or type is defined then your "best friend" is the documentation. We are using a tool to generate the documentation, so this is always correct.

Some subsystems have functions XSharp.Core DLL and in XSharp.VO.DLL as well.

Component	Description	Dialect used	Framework Version
XSharp.Core.DLL	This is the base DLL of the X# Runtime.	X# Core	4.6

Component	Description	Dialect used	Framework Version
XSharp.Data.DLL	This DLL contains support code for .Net SQL based data access and for SQL based cursors	X# Core	4.6
XSharp.RT.DLL	This DLL is required for all dialects apart from Core	X# non - core	4.6
XSharp.RT.Debugger.DLL	This DLL contains the functions and windows for the Runtime Debugger	X# core	4.6
XSharp.VO.DLL	This DLL adds features to the runtime that are needed for the VO and Vulcan dialects.	X# VO and X# Vulcan	4.6
XSharp.XPP.DLL	This DLL adds features to the runtime that are needed for the Xbase++ dialect.	X# XPP	4.6
XSharp.VFP.DLL	This DLL adds features to the runtime that are needed for the FoxPro dialect.	X# FoxPro	4.6
XSharp.Macrocompiler.DLL	This DLL is the X# "fast" macro compiler.	X# Core	4.6
XSharp.MacroCompiler.Full.DLL	This DLL is the X# "full" macro compiler.	X# Core	4.6
XSharp.RDD.DLL	This DLL contains the various RDDs implemented for X#.	X# Core	4.6
VO SDK Class libs: VOConsoleClasses.dll VOGUIClasses.dll VOInternetClasses.dll VORDDClasses.dll VOReportClasses.dll VOSQLClasses.dll VOSystemClasses.dll VOWin32APILibrary.dll	These DLLs represent the class libraries from Visual Objects	X# VO and X# Vulcan	4.6

Missing or incomplete Features

Feature	Description	Expected when
Some runtime functions are not supported yet: Crypt functions (Crypt(), CryptA()) Encoding functions (B64Enc.., UUEnc.., GetChunkBase64 etc)	These functions will most likely be added in one of the next betas. For now they will throw a notimplementedexception when you use them	

1.4.1 XSharp.Core

This is the base DLL of the X# Runtime. It contains:
* Common interfaces, such as IDate, IFloat, ICodeBlock, IMacroCompiler, used by other X# components
* Runtime functions that use standard .Net datatypes such as SLen(), AllTrim(), SLen().
* The Runtime State (SetDeleted() , SetExact() etc)
* The Workarea information
* The nation resources and collations.
* RDD related functions that do not depend on XBase types as DATE, USUAL and FLOAT.
* The low level File/IO

This assembly can be used in all dialects of X#

1.4.2 XSharp.Data

This DLL contains support code for .Net SQL based data access and for SQL based cursors (VFP)
The DLL also has data aware classes and functions that return DataTables or
DataSources based on workareas/cursors

1.4.3   XSharp.RT

This DLL adds the following features to the runtime:
- XBase specific types , such as DATE, ARRAY, USUAL, PSZ, SYMBOL etc
- CodeBlock type (base class for compile time codeblocks)
- _CodeBlock type (base class for macro compiled codeblocks)
- Runtime functions that accept parameters of these types.
- Memory related functions (MemAlloc, MemFree etc)
- Terminal IO functions (QOut(), Accept, Wait).
- RDD Related functions that have optional parameters or parameters of XBase Type
- Functions to compile and evaluate macros (MComp, MExec)
- Late binding functions such as Send(), IVarGet() etc
- VObject type
- OLEAutoObject type

This assembly should be linked in when you compile for anything other than the Core dialect

1.4.4 **XSharp.RT.Debugger.DLL**

This DLL adds the the runtime Debugger functions and windows:
- DbgShowGlobals()
- DbgShowWorkareas()
- DbgShowMemvars()
- DbgShowSettings()

1.4.5 XSharp.VO

This DLL adds some specific types and functions that are unique to the VO and Vulcan
dialect. Functions such as
QueryRtRegInt()
QueryRtRegString()
MB..()

Also many obsolete functions from VO are in this DLL.

This DLL also contains the classes
ErrorDialog
NameArg
OleAutoObject
OleAutoObjectFromFile
OleDateTime
VObject

1.4.6   **XSharp.XPP**

This DLL adds some specific types and functions that are unique to the Xbase++ dialect
and also many of the Xbase++ defines.
Functions such as
ClassCreate
ClassDestroy()
ClassObject()
XMLDocOpenFile()
XMLDocSetAction()
etc.

Also some Xbase++ specific types such as
Abstract
ClassObject

1.4.7 XSharp.VFP

This DLL adds some specific types and functions that are unique to the FoxPro dialect

Functions such as
CreateObject()
etc.

1.4.8 XSharp.VFP.UI

This DLL contains forms and controls that are used by applications converted from FoxPro to X# with the VFP Exporter.

1.4.9   XSharp.Macrocompiler

This DLL is the X# "fast" macro compiler. The Macro compiler is written in C# and has a hardcoded dependency on XSharp.Core.

Please note that there is no link between XSharp.RT.DLL and XSharp.Macrocompiler.DLL. When you compile a macro then the runtime will try to locate the macro compiler with the method listed below.
You can override this mechanism by calling SetMacroCompiler() with the type of the class that implements the macro compiler. This type should implement the XSharp.IMacroCompiler interface. If you want to use the standard macro compiler then you need to add a reference to XSharp.MacroCompiler.dll and add the following code to the startup code of your application:

SetMacroCompiler(typeof(XSharp.Runtime.MacroCompiler))

If you don't do this then the runtime will try to locate the macro compiler in the following locations:
• The directory where the XSharp.RT.DLL is loaded from
• The directories in the path. If you use this then make sure that the assemblies listed below are in the same folder as XSharp.MacroCompiler.DLL
• The Global Assembly Cache (GAC). If you use this mechanism then make sure that the assemblies listed below are also in the GAC.

This assembly depends on:
• XSharp.Core.DLL

Note

The XSharp installer adds the Macro compiler and the assemblies it depends on to the GAC so you will be able to run your apps without adding the macro compiler to the references list of your application. Please make sure you include the macro compiler in your installer when deploying your applications.

1.4.10 XSharp.Macrocompiler.Full.DLL

This DLL is the X# "full" macro compiler. The Macro compiler is created as a class wrapper on top of the X# scripting engine

We are working on a smaller and faster macro compiler. More news about that will follow.

Please note that there is no link between XSharp.VO.DLL and XSharp.Macrocompiler.DLL. When you compile a macro then the runtime will try to locate the macro compiler with the method listed below.
You can override this mechanism by calling SetMacroCompiler() with the type of the class that implements the macro compiler. This type should implement the XSharp.IMacroCompiler interface. If you want to use the full macro compiler in stead of the standard (fast) macro compiler then you need to add a reference to XSharp.MacroCompiler.Full.dll and add the following code to the startup code of your application:

SetMacroCompiler(typeof(XSharp.MacroCompiler))

If you don't do this then the runtime will try to locate the standard macro compiler in the following locations:
- The directory where the XSharp.R.DLL is loaded from
- The directories in the path. If you use this then make sure that the assemblies listed below are in the same folder as XSharp.MacroCompiler.DLL
- The Global Assembly Cache (GAC). If you use this mechanism then make sure that the assemblies listed below are also in the GAC.

This assembly depends on:
- XSharp.Scripting.DLL
- XSharp.CodeAnalysis.DLL
- System.Collections.Immutable
- System.Reflection.Metadata

Note

The XSharp installer adds the Macro compiler and the assemblies it depends on to the GAC so you will be able to run your apps without adding the macro compiler to the references list of your application. Please make sure you include the macro compiler in your installer when deploying your applications.

1.4.11 XSharp.RDD

This DLL contains the various RDDs that come with X#:
- DBFNTX (including DBT memos)
- DBFCDX (Including FPT memos)
- DBFVFP = DBFCDX with support for additional Visual FoxPro datatypes.

You can also uses "limited" RDDs with do not have all the functionality:
- CAVODBF or DBF: Just DBF Files
- DBFDBT: DBF files with DBT memos, no index support
- DBFFPT: DBF files with FPT memos, no index support

The Advantage RDDs (require external advantage DLLs, such as ACE32, ACE64, adsloc32.dll, adsloc64.dll etc)
- ADSADT
- AXDBFCDX
- AXDBFNTX
- AXDBFVFP

- Advantage.ADSADT
- Advantage.AXSQLCDX
- Advantage.AXSQLNTX
- Advantage.AXSQLVFP

In one of the coming releases we will add support for:
- DBFMEMO = DBF + NTX + DBV
- DBFBLOB = DBF without associated DBF file
- DELIM
- SDF

and may be, if there is enough interest:
DBFNSX = DBF + SMT + NSX

1.4.12   Installation in the GAC

The default behavior of the X# installer is to register the X# runtime in the GAC. The following files are registered there

- XSharp.Core.DLL
- XSharp.Data.DLL
- XSharp.RT.DLL
- XSharp.VO.DLL
- XSharp.XPP.DLL
- XSharp.VFP.DLL
- XSharp.RDD.DLL
- XSharp.Macrocompiler.DLL
- VOConsoleClasses.dll
- VOGUIClasses.dll
- VOInternetClasses.dll
- VORDDClasses.dll
- VOReportClasses.dll
- VOSQLClasses.dll
- VOSystemClasses.dll
- VOWin32APILibrary.dll

Some components are dynamically loaded at runtime and do not have to added as references to your application:
- XSharp.RDD.DLL
- XSharp.Macrocompiler.DLL

The X# runtime tries to locate these assemblies in the GAC and in the directory where the application is installed.
If you are not installing the runtime to the GAC, then you must make sure that these two DLLs are available in your application folder. Otherwise you will get a runtime error when these DLLs are needed.

1.4.13 Who is who in the X# team

The founders of the X# project started the X# project in the summer of 2015.
They have ample experience with the xBase language. They all worked on the Visual Objects and/or Vulcan.NET development teams, and some of them have created 3rd party additions to Visual Objects and/or Vulcan.NET or created tools for VO and Vulcan developers.

They are in alphabetical order:

Name	Country	Email	Role in the X# development team
Fabrice Foray	France	fabrice@xsharp.eu	Visual Studio integration, examples, tutorials
Nikos Kokkalis	Greece	nikos@xsharp.eu	(Macro) Compiler
Chris Pyrgas	Greece	chris@xsharp.eu	Support, tools, examples, tutorials, runtime
Robert van der Hulst	The Netherlands	robert@xsharp.eu	Management, Compiler, Runtime, Visual Studio integration, Documentation

If you are interested in participating in the development of X# please contact us and let us know what your area of expertise is. There are plenty of things to do !

1.4.14 XBase Types

The table below shows the type mapping between XBase types in X#.
All types in the namespace XSharp are implemented in XSharp.VO.DLL and are only supported when NOT compiling in the Core dialect

XBase Type	Implemented in type
ARRAY	XSharp.__Array
ARRAY OF <t>	XSharp.__ArrayBase <T>
BINARY	XSharp.__Binary
BYTE	System.Byte
CHAR	System.Char
CODEBLOCK	XSharp.CodeBlock
_CODEBLOCK	XSharp._CodeBlock
DATE	XSharp.__VODate
CURRENCY	XSharp.__Currency
DATETIME	System.DateTime
DECIMAL	System.Decimal
DWORD	System.UInt32
DYNAMIC	System.Dynamic
FLOAT	XSharp.__VOFloat
INT / LONG / LONGINT	System.Int32
INT64	System.Int64
LOGIC	System.Boolean or XSharp.__WinBool
OBJECT	System.Object
PSZ	XSharp.__Psz
PTR	System.IntPtr
REAL4	System.Float
REAL8	System.Double
SHORT / SHORTINT	System.Int16
STRING	System.String
SYMBOL	XSharp.__Symbol
UINT64	System.UInt64
USUAL	XSharp.__Usual
VOID	System.Void

WORD	System.UInt16

1.4.14.1 Array Of Type

This type is used for typed arrays. It has been introduced in the X# Runtime. The internal typename is XSharp.__ArrayBase
The code below shows how you can use it.
Many of the existing XBase runtime functions that accept arrays now also accept an ARRAY OF.
Runtime functions that expect a codeblock for Arrays expect a Lambda expression for ARRAY OF.
The difference is that the parameters to the Lambda expression will be typed, so there is no late binding necessary.
Parameters to a Codeblock are always of type usual and therefore either require Late Binding support or need to be casted to the right type inside the codeblock.
We have also introduced a special multi dimensional syntax. Given the example below you can also get the first name of the first developer in the array by using the following syntax:

```
cFirst := aDevelopers[1, "FirstName"]
```

This may be useful when you work with existing generated code and the existing code was using a define for the elements in the multi dimensional array.
If you had a (generated) define like

```
DEFINE DEVELOPER_FirstName := 1
```

then you can change the code generator and generate this in stead

```
DEFINE DEVELOPER_FirstName := "FirstName"
```

The code that uses the define can remain unchanged

```
cFirst := aDevelopers[1, DEVELOPER_FirstName]
```

Example code

```
FUNCTION Start AS VOID
// declare typed array of developer objects
LOCAL aDevelopers AS ARRAY OF Developer
// Initialize the array with the "normal" array syntax
aDevelopers   := {}
// AAdd also supports typed arrays
AAdd(aDevelopers, Developer  { "Chris","Pyrgas"})
AAdd(aDevelopers, Developer  { "Nikos", "Kokkalis"})
```

```
AAdd(aDevelopers, Developer  { "Fabrice", "Foray"})
AAdd(aDevelopers, Developer  { "Robert", "van der Hulst"})
// AEval knows that each element is a developer
AEval(aDevelopers, { d => Console.WriteLine(d)})
// ASort knows the type and passes the correct types to the lambda
expression.
// The compiler and runtime "know" that x and y are Developer
objects and will produce early bound code
ASort( aDevelopers, 1, ALen(aDevelopers), { x, y => x:LastName <
y:LastName})
// Foreach knows that each element is a Developer object
FOREACH VAR oDeveloper IN aDevelopers

    ? oDeveloper:LastName, oDeveloper:FirstName
NEXT
RETURN


CLASS Developer
    PROPERTY FirstName AS STRING AUTO
    PROPERTY LastName AS STRING AUTO
        CONSTRUCTOR()
            RETURN
    CONSTRUCTOR (cFirst AS STRING, cLast AS STRING)
        FirstName := cFirst
        LastName  := cLast
    METHOD ToString() AS STRING
        RETURN Firstname+" " + LastName
END CLASS
```

1.4.14.2 Array Type

This implements the so called "ragged" arrays.
The internal typename is XSharp.__Array

The ARRAY type is a dynamic array of USUAL values. Each element of the array may contain another array, so arrays can be multidimensional.

Implementation

The ARRAY type is implemented in the class XSharp.__Array.
The Usualtype of ARRAY has the value 5

1.4.14.3 CodeBlock

This is the parent class for compile time codeblocks.
There is also a subclass _CodeBlock which is the parent class for macro compiled (runtime) codeblocks
The internal type names are XSharp.CodeBlock and XSharp._CodeBlock

The **codeblock** type was introduced in the XBase language in the Clipper days.

They can be seen like unnamed functions. They can have 0 or more parameters and return a value.
The most simple codeblock that returns a string literal looks like this

```
FUNCTION Start() AS VOID
LOCAL cb AS CODEBLOCK
cb := {|| "Hello World"}
? Eval(cb)
WAIT

RETURN
```

To use a codeblock you call the Eval() runtime function
Codeblocks are not restricted to fixed expressions, because they can use parameters.
The following codeblock adds 2 parameters.

```
FUNCTION Start() AS VOID
LOCAL cb AS CODEBLOCK
cb := {|a,b| a + b}
? Eval(cb, 1,2)
? Eval(cb, "Helllo ", "World")
WAIT
RETURN
```

As you can see in the example, we can both use numeric parameters here or string parameters. Both work. That is because the parameters to a codeblock are of the so called USUAL type. They can contain any value. Of course the following will fail because the USUAL type does not support multiplying strings:

```
FUNCTION Start() AS VOID
LOCAL cb AS CODEBLOCK
cb := {|a,b| a * b}
? Eval(cb, 1,2)
? Eval(cb, "Helllo ", "World")
WAIT
RETURN
```

More complicated codeblocks

Codeblocks are not restricted to single expressions.
They may also contain a (comma seperated) **list of expressions**. The value of the last expression is the return value of the codeblock:

```
FUNCTION Start() AS VOID
LOCAL cb AS CODEBLOCK
```

```
cb := {|a,b,c| QOut("value 1", a) , QOut("value 2", b),
QOut("value 3", c), a*b*c}
? Eval(cb,10,20,30)
WAIT
RETURN
```

XSharp has also introduced codeblocks that contain of **(lists of) statements:**

```
FUNCTION Start() AS VOID
LOCAL cb AS CODEBLOCK
cb := {| a,b,c|
    ? "value 1" ,a
    ? "value 2" ,b
    ? "value 3" ,c
    RETURN a*b*c
    }
? Eval(cb,10,20,30)
WAIT
RETURN
```

Please note

- The first statement should start on a new line after the parameter list
- There should be NO semi colon after the parameter list.
- The statement list should end with a RETURN statement.

Implementation

The CODEBLOCK type is implemented in the abstract class XSharp.Codeblock
The Usualtype of CODEBLOCK has the value 9.

In your code you will never have objects of type XSharp.Codeblock.
Compile time codeblocks are translated into a subclass of XSharp.Codeblock
Runtime (macro compiled) codeblocks are translated into a subclass of the class
XSharp._Codeblock which inherits from Codeblock.
Depending on the type of the runtime codeblock this is either an instance of the
MacroCodeblock class or of the MacroMemVarCodeblock class (when the macro creates
dynamic memory variables)

1.4.14.4 Date Type

This structure holds year, month, day in 32 bits. For date calculations it uses the
System.DateTime calculation logic. There are implicit converters between Date and
DateTime.
The internal type name for this type is XSharp.__Date

The DATE type is an integral type that stores a date value.

The DATE is internally stored in 3 fields (DAY, MONTH and YEAR) that occupy a total of 32 bits in memory.

Implementation

The DATE type is implemented in the structure XSharp.__Date
The Usual type of DATE is 2.

1.4.14.5 Binary Type

This implements the FoxPro binary type
The internal typename is XSharp.__Binary

The BINARY type is represented a series of bytes.

- Binary literals are written as 0h12345678abcdef
- The value behind 0h is a sequence of hex numbers. Each pair of hex numbers (nibble) represents 1 byte. There must be an even number of 'nibbles'.
- The binary literals are encoded in an array of bytes. In the Core dialect the binary literals are represented as a byte[]. In the other dialects the binary literals are a new type (XSharp.__Binary) which can be specified as the new BINARY keyword.
- The UsualType() of BINARY is 29.
- The XSharp.__Binary type has operators to add a string to a binary and add a binary to a string.
  Binary + String will return a Binary
  String + Binary will return a String
  Binary + Binary will return a Binary.
  There are also comparison operators on the Binary type (>, <, >=, <=). These will use the string comparison routines that are defined with SetCollation() with the exception that an = comparison with a single equals operator does not return TRUE when the Right hand side is shorter than the Left hand side and the first bytes match.
- Conversions from Binary to String are done with the Encoding.GetBytes() and Encoding.GetString() functions for the current Windows Encoding.
  That means that on single byte code pages each character in the string will result in one byte and each byte will result in one character.
  For multibyte code pages (Chinese, Japanese, Korean etc) some characters will result in more than one byte and some byte pairs will result in a single character.
- There are implicit operators that convert a BINARY to a byte[] and back. There are also implicit operators that convert a Binary to a String and back.
- When compiling with the Vulcan Runtime then the byte[] array is stored in a USUAL value for the non core dialects.

Implementation

The BINARY type is implemented in the class XSharp.__Binary
The Usualtype of BINARY has the value 29

1.4.14.6 Currency Type

This implements the FoxPro Currency type
The internal typename is XSharp.__Currency

- The currency type stored numbers with a precision of 4 decimals. Internally it contains a .Net decimal value, rounded to 4 decimals.

-

Implementation

The CURRENCY type is implemented in the structure XSharp.__Currency
The Usual type of CURRENCY is 28.

1.4.14.7 Float Type

This structure is a combination of a REAL8 (System.Double) and a width and # of decimals.
It is not a reference type like in VO.
The internal type name for this type is XSharp.__Float

The FLOAT type is a type that stores a 64-bit floating point value, along with formatting information. The precision and range of a FLOAT are the same as that from a REAL8, since the value of the float is stored in a REAL8.

Implementation

The FLOAT type is implemented in the structure XSharp.__Float
The Usual type of FLOAT is 3.

1.4.14.8 Logic Ttype

For normal use the logic type is mapped to System.Boolean.
Inside VOSTRUCT and UNION the XSharp.__WinBool is used because this is a 4 byte value just like the Win32 api expects.

The LOGIC keyword represents the .Net Boolean type. This type can have either of two values: true, or false.

If you have members of type LOGIC in VOSTRUCT or UNION types then these will not be represented with .Net Boolean types because the size of these Boolean is 1 byte but in the Windows API LOGIC values are represented with 4 bytes. Therefore the compiler will replace these with a special type __WinBool which has 4 bytes and has implicit converters between Logic and __WinBool.

1.4.14.9 PSZ Type

This structure contains a pointer to a memory block with an Ansi string. When created with String2Psz() and Cast2Psz() then the compiler will take care of releasing these memory blocks. When created with other runtime functions such as StringAlloc() then you are responsible of releasing the memory yourself.
The internal type name for this type is XSharp.__Psz

Note
Please do not use the PSZ type for new code. It is only included for backward compatibility.
Even new code that calls the windows API can use a better alternative, for example using the [DllImportAttribute] from the framework

The PSZ type is a pointer type that points to a null terminated sequence of zero or more bytes, typically representing a printable character string. This type is for backward compatibility only. Don't use this type in new code unless you have to.

Implementation

The PSZ type is implemented in the class XSharp.__Psz
The Usual type of PSZ is 17.

1.4.14.10 Symbol Type

This structure holds just a number. This number is a reference to a string table that contains the actual string representation of the symbol.
The internal type name for this type is XSharp.__Symbol

The SYMBOL type is a 32-bit integer that represents an index into an array of strings.

Since a SYMBOL represents a string, there is a built-in implicit conversion from SYMBOL to STRING, and from STRING to SYMBOL.

Since the underlying value of a SYMBOL is an integer, there is a built-in explicit conversion from SYMBOL to DWORD and from DWORD to SYMBOL. A cast is necessary in order to perform explicit conversions.

Unlike with Visual Objects, the number of symbols is not limited by available memory or symbols that are declared in another library.

Literal symbols consist of the '#' character followed by one or more alphanumeric character. The first character must be a letter or an underscore.

Some examples of literal symbols are shown below:
#XSharp
#XSHARP

Note that although literal symbols can be specified with lower or upper case letters, the strings they represent are converted to uppercase at compile time, for compatibility with Visual Objects. It is not possible to specify a literal symbol that contains lower case letters, the StringToAtom() function must be used.

The compiler-defined constant NULL_SYMBOL can be used to express a null symbol, i.e. a symbol that has no associated string value.

Implementation

The SYMBOL type is implemented in the structure XSharp.__Symbol
The Usual type of SYMBOL is 10.

1.4.14.11 Usual Type

The USUAL type in X# is implemented as a .Net structure. It contains a type flag and a value. The value can be one of the following types:
NIL, Long, Date, Float, Array, Object, String, Logic, Codeblock, Symbol, Ptr, Int64, DateTime, Decimal, DateTime
The internal type name for this type is XSharp.__Usual

Type	Usual Type Number
NIL	0

Type	Usual Type Number
Long	1
Date	2
Float	3
Array	5
Object	6
String	7
Logic	8
Codeblock	9
Symbol	10
Ptr	18
Int64	22
DateTime	26
Decimal	27
Currency	28
Binary	29

Note that some Usual Type numbers are not included in this table. There are defines in the compiler for these numbers, but **they are never stored** inside a USUAL.
So you can write UsualType(uValue) == REAL8 but that will NEVER be true.

You can assign values of these other types to a USUAL but the values will be converted to a type listed in the first table. For example if you assign a DWORD to a USUAL then the runtime will look at the value of the DWORD and if it is smaller or equal to MAX_LONG then it will store the value as a LONG. Otherwise it will store it as a FLOAT. Please note that although we support the Int64 type the DWORD conversion does not use this to be compatible with VO. Also if you assign a PSZ to a USUAL then it will be stored as a STRING. So the runtime will automatically call Psz2String() to get the string representation.

Name	Usual Type Number
Byte	11
Short	12
Word	13
DWord	14
Real4	15
Real8	16
PSZ	17

Name	Usual Type Number
Usual By Ref	19
UInt64	23
Char	24
Dynamic	25

The USUAL type is datatype that can contain any data type. It consists internally of a type flag and a value. This type can store any value.
The compiler treats this type in a special way. The compiler will not warn you when you assign a value of type USUAL to another type, but will automatically generate the necessary conversion operation/

USUAL is provided primarily for compatibility with untyped code. It is not recommended for use in new code because the compiler cannot perform any type checking on expressions where one or more operands are USUAL. Any data type errors will only be discovered at runtime.

Locals, parameters and fields declared as USUAL also incur considerably more runtime overhead than strongly typed variables.

The literal value NIL may be assigned into any storage location typed as USUAL. The value NIL indicates the absence of any other data type or value, and is conceptually equivalent to storing NULL into a reference type. NIL is the default value for a local USUAL variable that has not been initialized.

When the left operand of the ':' operator is a USUAL, the compiler will generate a late bound call to the method, field or property specified as the right operand. This call may fail if the value contained in the USUAL at runtime does not have such a member, the member type is incorrect or inaccessible, or if the name evaluates to a method and the number of parameters or their types is incorrect. The /lb compiler option must be enabled in order to use a USUAL as the left operand of the ':' operator, otherwise a compile-time error will be raised.

Numeric operations and USUAL variables of mixed types.

When you combine 2 USUAL variables in a numeric operation then the type of the result is derived from the types of operands.
The leading principle has been that we try not to loose decimals.

The generic rule is:
- When the Left Hand Side is fractional then the result is also fractional of the type of the LHS
- When the LHS is NOT fractional and the Right Hand Side (RHS) is fractional then the result is the type of the RHS
- When both sides are integral then the result has the type of the largest of the two.

LHS \ R	LONG	INT64	FLOAT	CURRENCY	DECIMAL

	HS				
LONG	LONG	INT64	FLOAT	CURRENCY	DECIMAL
INT64	INT64	INT64	FLOAT	CURRENCY	DECIMAL
FLOAT	FLOAT	FLOAT	FLOAT	FLOAT	FLOAT
CURRENCY	CURRENCY	CURRENCY	CURRENCY	CURRENCY	CURRENCY
DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL

Implementation

The USUAL type is implemented in the structure XSharp.__Usual

1.4.15 Workarea Events

In build 2.08 we have added the option to register an event handler that receives workarea events. This can be used to monitor access to workareas and for example write this information to a log file.

There are 2 ways to install this event handler:
1. You create a function/ method that implements the DbNotifyEventHandler delegate.

```
PUBLIC DELEGATE XSharp.DbNotifyEventHandler(osender AS XSharp.RDD.IRDD, e AS
XSharp.DBNotifyEventArgs) AS VOID
```

2. You create a class that implements the IDbnotify interface, which only has a method named Notify() with the same prototype as the delegate from 1)

Important

- Your event handler should do as little as possible if you don't want to slow down the whole RDD system.
- Do not manipulate any workareas from within your event handler to avoid recursion.
- Unregister your event handler as soon as possible.
- If you use an object for event handling you are responsible yourself to manage the life time of this object. And be sure to unregister the object before it runs out of scope
- In some cases (operations that work on more than one workarea, such as DbCommitAll() , DbUnLockAll()) the event handler will not always receive a sender parameter. Be prepared for that !
- The events are sent from the CoreDb level. So if a method inside an RDD calls another method inside that RDD (for example CreateOrder() might call GoTop() after creating the order) then you will not see that second event, only the events related to the method that was called from the CoreDb() level, so the events related to the Ordercreation in this example.

Examples

If you use approach 1 then you need to add the event handler to CoreDb.Notify like in the following example:

```
FUNCTION NotifyRDDOperations(oRDD AS XSharp.RDD.IRdd, oEvent AS
XSharp.DbNotifyEventArgs) AS VOID
    IF oRDD != NULL
        ? oRDD:Alias, oEvent:Type:ToString(), oEvent:Data
    ELSE
        ? "(no area)",oEvent:Type:ToString(), oEvent:Data
    ENDIF
    RETURN

FUNCTION Start() AS VOID
    CoreDb.Notify += NotifyRDDOperations
```

X#

```
    ? DbCreate("Test", {{"FLD1" , "C" , 10 , 0} })     // This will
trigger a notification
    CoreDb.Notify -= NotifyRDDOperations                // Do not
forget to unregister!
    WAIT
RETURN
```

The parameters to the function are:

oRDD The RDD for which the event is triggered. This may be NULL for events that involve multiple workareas, such as DbUnlockAll()

oEvent An event handler object that has 2 properties

Type   A Value of the DbNotificationType enum

Data   An object that has additional information about the event, such as the fieldname for a PieldPut and the recordnumber for a Append or Delete event..

For the second approach you create a class and register the class with the DbRegisterClient() function and unregister the class with the DbUnRegisterClient function The method in the class will get the same arguments as the event handler function:

```
CLASS Notifier IMPLEMENTS IDbnotify
    METHOD Notify(oRDD AS XSharp.RDD.IRdd, oEvent AS
XSharp.DbNotifyEventArgs) AS VOID
        IF oRDD != NULL
            ? oRDD:Alias, oEvent:Type:ToString(), oEvent:Data
        ELSE
            ? "(no area)",oEvent:Type:ToString(), oEvent:Data
        ENDIF
        RETURN
END CLASS

FUNCTION Start() AS VOID
    LOCAL oNot AS Notifier
    oNot := Notifier{}
    DbRegisterClient(oNot)
    ? DbCreate("Test", {{"FLD1" , "C" , 10 , 0} })     // This will
trigger a notification
    DbUnRegisterClient(oNot)                            // Do not
forget to unregister!

    WAIT
RETURN
```

See Also

DbNotificationType
DbRegisterClient()
DbUnRegisterClient

DbNotifyEventHandler
IDbnotify

1.4.16 Dialect (in)compatibilities

The following pages list incompatibilties between X# and the various dialects. Please note that this list is not complete yet.

1.4.16.1 VO

Compiler

Operation	Difference
Adding methods to existing classes	In X# (actually in .Net) you cannot add methods to classes in other assemblies. As work around you can: • write an extension method • create a subclass

Runtime functions

Function	Difference

1.4.16.2 Vulcan.NET

Compiler

Operation	Difference
Adding methods to existing classes	

Runtime functions

Function	Difference

1.4.16.3 Xbase++

Compiler

Operation	Difference
[] operator on usuals	The [] operator on usuals does not work with numeric usuals in X#.

Runtime functions

Function	Difference
FErase()	returns a LOGIC in X# and a Number Xbase++

1.4.16.4 FoxPro

Compiler

Operation	Difference
Bracketed strings	The preprocessor in FoxPro translates #defines inside bracketed strings. X# does not touch the contents of bracketed strings in the preprocessor

Runtime functions

Function	Difference

1.4.16.5 Harbour

Compiler

Operation	Difference

Runtime functions

Function	Difference

1.4.17   Subsystems of the X# Runtime

Description of Various Subsystems

Subsystem	Remarks
Low Level File IO	These functions are implemented in XSharp.Core. There is an important difference between the implementation in XSharp.Core when compared to VO. In VO the file handles returned from functions such as FCreate() and FOpen() are OS file handles. That means that you can also pass them directly to Win32 API Functions. In the X# Runtime that is no longer possible. We use .Net FileStream objects for the File IO. The File handler returned (which is of the IntPtr type) is a unique key into a table where we are storing these File IO objects. The keys are generated from random numbering. You can't and shouldn't rely on the values of these keys.
Static Memory IO	The static memory subsystem allocates memory using the Marshal.AllocHGlobal functionality. Each memory block has 2 guard blocks that contain information about the group number, size and a magic number. We have also implemented memory groups. Unlike in VO you cannot release all blocks in a group by simply closing the Memory Group. If you want to enumerate allocated blocks you should first call MemTrace(TRUE) to enable block tracing. Then create a function with the following prototype `FUNCTION MyMemWalker(pMem AS IntPtr, nSize AS DWORD) AS LOGIC` Then call MemWalk and pass your function as parameter. The runtime will call your function and will pass in all memory blocks that have been allocated and not released yet.
Late Binding Support	The Runtime fully supports late binding. The late binding support still needs some optimizations. We recommend to only use this when necessary. New code should either use the DYNAMIC type or try to use early bound code as much as possible.

1.4.18 Combining X# Runtime and Vulcan Runtime

Technically it is possible to include both the X# and the Vulcan runtime libraries in your application. When you do so then the compiler will assume that you want to use the X# implementations for the XBase types such as USUAL and DATE. If the compiler does not find the XSharp.Core and XSharp.VO assemblies then it will assume you want to map these types to the Vulcan runtime types.

So you can mix things. However if you want to call code in the Vulcan runtime DLLs you may have to use the fully qualified classnames or typenames.

And remember: there is no automatic translation between the X# types and Vulcan types. If you want to convert an X# variable to a Vulcan variable you may have to cast it to an intermediate type first.

Call Vulcans implementation of Left()

```
LOCAL cValue as STRING
cValue := VulcanRTFuncs.Functions.Left("abcdefg",2)
```

If you want to convert an X# usual to a Vulcan usual, cast to OBJECT

```
LOCAL xUsual as USUAL
LOCAL vUsual as Vulcan.__Usual
xUsual := 10
vUsual := (OBJECT) xUsual
```

For dates you can do something similar. In that case you should cast the X# date to a DateTime.

```
LOCAL xDate as DATE
LOCAL vDate as Vulcan.__VODate
xDate := ToDay()                // will call the X# implementation
of ToDay()
vDate := (System.DateTime) xDate
```

1.5   X# Scripting

Below is the text from the presentation from the session that Nikos did in Cologne during the XBase.Future 2017 conference.

The examples from this session are stored during the installation of X# in the folder c:\Users\Public\Documents\XSharp\Scripting

Why endorse scripting?

- Dynamic behavior at runtime

  - Extensibility and flexibility
  - User-defined behavior
- Platform independence
  - System operations defined in a script
- Behavior as data
  - Stored in files, database, cloud
  - Updated at runtime
- Rapid prototyping

Scripting is...

- Expression evaluation
  - Built-in interpreter
  - Self-contained functionality
  - Simple expressions or full statements
- Source file(s) without a project
  - Single file (multiple sources may be loaded by a single script)
  - No need for a full IDE or SDK
  - Dynamic compilation without an IDE
  - Definition of complex structures, classes

X# as a scripting language

- Roslyn script engine
  - C# scripting
- Standalone expressions
  - No START function
  - Global statements, expressions
  - Similar to the macro compiler (but not the same!)
  - Host object maintains state

X# as a scripting language

- Complex declarations allowed
  - Types, functions can be declared
  - No namespaces!
- External references

- o Loading assemblies
- o No implicit access to host assembly
- o No isolation (e.g. separate AppDomain)

The X-Sharp interpreter (xsi.exe)

- Read-Eval-Print Loop (REPL)
- Console application
- Return values are printed to the console
  - o With pretty formatting!
- Maintain context
- Declare LOCAL variables

The X-Sharp interpreter (xsi.exe)

- Can load assemblies, script files
  - o .PRGX extension
  - o #R directive
  - o #LOAD directive
- Can runs scripts from command line
  - o Xsi.exe <script.prgx>
- Passes command-line arguments to scripts
  - o Xsi.exe <script.prgx> <arg> ...

Alternative ways to run scripts

- Setting xsi.exe as default app for .prgx
  - o Also creates file association, but without args
  - o Edit file association in registry
- Manually setting file association
  - o assoc, ftype
- Invoking without the .prgx extension
  - o PATHEXT
- Run without console?
  - o Not possible with xsi.exe since it is a console application

Scripting internals: the submission

- Every script is compiled into a "submission"
  - o Roslyn terminology
- Every line given to the xsi prompt creates a new submission
  - o Inherits previous submission
  - o Previously declared variables remain visible
- Cannot be inspected directly
  - o "SELF" and "SUPER" are not accessible

Scripting internals: the global object

- Statements execute in the context of the global object

- Generated by xsi.exe
  - o InteractiveScriptGlobals class
- Provides access to command-line arguments
- Print function with pretty-formatting options

Scripting internals: script declarations

- Are LOCALs declared in a script really local?
  - o Not when they are declared outside of a method
  - o They become fields of the submission class
- What about FUNCTIONs and PROCEDUREs?
  - o They become methods of the submission class
- Declared types? (CLASSes, STRUCTUREs, ENUMs)
  - o They become nested types in the submission class
  - o Not possible to have extension methods!

Application scripting: the first steps

- Add scripting capabilities to your application!
- Reference the script hosting and code analysis assemblies
  - o XSharp.CodeAnalysis.dll
  - o XSharp.Scripting.dll
- Important namespaces
  - o LanguageService.CodeAnalysis.Scripting
  - o LanguageService.CodeAnalysis.Xsharp.Scripting
- Run a script
  - o XSharpScript.RunAsync("? 'HELLO'")
  - o CompilationErrorException is thrown if the source has errors

Problem: how to pass arguments to a script?

- Passing arguments: the globals object
- The script can access public members of the globals object
  - o The type of the globals object can be custom
- An instance of the globals object can be passed to RunAsync()
  - o Public fields of the globals object can be used to pass arguments to the script
  - o The script will access them as variables

Problem: how to provide an API to the script?

Script API: the globals object

- Public members of the globals object are accessible by the script
  - o Remember: the script is compiled and executed in a different assembly in-memory!
- Not an elegant method to give access to types
  - o But excellent for a function-based API
  - o Self-contained, not prone to errors
- The script does not have direct access to all application types

- Not a security measure!
  - o The script is run in the same AppDomain (in a dynamic assembly)

Script API: using a common assembly

- Scripts can reference assemblies
  - o Through the #R directive
  - o Through the options passed to the RunAsync() call
- Move functions and types that should be accessible by the script to a separate assembly
  - o The assembly can then be referenced by the script
- Can be used in conjunction with the globals object

Problem: how to get results back from a script?

Script result: return value

- Scripts can return a value with a RETURN statement
  - o ...or from a standalone expression!
  - o EvalAsync() returns that value
  - o RunAsync() returns a ScriptState object, from which the return value can also be fetched

Script result: examine script state

- Variables declared by the script can be examined
  - o The ScriptState object returned by RunAsync() includes methods to examine the variables
- ScriptState.GetVariable(string name)

Advanced topics: handling errors

- Compilation errors
  - o CompilationErrorException thrown
  - o Roslyn API provides access to compilation messages
  - o Create script object with XsharpScript.Create()
  - o Compile with script:Compile()
- Returns list of diagnostic messages
- Runtime errors
  - o Exception is thrown
- AggregateException because script is ran as a Task
- e:InnerException property contains the real exception

Advanced topics: strong-typed return value

- By default a script returns OBJECT
- Custom return type can be specified
  - o Create<T>()

- o RunAsync<T>()
- o EvaluateAsync<T>()

Advanced topics: performance tuning

- Pre-compile scripts
  - o Script:Compile()
  - o Compiled scripts can be ran multiple times
  - o Similar to macros
- Delegate can be created with script:CreateDelegate()
- Native image with ngen.exe
  - o Speed-up initial compilation
  - o 64-bit version of ngen must be used for 64-bit CLR!!!
  - o Useful for command-line scripts (xsi.exe)

Advanced topics: functional scripts

- A script cannot be used exactly like a function
  - o Does not accept arguments
- Instead, it needs a global object instance
  - o Is run via a script hosting object
- Additional overhead
- But scripts can evaluate to functions!
  - o Lamda functions or delegates as return type

Advanced topics: accessing application

- Provide a reference to current assembly inmemory
  - o Assembly.GetExecutingAssembly()
  - o Does not work with CoreCLR
- Entities declared in current assembly can be used
  - o Functions & procedures
  - o Types (classes, structures, etc.)
  - o Namespaces

Advanced topics: support for dynamics

- Need to reference the proper assembly
  - o Microsoft.Csharp.dll

1.6   Using X# in Visual Studio

XSharp comes with 2 options for development environments:

1. Chris Pyrgas has adjusted his existing IDE to support XSharp (it is now called XIDE, and compiled in XSharp !)
2. We supply Visual Studio integration for Visual Studio 2017 and later. If you do not have Visual Studio, you can download a (**free!**) community edition from Visual Studio from the web:

https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx

The Visual Studio integration is also work in progress. What is supported in this version is:

- Edit
- Compile
- Debug
- Source control
- Windows Forms Editor
- WPF Editor
- Resource editor
- Settings editor

Known issues:

- Some of the intellisense features are not supported yet.
- There are no editors for VO Binary entities yet
- No support yet for .Net Core, .Net Standard, Portable class libraries and shared projects

1.6.1 Project System

The X# Visual Studio project system allows you to create X# projects with Visual Studio. It has support for a special kind of project file format for X# (the .xsproj files) and it also recognizes various X# specific file types such as PRG files and several "binary" files for VO compatible Form, Server, Menu and FieldSpec definitions.

Part of this project system is the MsBuild support which uses the .xsproj files and creates command lines for the X# compiler based on properties and values in the .xsproj files. The following chapters will briefly discuss solutions and projects and will describe the various project property pages for X# inside Visual Studio.

1.6.1.1 Solution

Visual Studio organizes your source code in Solutions and one or more projects.

A solution is a container for one or more related projects, along with build information, Visual Studio window settings, and any miscellaneous files that aren't associated with a particular project.
A solution is described by a text file (extension .sln) with its own unique format; it's not intended to be edited by hand.
Visual Studio uses two file types (.sln and .suo) to store settings for solutions:

Extension	Name	Description
.sln	Visual Studio Solution	Organizes projects, project items, and solution items in the solution.
.suo	Solution User Options	Stores user-level settings and customizations, such as breakpoints.

Different projects in Visual Studio may target different development languages.

It is very well possible to use C# and X# projects next to each other in the same solutions. And you can set dependency relations between projects for different development languages without problems.
Visual Studio will automatically determine the order in which projects must be build.

If you want you can control this order by opening the Solution Properties dialog (right click on the Solution node in the Solution Explorer and choose "Properties").
The image below shows the Solution Properties dialog for the X# runtime solution.
On this dialog you can also set which project needs to be the "startup project" when you start debugging inside Visual Studio.
As you can see you can also start multiple projects at the same time.
The "Action" combo offers you the choice to Start a program or to start Debugging a program when you start the debugger inside Visual Studio.
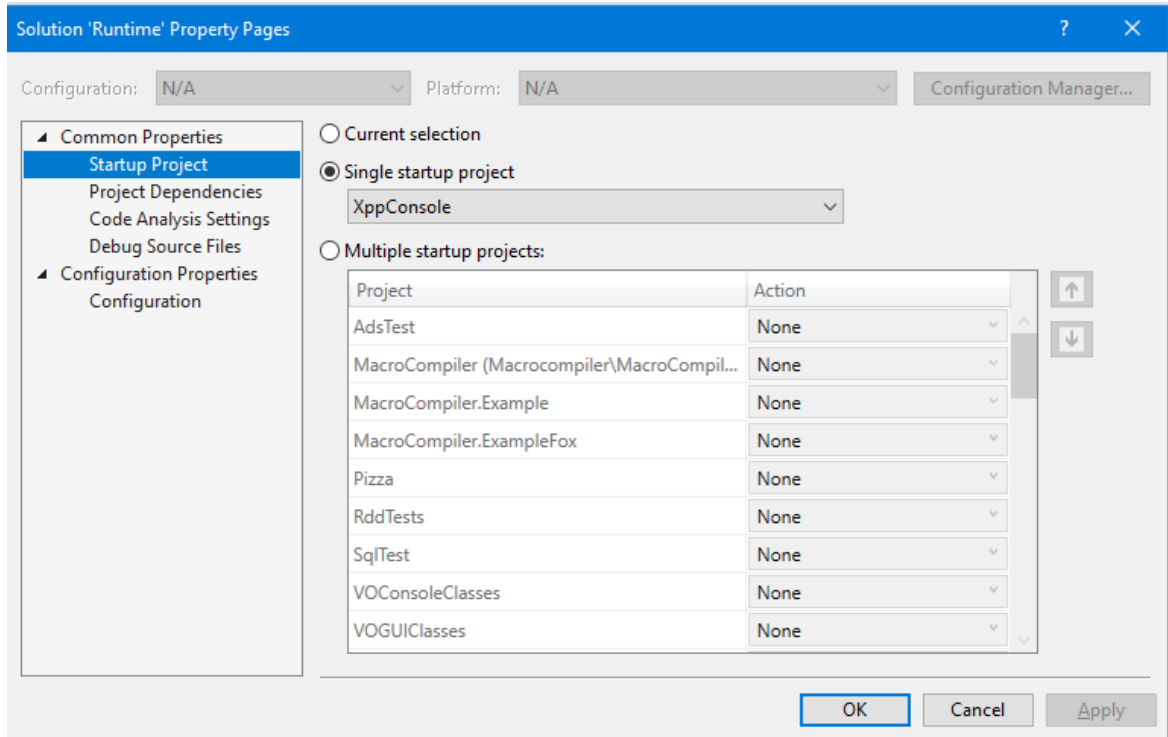
On the "Project Dependencies" tab page you can set the dependencies between projects
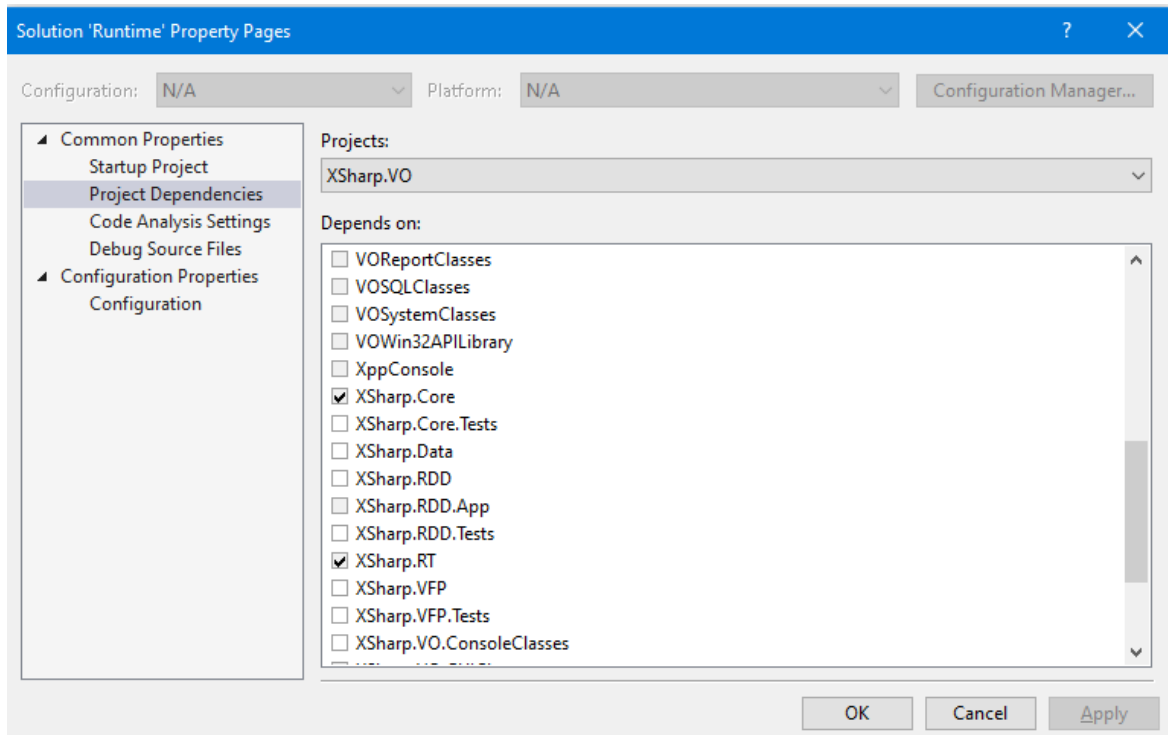
On this dialog you can see that 2 projects are already marked with a check box: these are projects that are added in the references list of the XSharp.VO project.
Other projects have a gray checkbox, which means that they depend on XSharp.VO, so you cannot make XSharp.VO dependent on them (that would introduce a circular relation). Other projects ave w hite checkbox. You could add these to the "Depends on:" list for

XSharp.VO if you want, which would mean that Visual Studio would always build them before XSharp.VO is built.

From this dialog you can also open the "Configuration Manager" with which you can maintain the various Configurations (Normally Debug and Release, but you can to that it you want and the various "Platforms". Normally there is only one Platform called "AnyCpu". But if your project contains C++ code you may have a x86 and a x64 platform as well. This configuration manager is also available in the Visual Studio Build menu. See the next chapter for more information about build configurations.

1.6.1.2   Build Configurations

You need build configurations when you need to build your projects with different settings. For example, Debug and Release are configurations and different compiler options are used accordingly when building them. One configuration is active and is indicated in the command bar at the top of the Visual Studio IDE.
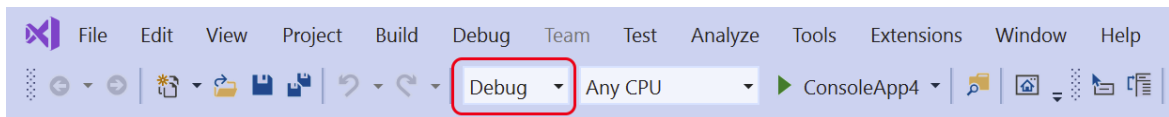
The configuration and the platform control where built output files are stored. Normally, when Visual Studio builds your project, the output is placed in a project subfolder named with the active configuration (for example, bin/Debug), but you can change that.
You can create your own build configurations at the solution and project level. The solution configuration determines which projects are included in the build when that configuration is active. Only the projects that are specified in the active solution configuration will be built. If multiple target platforms are selected in Configuration Manager, all projects that apply to that platform are built. The project configuration determines what build settings and compiler options are used when you build the project.
To create, select, modify, or delete a configuration, you can use the Configuration Manager. To open it, on the menu bar, choose Build > Configuration Manager, or just type Configuration in the search box. You can also use the Solution Configurations list on the Standard toolbar to select a configuration or open the Configuration Manager.
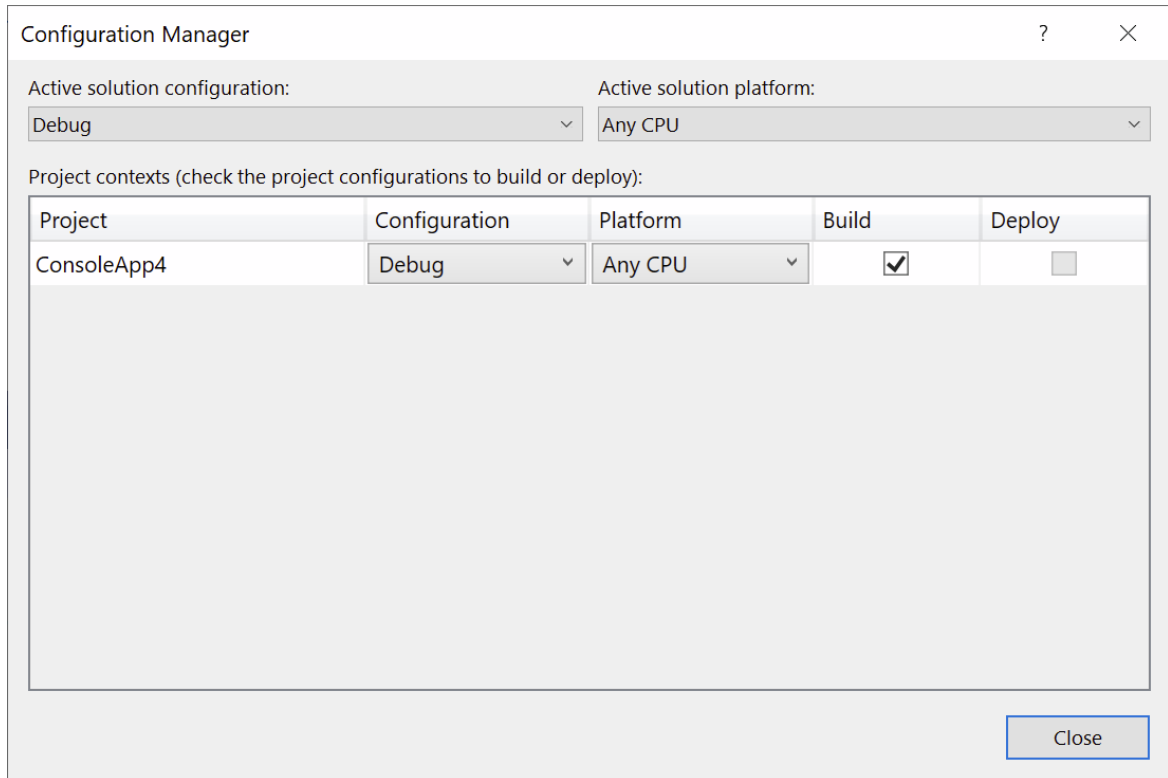
By default, Debug and Release configurations are included in projects that are created by using Visual Studio templates. A Debug configuration supports the debugging of an app, and a Release configuration builds a version of the app that can be deployed. For more information, see How to: Set debug and release configurations. You can also create custom solution configurations and project configurations. For more information, see How to: Create and edit configurations.

Solution Configurations

A solution configuration specifies how projects in the solution are to be built and deployed. To modify a solution configuration or define a new one, in the Configuration Manager, under Active solution configuration, choose Edit or New.
Each entry in the Project contexts box in a solution configuration represents a project in the solution. For every combination of Active solution configuration and Active solution platform, you can set how each project is used. (For more information about solution platforms, see Understand build platforms.)
When you define a new solution configuration and select the Create new project configurations check box, Visual Studio automatically assigns the new configuration to all of the projects. Likewise, when you define a new solution platform and select the Create new project platforms check box, Visual Studio automatically assigns the new platform to all of the projects. Also, if you add a project that targets a new platform, Visual Studio adds that platform to the list of solution platforms and assigns it to all of the projects. You can still modify the settings for each project.
The active solution configuration also provides context to the IDE. For example, if you're working on a project and the configuration specifies that it will be built for a mobile device, the Toolbox displays only items that can be used in a mobile device project.

Project Configurations

The configuration and platform that a project targets are used together to specify the build settings and compiler options to use when it's built. A project can have different settings for each combination of configuration and platform. To modify the properties of a project, open the shortcut menu for the project in Solution Explorer, and then choose Properties.
At the top of the Build tab of the project designer, choose an active configuration to edit its build settings.
See the project properties dialogs for more information about which project properties are configuration dependent and which project properties are configuration independent.

1.6.1.3   Projects

Project properties in Visual Studio are stored in the project files. The file names for X# projects end with the .xsproj extension.
Project files are a special kind of XML files.
These XML files contains various groups of information.
Properties at the project level are stored in <PropertyGroup> nodes.
Some of these properties are "Global" and some of these properties have a condition through which they are only active when a certain "configuration" is selected, such as "Debug" or "Release".
Properties as the dialect and the output filename are global.
Properties such as the optimization, PDB generation and output folders are configuration specific.

The project files also contain the list of references to external assemblies, other projects and COM components and the list of files.
The build system inside Visual Studio uses the contents of the project files and construct a commandline that is passed to the X# compiler to produce the output for the project.
We have developed a set of dialogs that allows you to set the various properties for a project. The following chapters will show you these dialogs and will discuss the various options that you can set on each page.

> *Tip*
>
> *Since the project file is a XML file you can also directly edit the project file if you want and for example add <Import> lines to import common properties from "include" files.*
> *We use that internally for our assemblies, so common properties are declared at one location.*
> *If you do that then you must be careful when using the project property dialogs inside Visual Studio.*
> *These dialogs will not understand that some values were read from an imported file and will change the values in the xsproj file, even when they were imported from an external file.*
> *Also the order of the various property groups is important:*
> *The PropertyGroups with the Pre and PostBuild events are expected to be at the end of the XML file.*

The lists of external references and lists of items inside a project are stored in <ItemGroup> nodes.

1.6.1.3.1 Project Properties

There are several ways that you can set project properties in Visual Studio:
1. There is a <Myprojectname> Properties menu option in the Project Menu
2. There is a context menu option "Properties" on the project icons in the Solution Explorer
3. By choosing the context menu option "Open" on the Properties folder below the Project Icon in the Solution Explorer.

All these options do the same and open the project properties dialog. This dialog has several pages.
- Application
- Language
- Dialect
- Build
- Build Events
- Debug
- Resources
- Settings

A description of each of these pages will follow in the next chapters.

1.6.1.3.1.1 Application

The Application page contains the most important settings for your Visual Studio project

Item	Description	Command item
Default Namespace	Specifies the default namespace for added items, such as classes, that	This is not a commandline option for the compiler but used by Visual Studio.

	are added via the Add New Item Dialog Box.	
Dialect	Select the compiler dialect to use when compiling this project. Changing the dialect may also change the 'Allow NamedArguments' setting on the Language page."	-dialect
Output Type	The type of application to build.	
Startup Object	The name of the class that contains the Start method that you want called when you launch your application. The default for X# applications is the Start function inside the compiler generated Functions class. You can override this setting by specifying another classname here.	-main:
Target Framework Moniker	The version of the Common Language Runtime to use for output assembly.	This is not a commandline option for the compiler but controls the location from which reference assemblies are loaded.
Assembly Name	The name of the output file that will hold assembly metadata.	-out
Output File	The name of the protject's primary output file.	This is derived from the assembly name
Project File	The name of the file containing build, configuration, and other information about the project.	This is the name from the current project
Project Folder	The absolute location of the project	This is the location from the current project
Application Icon	Sets the .ico file to use as your application icon. Please note that if your application contains native resources then this value will be ignored.	-win32icon
Prefer native resource over managed resource	When your application includes a native resource, use this native resource and	-usenativeversion

X#

	do not generate a resource based on the global assembly properties such as AssemblyTitle, AssemblyVersion etc.	
Suppress default Win32 manifest	Suppress default Win32 manifest. You will have to supply your own Win32 manifest if you suppress the default one.	-nowin32manifest
Vulcan Compatible Managed Resources	Use Vulcan Compatible Managed Resources (when 'True' then resources files are included in the assembly without namespace prefix. When 'False' then the resource files are prefixed with the namespace of the app, just like in other .Net languages, such as C#)	-resource

.

1.6.1.3.1.2 Language

The Language page contains some settings that control X# Language specific options. These settings apply to all dialects.

Item	Description	Command item
Allow Late Binding	Allow property access and method calls on expressions of type OBJECT and USUAL.	/lb

Allow Named Arguments	Allow named arguments (Default = TRUE for the Core dialect and FALSE for the other dialects). Changing the dialect may also automatically change this setting.	/namedargs
Allow Unsafe Code	Allow Unsafe code inside this assembly.	/unsafe
Case Sensitive	Enable/Disable case sensitivity.	/cs
Initialize Local variables	Automatically initialize local variables without initialization expression. Please note that for locals of type string the initial value will depend on the 'Initialize strings' setting from the Dialect page.	/initlocals
Overflow Exceptions	Check for Overflow and Underflow for numeric expressions, like the CHECKED keyword.	/ovf
Use Zero Based Arrays	Use Zero Based Arrays.	/az
Enforce SELF	When checked then all references to methods and fields/properties inside a class should be prefixed with SELF (or SUPER)	/enforceself
Allow DOT for instance members	When checked then you can also use the DOT (.) operator to access instance fields, properties and methods. Otherwise the COLON (:) operator is needed	/allowdot
Enforce VIRTUAL/OVERRIDE	When checked then you MUST prefix methods with VIRTUAL and/or OVERRIDE when overriding methods in a parent class or when defining a new method that can be overridden	/enforceoverride
Allow Old Style assignments	When checked then allow the use of the single equals operator (=) for	/allowoldstyleassignments

	assignments. Otherwise the Colon Equals (:=) operator is mandatory	
Modern Syntax	This options disables some old style features such as certain comment characters	/modernsyntax
Enable Memvar support	Enable support for memory variables (MEMVAR, PUBLIC, PRIVATE & PARAMETERS). Please note that this is NOT supported for the Core and Vulcan dialects	/memvar
Enable Undeclared variables support	Enable support for undeclared variables (these are resolved to MEMVARs). Please note that this requires /memvar to be enabled as well.	/undeclared
Enable Implicit Namespace lookup	Enable the implicit lookup of classes defined in assemblies with an Implicit Namespace attribute.	/ins
Prefix classes with default Namespace	Prefix all classes that do not have a namespace prefix and are not in a begin namespace ... end namespace block with the namespace of the assembly.	/ns:<Namespace>
Additional Include paths	Additional include paths for the preprocessor (it also looks through the folders set with the include environment variable).	/i
Alternate standard header file	Name of an alternative standard header file (alternative for XSharpDefs.xh).	/stddefs
Suppress standard header file	Suppress inclusion of the standard header file (XSharpDefs.xh) in every file.	/nostddef

X#

1.6.1.3.1.3 Dialect

The Dialect page contains some additional language settings. Some of these settings apply to all dialects. Others only apply to the given dialect and will only be enabled when the appropriate dialect is chosen.

Item	Description	Command item
All instance methods virtual	Add the virtual modifier to all methods by default (which is the normal Visual Objects behavior).	-vo3
Allow Init() and Axit() as aliases for Constructor/Destructor	Allow Init() and Axit() as aliases for Constructor/Destructor.	-vo1
Compatible IIF Behavior	Compatible IIF Behavior, allow different types of return values in TRUE and FALSE expression.	-vo10
Compatible preprocessor	Makes the preprocessor case insensitive and also controls how #ifdef inspects #defines.	-vo8
Handle problems with incorrect or missing return statements	Allow missing return statements or allow return statements with incorrect return values.	-vo9
Implicit numeric conversions	Implicit numeric integer conversions	-vo4
Initialize strings	Initialize strings to empty string (String.Empty).	-vo2

	Please note that in .NET a NULL_STRING is not the same as a string with length 0. When enabled this will initialize local string variables regardless of the setting of 'initialize locals' setting from the Language page.	
Clipper Compatible intiger divisions	Compatible integer divisions, integer divisions may return a float.	-vo12
Compatible numeric conversions	Compatible arithmetic conversions.	-vo11
Compatible string comparisons	Compatible string comparisons, respects SetExact and collation table.	-vo13
Generate Clipper constructors	Automatically create clipper calling convention constructors for classes without constructor where the parent class has a Clipper Calling convention constructor.	-vo16
Implicit casts and conversions	Compatible implicit casts and Conversions.	-vo7
Implicit Clipper calling convention	Methods without parameters and calling convention are compiled as Clipper calling convention. Please note that without this switch all methods without parameters will be seen as STRICT. Methods with untyped parameters are always seen as CLIPPER calling convention.	-vo5
Implicit pointer conversions	Implicit conversions between typed function PTR and PTR.	-vo6
Treat missing types as USUAL	Missing type clauses for locals, instance variables and parameters are treated as USUAL (VO and Vulcan dialect). The default = TRUE for the VO dialect and FALSE for the other dialects. We strongly	-vo15

	recommend to set this to FALSE because this will help you to find problems in your code and non optimal code. If you have to use the USUAL type we recommend to explicitly declare variables and parameters as USUAL.	
Use FLOAT literals	Store floating point literals as FLOAT and not as System.Double (REAL8).	-vo14
Compatible BEGIN SEQUENCE .. END	Generate code to fully implement the VO compatible BEGIN SEQUENCE .. END SEQUENCE. The compiler generates calls to the runtime functions _SequenceError and _SequenceRecover that you may override in your own code.	-vo17
Inherit from Custom class	All classes are assumed to inherit from the Custom class. This also affects the way in which properties are processed by the compiler.	-fox1
Compatible Array Handling	FoxPro compatible array handling (Allows parenthesized arrays and assigning a single value to an array to fill all elements). WARNING Allowing parenthesized arrays may slow down the execution of your program !(/fox2)	-fox2
Inherit from Abstract class	All classes without parent class inherit from the XPP Abstract class.	-xpp1

1.6.1.3.1.4 Build

The Build properties page contains *Configuration Specific* properties. So you can have different properties here for a Debug and a Release configuration. Please read the Build Configuration topic for more information about build configurations

Item	Description	Command item
Platform Target	Select the platform target when compiling this project. This should be AnyCPU, X86, x64,Arm or Itanium.	-platform
Prefer 32 Bit	Prefer 32 bit when AnyCpu platform is selected.	-platform
Intermediate Output Path	Intermediate Output Path (macros are allowed).	used by Visual Studio/MsBuild
Output Path	Output Path (macros are allowed).	used by Visual Studio/MsBuild
Code Signing KeyFile	Choose a code signing key file.	-keyfile
Delayed sign only	Delayed signing.	-delaysign

Sign the output assembly	Sign the assembly	-keyfile
Extra Command Line Options	User-Defined Command Line options.	you can specify additional commandline options here
Optimize	Should compiler optimize output?	-optimize
Register for COM interop	Register the output assembly for COM Interop (requires administrator rights	This will run a tool after the build process to register the assembly for COM interop.
Use Shared Compiler	Should the shared compiler be used to compile the project? (Faster, but may hide some compiler errors)	-shared
Defines for the preprocessor	Defines for the preprocessor.	-define
Generate preprocessor output	Save the output from the preprocessor to .ppo files.	-ppo
Suppress Specific Warnings	Specify a list of warnings to suppress.	-nowarn
Warning Level	Set the warning level to a value between 0 and 4.	-warn
Warnings As Errors	Treat warnings as errors.	-warnaserror
Generate XML doc comments file	Generate XML doc comments file.	-doc
XML doc comments file name	XML doc comments file name.	-doc

1.6.1.3.1.5 Build Events

On the Build Events property page you can specify command lines that need to be run before or after the build process.
You can use this for example to run a tool that generates source code or to copy the output DLLs to another folder.

1.6.1.3.1.6 Debug

Item	Description	Command item
Command	The debug command to execute.	
Command Arguments	The command line arguments to pass to the application.	

Generate Debug Information	Generate Debug Information (none, full, pdbonly).	
Working Directory	The application's working directory. By default, the directory containing the project file.	
Enable unmanaged debugging	Enable unmanaged debugging.	

1.6.1.3.1.7 Resources

The Resources Property page allows you to open up a resources editor for project wide resources. These resources can be files, strings, images etc.
The Resource editor is the normal Visual Studio resource editor.
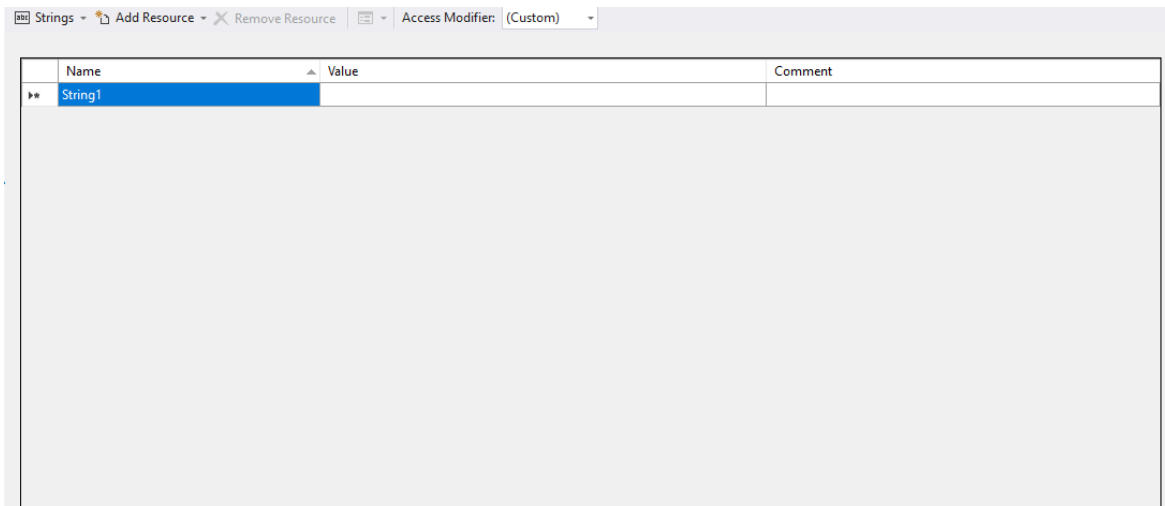
This resource editor will generate a Resources.Resx file in the Properties folder of the project.
The "Access Modifier" combobox at the top of the screen allows you to control the code generation for these resources:

- With (Custom) then no code will be generated
- With "Internal" then X# code will be generated with internal visibility
- With "Public" then X# code will be generated with public visibility

The value of the Access Modifier also sets the "Custom tool" in the properties dialog for the Resources.Resx file.

1.6.1.3.1.8 Settings

Application settings enable you to store application information dynamically.

Settings allow you to store information on the client computer that should not be included in the application code (for example a connection string), user preferences, and other information you need at run time.

Each application setting must have a unique name. The name can be any combination of letters, numbers, or an underscore that does not start with a number, and it cannot have spaces. The name is changed through the Name property.

Application settings can be stored as any data type that is serialized to XML or has a TypeConverter that implements ToString/FromString. The most common types are String, Integer, and Boolean, but you can also store values as Color, Object, or as a connection string.

Application settings also hold a value. The value is set with the Value property and must match the data type of the setting.

In addition, application settings can be bound to a property of a form or control at design time.

There are two types of application settings, based on scope:
- Application-scoped settings can be used for information such as a URL for a web service or a database connection string. These values are associated with the application. Therefore, users cannot change them at run time.
- User-scoped settings can be used for information such as persisting the last position of a form or a font preference. Users can change these values at run time.

You can change the type of a setting by using the Scope property.

The project system stores application settings in two XML files:

- an app.config file, which is created at design time when you create the first application setting
- a user.config file, which is created at run time when the user who runs the application changes the value of any user setting.

Notice that changes in user settings are not written to disk unless the application specifically calls a method to do this.

The Settings Property page allows you to open up a settings editor for project settings. The definition for these settings will be stored in the Properties folder of your project.

The values of these settings will be stored in a app.config file in the root folder of your project.

Just like in the resource editor you can also choose the visibility of the generated code (the IDE generates a file called Settings.Designer.prg). This also changes the "Custom Tool" property of the Settings file.

When you compile your project then MsBuild will generate a copy of app.config and will rename it to <yourapp>.exe.config in the output folder of your project.

1.6.1.3.2 References

XSharp projects inside Visual studio work with source code items and may contain references to code defined in external libraries.
Inside Visual Studio you can set these references through the References dialog.
In short there are 3 kinds of references:
- External .Net assemblies
- External COM components
- Other projects inside the same Visual Studio solution.

References to unmanaged code

You cannot add references to unmanaged code using the Project References.
To call unmanaged code you will have to declare either functions or procedures with the _DLL prefix, or you declare static methods or functions and add a special [DllImport()] attribute to them.

1.6.1.3.2.1 .Net

The page with assembly references shows the list of assemblies that were found on the developers machine.
You can select a reference from this list and this reference will be used without extra work.

Component Name	Version	Runtime	Path
Accessibility	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
CustomMarshalers	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
ISymWrapper	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.Activities.Build	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.Build.Conversio...	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.Build	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.Build.Engine	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.Build.Framework	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.Build.Tasks.v4.0	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.Build.Utilities.v4.0	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.CSharp	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.JScript	10.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.VisualBasic.Com...	10.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.VisualBasic.Com...	10.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...
Microsoft.VisualBasic	10.0.0.0	v4.0.30319	C:\Program Files (x86)\Re...

1.6.1.3.2.2 COM

The COM page in the Add Reference dialog lists the COM components that were found in the registry on your machine.
You can select a component from this list.

However, these components cannot be consumed directly by the X# compiler. Therefore Visual Studio calls a tool (tlbimp.exe) that reads the typelibrary from the COM component and produces a managed wrapper around this COM object. This wrapper usually has a name that starts with "interop". In the Email example this is the case for the Internet Explorer component for which a Interop.SHDocVw.dll is generated.
If the COM component is an ActiveX then a second assembly will be generated that has code that declares an object that inherits from System.Windows.Forms.AxHost, for the ActiveX control. Visual Stdio calls the tool "aximp.exe" for this. The file names for these wrappers usually start with "axinterop", such as "AxInterop.SHDocVw.dll"
Some COM components are used a lot and for these components a so called "Primary Interop Assembly" is installed on your machines. For these components no "interop" assemblies will be generated but the primary interop assemblies will be used when compiling. For example for ADO there is a primary interop assembly in the GAC.

Component Name	TypeLib Version	Path
AccessibilityCplAdmin 1.0 Type L...	1.0	C:\Windows\SysWOW64\Acces...
ActiveMovie control type library	1.0	C:\Windows\System32\quartz.dll
AddinLoaderLib	1.0	C:\Users\Geert\AppData\Local\...
Amyuni Document Converter Ac...	5.5	C:\PROGRA~2\COMMON~1\VI...
AP Client 1.0 HelpPane Type Libr...	1.0	C:\Windows\SysWOW64\HelpP...
AP Client 1.0 Type Library	1.0	C:\Windows\HelpPane.exe
AppIdPolicyEngineApi 1.0 Type L...	1.0	C:\Windows\SysWOW64\AppId...
Assistance Platform Client 1.0 Da...	1.0	C:\Windows\SysWOW64\ApDs....
ATL 2.0 Type Library	1.0	C:\Windows\SysWOW64\atl.dll
AutoHelper 1.0 Type Library	1.0	C:\Program Files (x86)\Microsof...
azroles 1.0 Type Library	1.0	C:\WINDOWS\system32\azroles...
bbListView OLE Control module	2.7	C:\Windows\SysWOW64\selfre...
CertCli 1.0 Type Library	1.0	C:\WINDOWS\system32\certcli....
CertEnc 1.0 Type Library	1.0	C:\WINDOWS\system32\certen...
CertEnroll 1.0 Type Library	1.0	C:\WINDOWS\system32\CertEn...
... 1.0 T... L:L..	1.0	C:\WINDOWS\ ... 32\ : .ll

1.6.1.3.2.3 Project

A third type of dependency is between Visual Studio projects. When you add a dependency of this type then Visual Studio (or actually MsBuild) will try to build that project first before building the project that depends on it. Unlike in Visual Objects you do not have to import a prototype library for the output generated by the project. Visual Studio (and our editor integration) will directly consume the output from the other projects.

If the other project is also a X# Project then our source code editor support code will be able to jump between the declaration of a type and the place where it is used, even if it is in another project. When your X# project depends on project in another languages (such as C#) then we will consume the output of that project like we do with "normal" external .Net assemblies.

Project Name	Project Directory
Project Package	C:\XSharp\DevPublic\VisualStudio\ProjectPackage
VODesigners_tester	C:\XSharp\DevPublic\VisualStudio\VODesigners_t...
XsCodeModelTest	C:\XSharp\DevPublic\VisualStudio\XsCodeModelT...
XSharp.Build	C:\XSharp\DevPublic\VisualStudio\XSharp.Build
XSharpCodeDomProvider	C:\XSharp\DevPublic\VisualStudio\XSharpCodeDo...
XSharpCodeGenerator	C:\XSharp\DevPublic\VisualStudio\CodeGenerator

1.6.1.4 Project Items

Enter topic text here.

1.6.1.4.1 Source code Items

1.6.1.4.2 Forms

1.6.1.4.3 Other Item types

1.6.1.4.4 Native Resources

1.6.1.4.5 Managed Resources

Enter topic text here.

1.6.1.4.6 Settings

1.6.2   Source Code Editor

1.6.2.1   Text Editor Options

The various settings for the Visual Studio editors can be controlled from the Tools/Options menu.
The values for the settings are stored in the file XSharp\EditorSettings.json in your roaming profile folder.
If you have more that one installation of Visual Studio then the settings will be shared between these installations.
When you open that menu point you will see a window like this:
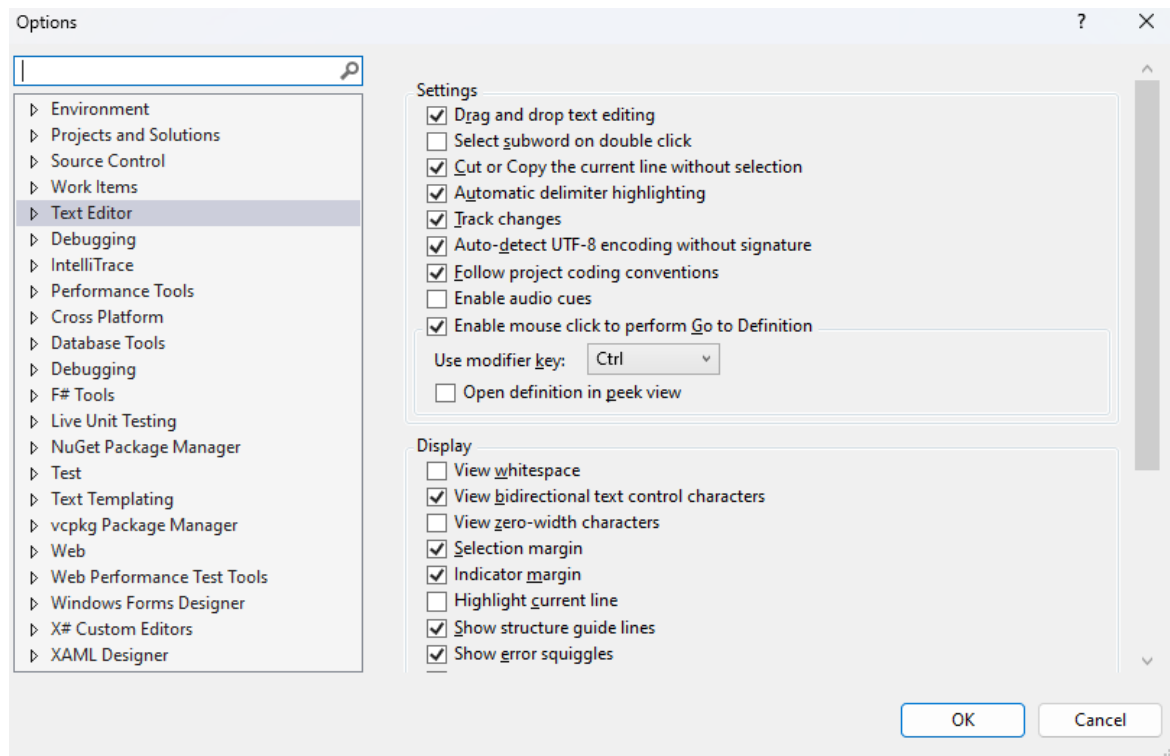
Tools Options dialog
On the left hand side here is a treeview from where you can select the category of the option you are interested in.
This dialog contains a LOT of option.
To ease the navigation you can use the edit control on top to filter.
All X# options can be found by typing X# in this control:
You will then see the categories for X#:
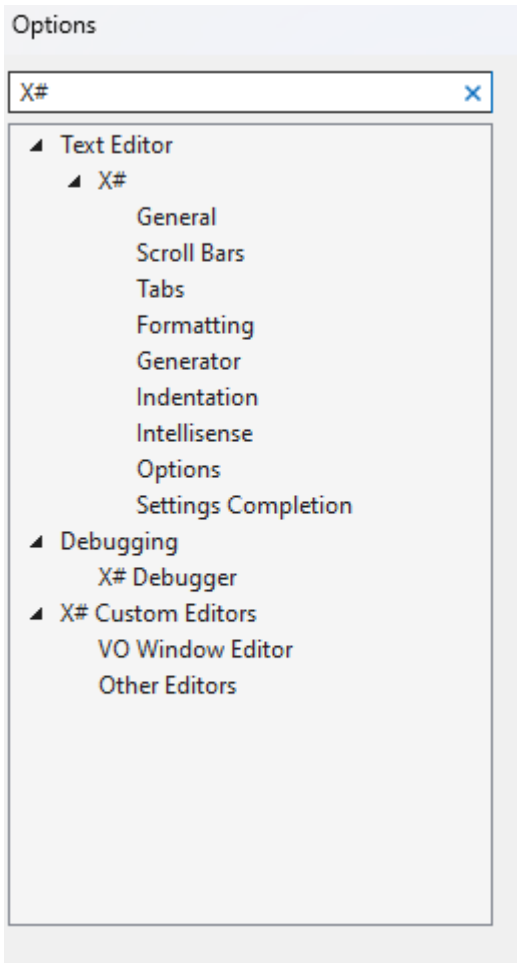
Tools Options filtered on X#

The Text Editor category (which was also visible on the first image) contain settings for all languages that are supported by Visual Studio or by an extension installed inside Visual Studio (such as our Visual Studio extension for X#).

There is also an entry under the Debugging category for settings used by the X# expression evaluator inside the debugger

The last category contains some settings for the VO compatible form editor and for some of the other X# specific editors.

1.6.2.1.1 General Options

This page contains generic editor settings for the X# language.
The page may change between different versions of Visual Studio
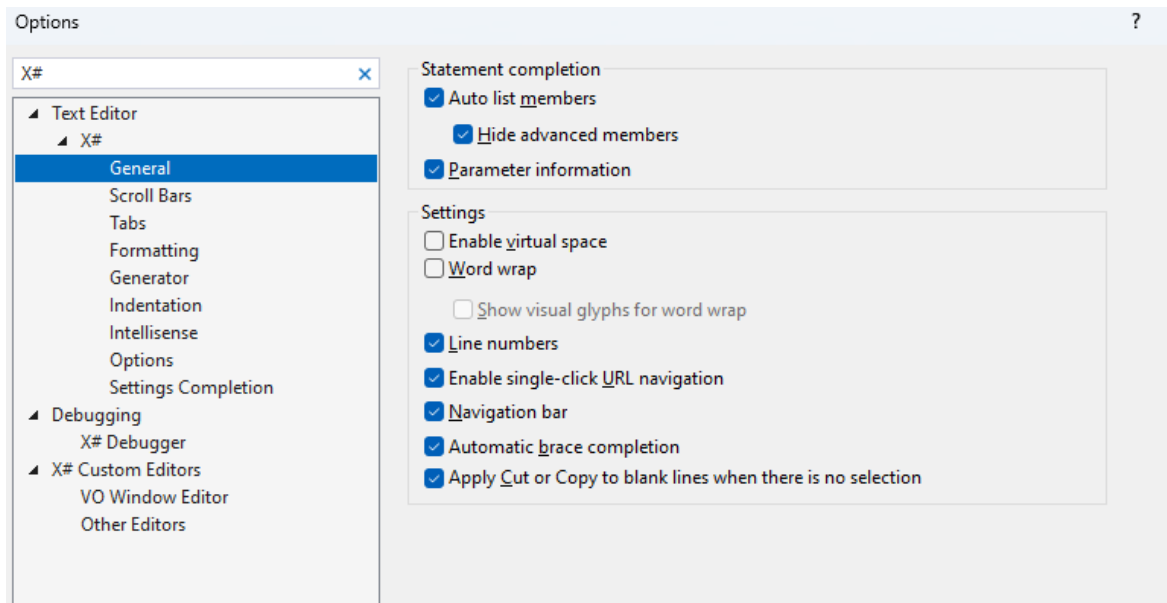
1.6.2.1.2 Scroll Bars

This page contains generic settings w.r.t. scroll bars.
The page may change between different versions of Visual Studio.
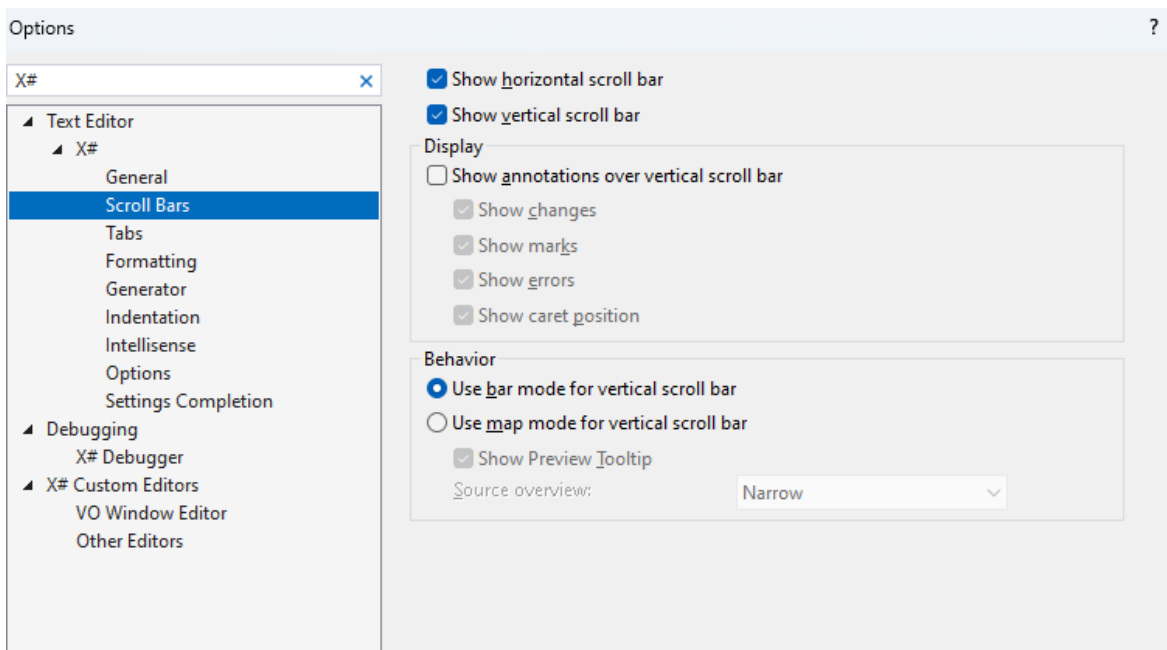
1.6.2.1.3 Tabs

This page contains generic settings w.r.t. tabs..
The page may change between different versions of Visual Studio.
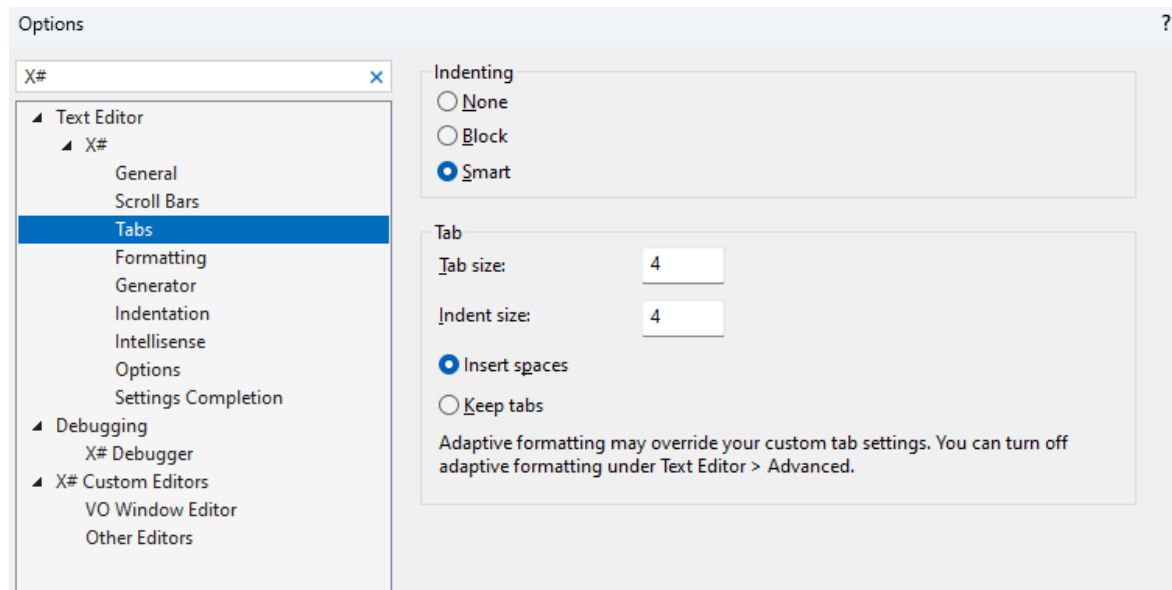
1.6.2.1.4 Formatting

This page X# specific settings w.r.t. source code formatting
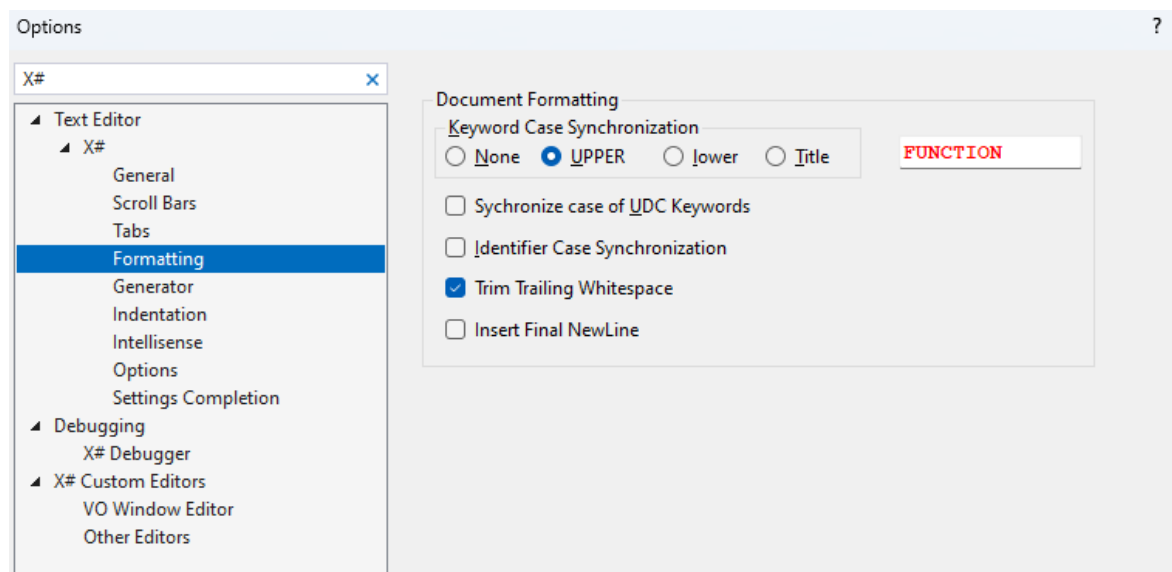
1.6.2.1.5 Generator

This page X# specific settings w.r.t. source code generating inside VS.
This applies to the source code generated by the Windows Forms editor but also for the source code generated when you choose "Goto Definition" for a member that is defined in an external assembly.
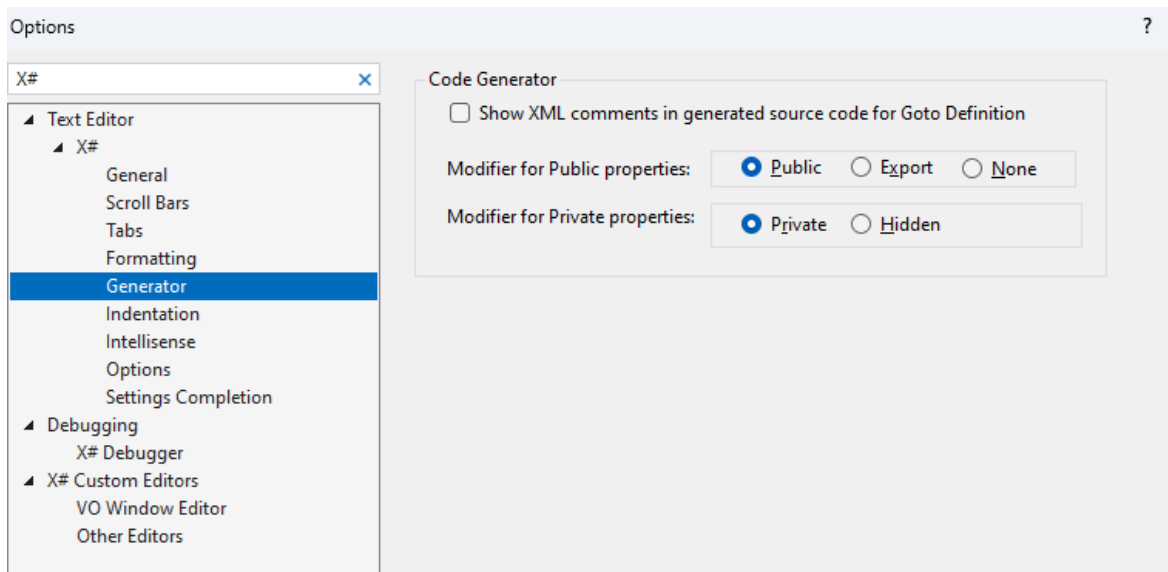
1.6.2.1.6 Indentation

This page contains X# specific settings w.r.t. source indentation. This setting is used by the editor when you choose the option to indent "Smart" on the "Tabs" page.
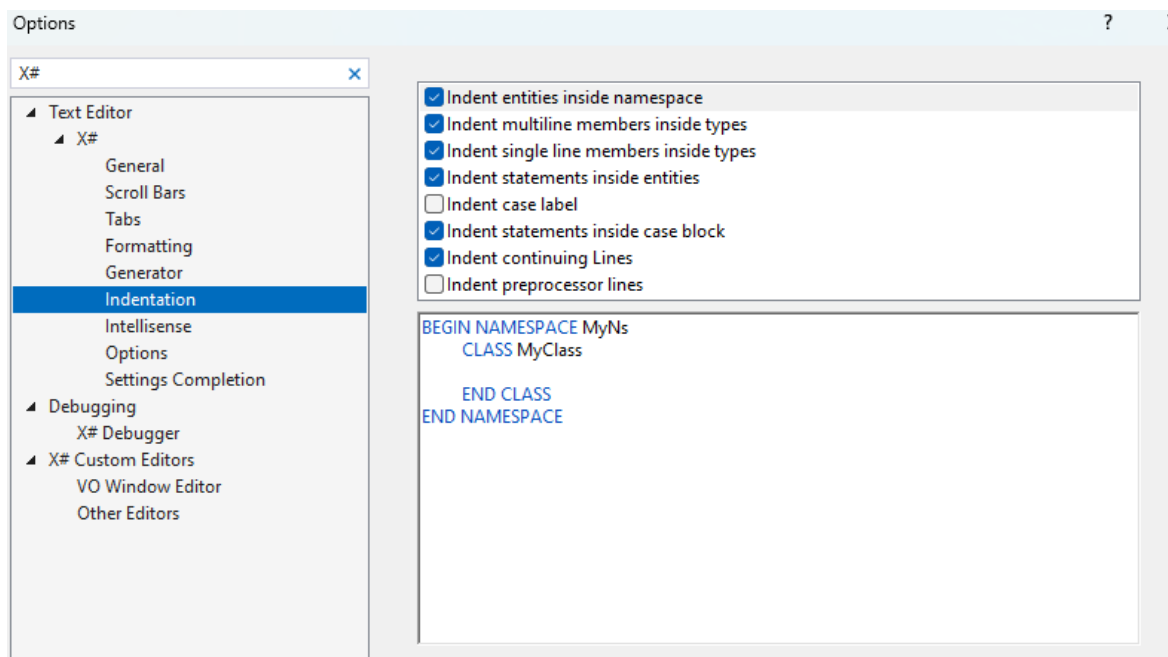
The listview on top shows the options that you can choose from. The edit control on the bottom shows the result of the setting.

1.6.2.1.7 Intellisense

This page contains specific X# settings w.r.t. code completion and the dropdowns in the editor.
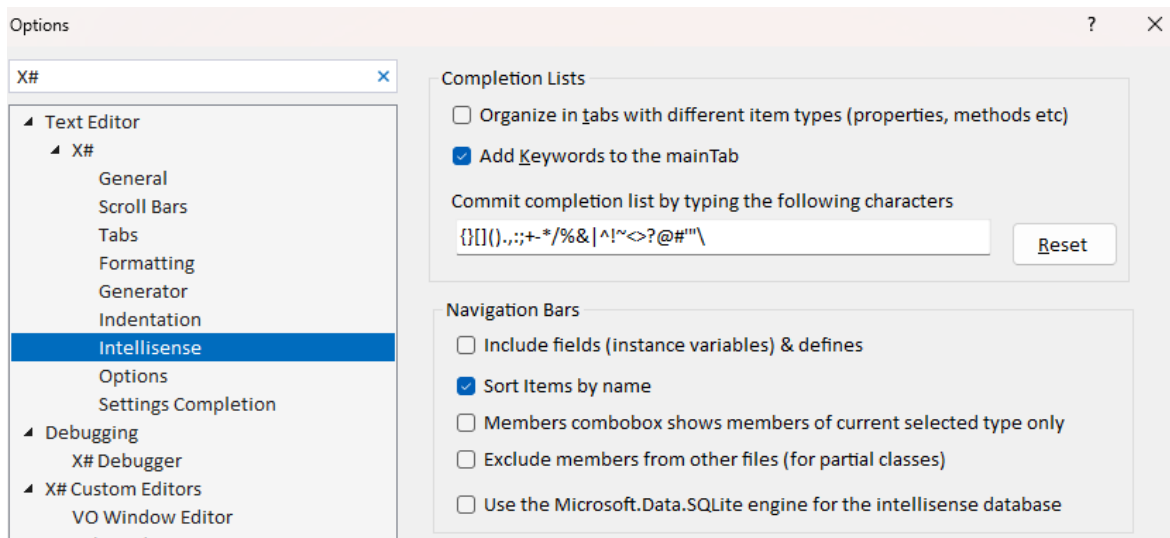
1.6.2.1.8 Options

This page contains X# specific switches that allow you to control features inside the X# source code editor.
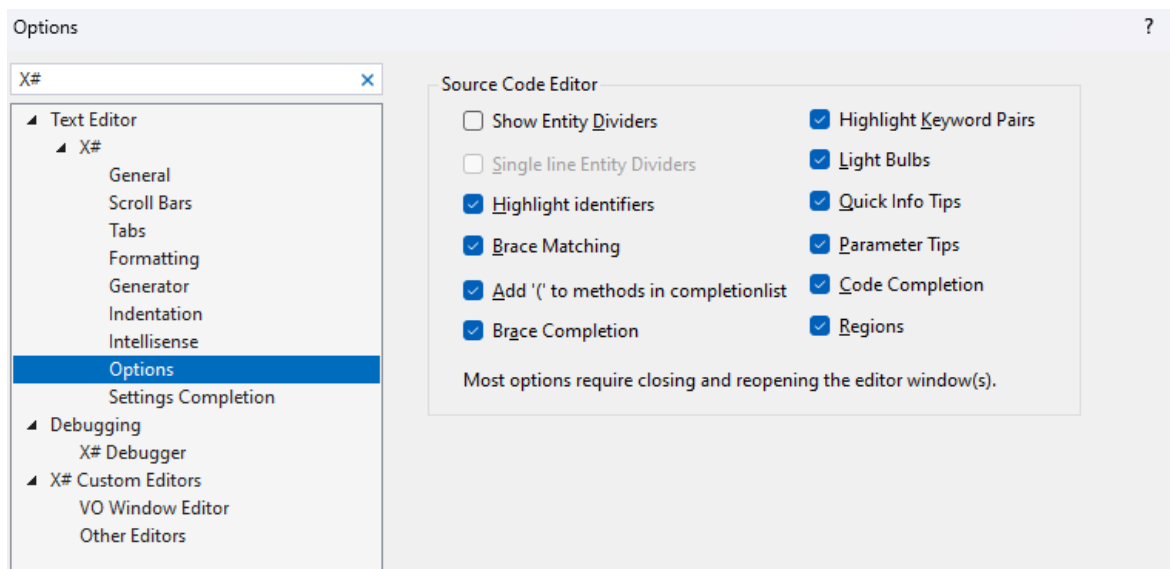
1.6.2.1.9 Settings Completion

This page contains X# specific settings related to code completion.
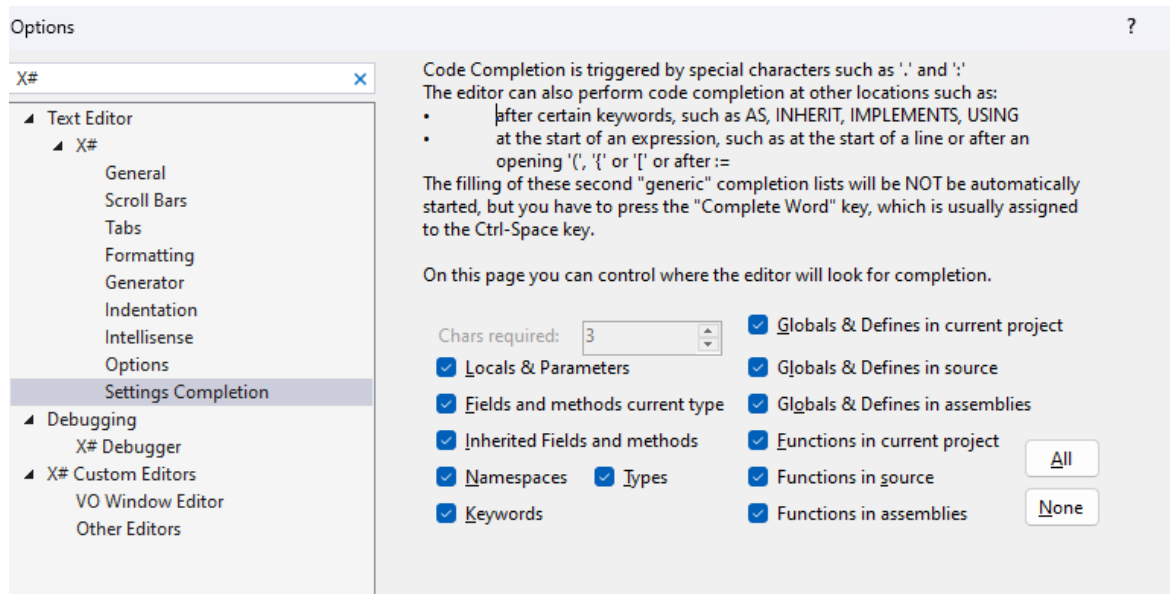
1.6.2.2 Keyword Coloring

Visual studio gives keywords and other parts of the syntax colors based on their purpose, the following items are supported by X#:

Keywords
Identifiers
Comment s (single line, multi line etc)
Operators   (+. -, := etc)
String s   (also for Char constants)
Numbers   (also for date and datetime literals)
Literals   (NIL, Symbols, _AND, _OR, _NOT, _XOR)
Preprocessor keywords
* contents of Text .. EndText
Inactive/Hidden code inside #IFDEF
* Highlighted Word

* These are special XSharp colors. The other colors take the default Visual Studio editor colors for that category

You can change the options for this by going to Tools > Options >
Note that your settings will reset if you switch your color scheme in Visual Studio.

1.6.2.3 Highlighting Errors

At this moments errors are not highlighted in the source code editor. This is planned for a future version of X#.

1.6.2.4 Regions

To unclutter your view of the code, choose the small gray box with the minus sign inside it in the margin of the first line of the constructor or place the cursor anywhere in the constructor code and press Ctrl+M, Ctrl+M.

The code block collapses to just the first line, followed by an ellipsis. To expand the code block again, click the same gray box that now has a plus sign in it, or press Ctrl+M, Ctrl+M again. This feature is called Outlining and is especially useful when you're collapsing long methods or entire classes.

```
60      IF InStr( "MDX", cIndexExt )
61          lSelectionActive := TRUE
62          siSelectionStatus := DBSELECTIONNULL
63          cbSelectionIndexingExpression := DBSelectionIndex{ SELF, __DBSDBOrderInfo( DBOI_EXPRESSION, "", 0 ), wWorkArea }
64      ELSE
65          lCDXSelectionActive := TRUE
66      ENDIF
67
68      __DBSSetSelect( dwCurrentWorkArea  ) //SE-060527
69
70      #IFDEF __DEBUG__
71          DBFDebug("Leaving "+__ENTITY__)
72      #ENDIF
73      RETURN
```

```
60  +    IF InStr( "MDX", cIndexExt )...
64  +    ELSE...
66       ENDIF
67
68       __DBSSetSelect( dwCurrentWorkArea  ) //SE-060527
69
70  +    #IFDEF __DEBUG__...
73       RETURN
```

1.6.2.5 Blocks

Visual Studio makes it straightforward to select, cut, copy, and paste sections of your code. Rectangular blocks of code can be highlighted and selected by using Alt + Drag on the code block. The formatting within this block selection is retained when pasting the block.

Block selections can also be edited to modify text on multiple lines at once. A zero-width block selection can be used to place the caret in front of many lines at once, and allows typing identical text on multiple lines:

example
example
example
example
example

1.6.2.6 Parameter Tips

Parameter Info gives you information about the number, names, and types of parameters required by a method, attribute generic type parameter.
The parameter in bold indicates the next parameter that is required as you type the function. For overloaded functions, you can use the Up and Down arrow keys to view alternative parameter information for the function overloads.

When you annotate functions and parameters with XML Documentation comments, the comments will display as Parameter Info. For more information, see Supply XML code comments.
You can manually invoke Parameter Info by choosing Edit > IntelliSense > Parameter Info, by pressing Ctrl+Shift+Space, or by choosing the Parameter Info button on the editor toolbar.

1.6.2.7 Quick Info

Quick Info displays the complete declaration for any identifier in your code. When you select a member from the List Members box, Quick Info also appears.

You can manually invoke Quick Info by choosing Edit > IntelliSense > Quick Info, by pressing Ctrl+I, or by choosing the Quick Info button on the editor toolbar.

If a function is overloaded, IntelliSense may not display information for all forms of the overload.

```
FUNCTION Start() AS VOID STRICT
    ? "Hello World! Today is ",ToDay()
    System.String.Compare()
    WAIT
```

Dev | C:\XSharp\Dev
DevPublic | C:\XSharp\DevPublic
DevRt | C:\XSharp\DevRt
radix | C:\Work\radix
RP2Net | C:\ReportPro\RP2Net
SHFB | C:\XSharp\SHFB
VSSDK-Extensibility-Samples | C:\XSharp\VSSDK-Extensibility-Samp

Static Export Method Compare(strA AS STRING, strB AS STRING) AS INT (mscorlib.dll)

Compares two specified System.String objects and returns an integer that indicates their relative position in the sort order.
Returns: A 32-bit signed integer that indicates the lexical relationship between the two comparands.Value Condition Less than zero strA precedes strB in the sort order.
Zero strA occurs in the same position as strB in the sort order.
Greater than zero strA follows strB in the sort order.

```
FUNCTION Start() AS VOID STRICT
    ? "Hello World! Today is ",ToDay()
    System.String.Compare(
    WAIT
```

DevPublic | C:\XSharp\DevPublic
DevRt | C:\XSharp\DevRt
radix | C:\Work\radix
RP2Net | C:\ReportPro\RP2Net
SHFB | C:\XSharp\SHFB
VSSDK-Extensibility-Samples | C:\XSharp\VSSDK-Extensibility-Samples

▲ 3 of 10 ▼ Compare(strA AS STRING, strB AS STRING, comparisonType AS System.StringComparison) AS INT
Compares two specified System.String objects using the specified rules, and returns an integer that indicates their relative position in the sort order.
strA: The first string to compare.

```
FUNCTION Start() AS VOID STRICT
    ? "Hello World! Today is ",ToDay()
    ? Test()
    WAIT
    RETUR
```

Export Function Test() AS STRING (C:\Users\robert\source\repos\ConsoleApplication4\ConsoleApplication4\Program.prg)

Some summary
Returns: Some text
Remarks: Some remarks

```
/// <summary>Some summary</summary>
/// <returns>Some text</returns>
/// <remarks>Some remarks</remarks>
```

1.6.2.8 Code Completion

IntelliSense is an invaluable resource when you're coding. It can show you information about available members of a type, or parameter details for different overloads of a method. You can also use IntelliSense to complete a word after you type enough characters to disambiguate it.
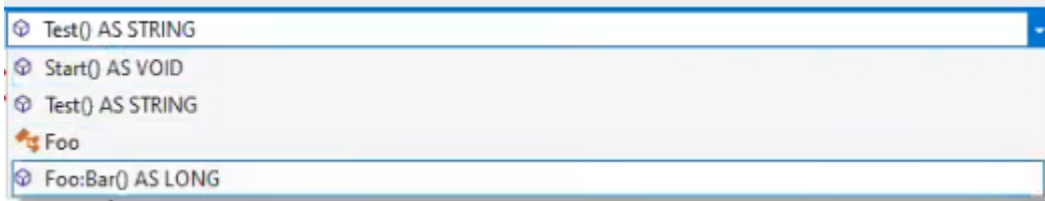
While typing bits of code, you see IntelliSense show you Quick Info about the query symbol.
To insert the rest of the word query by using IntelliSense's word completion functionality, press Tab.

```
FUNCTION Start() AS VOID STRICT
        local aList as List<Int>
    ? "Hello World! Today is ",ToDay()
    ? Test()                I
    aList:
    WAIT            ⊕ Add
                    ⊕ AddRange
    RETURN         ⊕ AsParallel        AsParallel(source AS System.Collections.IEnumerable) AS System.Linq.ParallelQuery
                    ⊕ AsQueryable
                    ⊕ AsReadOnly
/// <summa     ⊕ BinarySearch   ummary</summary>
                    ⚑ Capacity
                    ⊕ Cast
                    ⊕ Clear
```

```
loc|
    ≣ LOCAL              LC
    ⊕ Locate             F
    ≣ LOCK
    ⊕ LockCurrentRecord
    ⊟ LockMethod
    ⊕ LockSelection
```
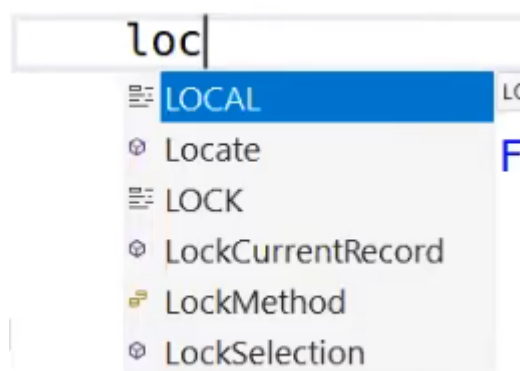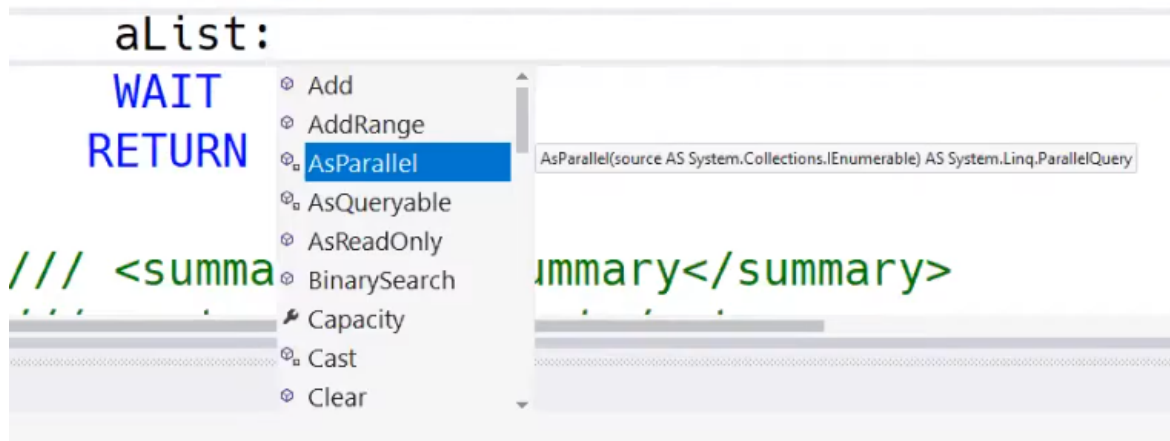
1.6.2.9   Editor combo boxes

1.6.2.10   Goto definition

The Visual Studio editor makes it easy to inspect the definition of a type, method, etc. One way is to navigate to the file that contains the definition, for example by choosing Go to Definition or pressing F12 anywhere the symbol is referenced.
To do this, right-click on any occurrence of string and choose Peek Definition from the content menu. Or, press Alt+F12.
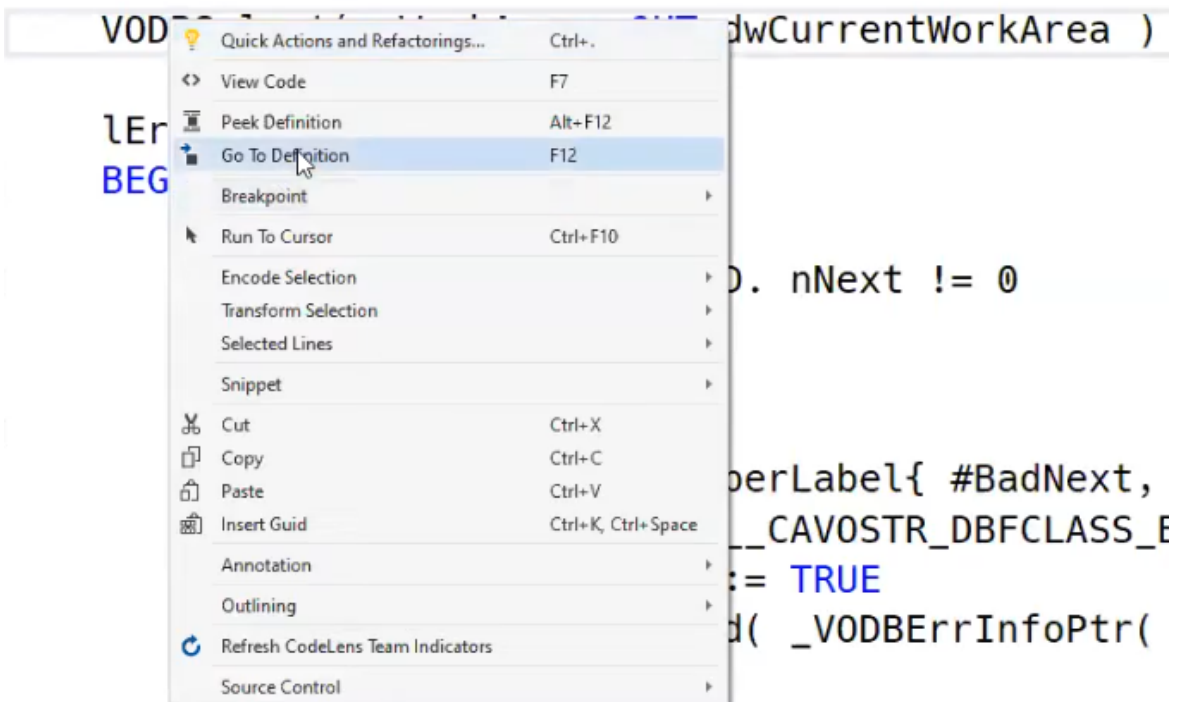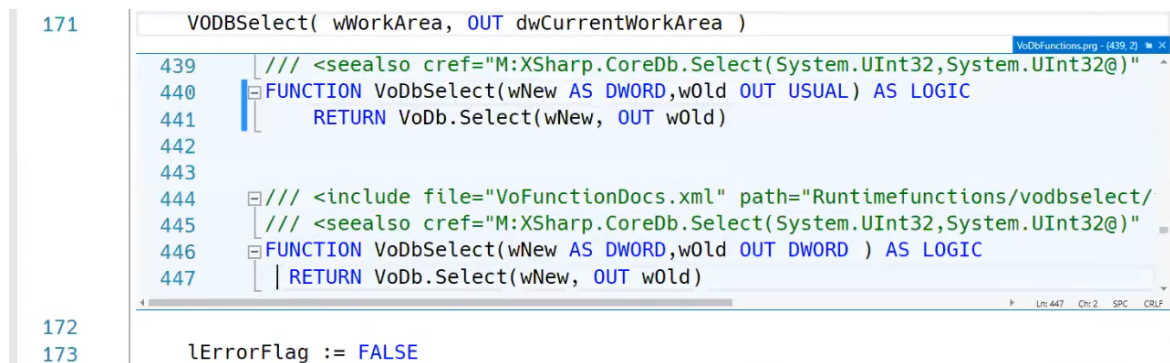
1.6.2.11 Peek definition

The Visual Studio editor makes it easy to inspect the definition of a type, method, etc. One way is to navigate to the file that contains the definition, for example by choosing Peek Definition anywhere the symbol is referenced.
To do this, right-click on any occurrence of string and choose Peek Definition from the content menu. Or, press Alt+F12.
A pop-up window appears with the definition of the String class. You can scroll within the pop-up window, or even peek at the definition of another type from the peeked code.
Close the peeked definition window by choosing the small box with an "x" at the top right of the pop-up window.

```
171        VODBSelect( wWorkArea, OUT dwCurrentWorkArea )
                                                          VoDbFunctions.prg - (439, 2)  ⬚ X
   439    /// <seealso cref="M:XSharp.CoreDb.Select(System.UInt32,System.UInt32@)"  ▲
   440   FUNCTION VoDbSelect(wNew AS DWORD,wOld OUT USUAL) AS LOGIC
   441        RETURN VoDb.Select(wNew, OUT wOld)
   442
   443
   444   /// <include file="VoFunctionDocs.xml" path="Runtimefunctions/vodbselect/
   445    /// <seealso cref="M:XSharp.CoreDb.Select(System.UInt32,System.UInt32@)"
   446   FUNCTION VoDbSelect(wNew AS DWORD,wOld OUT DWORD ) AS LOGIC
   447    │ RETURN VoDb.Select(wNew, OUT wOld)
                                                        ▶  Ln: 447   Ch: 2   SPC   CRLF
172
173        lErrorFlag := FALSE
```

1.6.2.12 Inactive conditional regions

If you use conditional compilation with #ifdef .. #endif regions then the inactive regions will be visible with different color in the editor:
Below is a piece of code from the CDX RDD. The CHECKVERSIONS define is not defined, so there is an inactive region in the editor, which is shown in light gray.

```
        METHOD Write(oPage AS CdxPage) AS LOGIC
            LOCAL isOk AS LOGIC
#ifdef CHECKVERSIONS
            SELF:_PageList:CheckVersion(SELF:Root:RootVersion)
#endif
        IF oPage:PageNo == -1
            oPage:PageNo := SELF:FindFreePage()
            oPage:IsHot  := TRUE
            SELF:_PageList:SetPage(oPage:PageNo, oPage)
        ENDIF
        if oPage:PageNo < 0
            var cMessage := i"Trying to write to negative pageno {oPage:PageNo} in CDX"
            DebOut32(cMessage)
            THROW IOException{cMessage}
        ENDIF
        IF oPage:IsHot
```

1.6.2.13 Brace matching

When using Visual Studio, if your cursor is next to a brace it will highlight the corresponding opening or closing brace on screen.

```
(sample text)
```

The color for the brace matching can be set in the Tools/Options dialog under the Environment/Fonts and Colors node.

The X# specific colors all start with the text "X#"

1.6.2.14 Highlight Identifiers

When you select a specific word, it shows you all the places this word is used, this is case sensitive.

The color for the highlights can be set in the Tools/Options dialog under the Environment/Fonts and Colors node. The X# specific colors all start with the text "X#"

1.6.2.15 Highlight Keywords

The keyword matching feature will highlight keyword pairs in the editor as you can see in the image below.

```
FUNCTION Start() AS INT
    LOCAL oXApp AS XApp
    TRY
        oXApp := XApp{}
        oXApp:Start()
    CATCH oException AS Exception
        ErrorDialog(oException)
    END TRY
RETURN 0
```

The TRY, CATCH and END TRY keywords are highlighted so you can see that they belong to each other

If the cursor is located on a RETURN statement then the matching FUNCTION or METHOD will be highlighted,

```
FUNCTION Start() AS INT
    LOCAL oXApp AS XApp
    TRY
        oXApp := XApp{}
        oXApp:Start()
    CATCH oException AS Exception
        ErrorDialog(oException)
    END TRY
RETURN 0
```

The color for the highlights can be set in the Tools/Options dialog under the Environment/Fonts and Colors node. The X# specific colors all start with the text "X#"

1.6.2.16 Indenting code

Visual studio automatically formats your code, so that you have a clearer overview of your code, and which parts do what.
This only happens if you have set the Indenting option on the Tabs page to "Smart"
The rules for the formatting are defined on the Indentation page

```
lErrorFlag := FALSE
IF true
    sf;ljfas;afs
else
    ;lj;asfljl;afs
endif
```

1.6.2.17 Snippets

Visual Studio provides useful code snippets that you can use to quickly and easily generate commonly used code blocks.
To do this, place your cursor just above the final closing brace in the file, a pop-up dialog box appears with information about the code snippet.
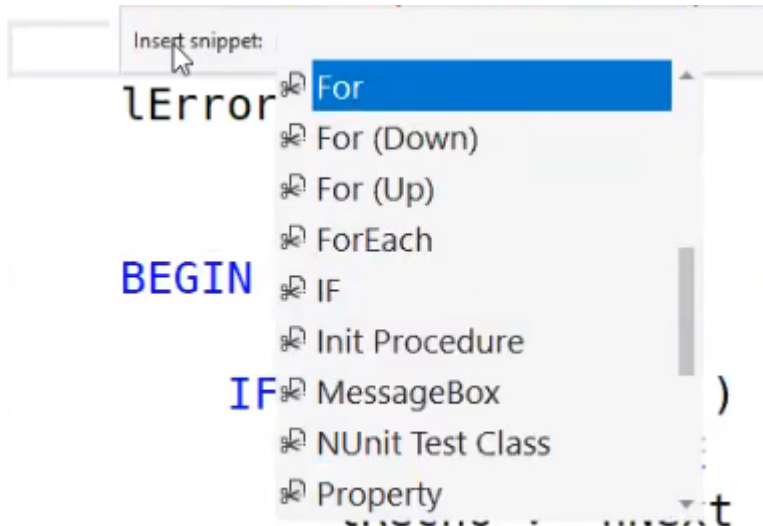
You can fill in the yellow sections, press tab to switch between these sections.

You can look at the available code snippets for X# by choosing Edit > IntelliSense > Insert Snippet or right-click > Snippet > Insert Snippet or pressing Ctrl+K, Ctrl+X.
The list includes snippets for creating a class, a constructor, a for loop, an if or switch statement, and more.

1.6.2.18 .EditorConfig files

Since X# 2.8 we now support the use of .editorconfig files. These files can be used in your solution or project to control the source code editor settings for several options, such as: This allows you to synchronize the editor settings for a team and ignore individual differences between team members.

- use tab or space
- tab with
- encoding for the source files

We are supporting the default tokens for .editorconfig as listed on https://editorconfig.org/:
- indent_style (tab or space)
- indent_size (number)
- tab_width (number)
- end_of_line (cr, lf or crlf)
- charset (latin1, utf-8, utf-8-bom, utf-16be or utf-16le)
- trim_trailing_whitespace (true or false)
- insert_final_newline (true or false)

Additionally we have added a few X# specific options

- keyword_case (upper, lower, title or none)
- identifier_case (true or false)
- indent_namespace (true or false)
- indent_type_members (true or false)
- indent_type_fields (true or false)
- indent_entity_content (true or false)
- indent_block_content (true or false)
- indent_case_label (true or false)
- indent_case_content (true or false)
- indent_continued_lines (true or false)
- indent_preprocessor (true or false)

The settings inside the .editorconfig overrule the settings on Tools/Options for the X# editor.

1.6.3   Debugger

The debugger inside Visual Studio is language agnostic. We have added support for our X# language, so you will see variables in the locals and autos windows with X# specific types and so you can enter expressions for breakpoint conditions, in the watch window and in the intermediate window in X# (case insensitive for example).

Behind the debugger expression evaluator (that's how this is called) is the complete X# compiler.

We have added an options page to Visual Studio from where you can control how the X# debugger Expression evaluator parses and compiles expressions.

Normally the debugger works with the settings from the main application.

However, when the startup application is not a X# application then the settings from this dialog are used.

The first 2 options work always.

When your application uses late binding and you see problems with late binding in the debugger, then you can disable late binding from this dialog.
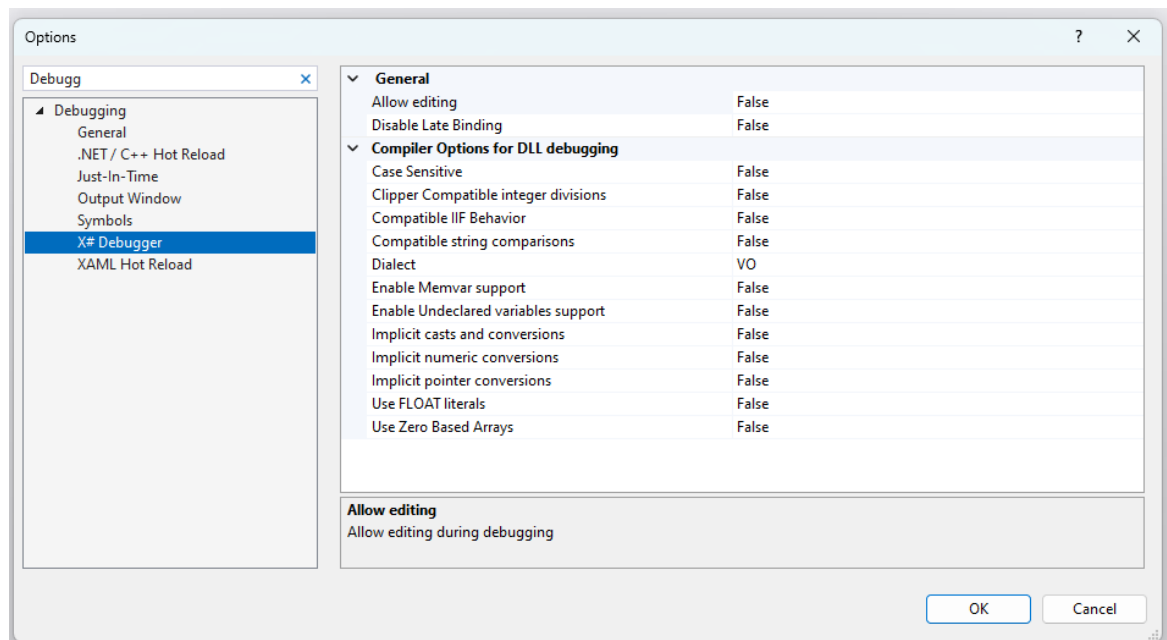
1.6.3.1   Toolbar and Menu

When the Visual Studio integration for XSharp is installed then you will have see new menu entry when debugging. This menu entry can be found in the Debug menu and is called "XSharp".

From that menu you can open 4 different windows with XSharp specific information.

- Globals
- Publics and Privates
- Workareas
- Settings

These windows are shown as pane windows in the lower half of the debugger.

The location of these windows are (initially) associated with existing Visual Studio windows in the following way

Window	Associated with
Globals	Watch window
Publics and Privates	Output window
Workareas	Autos window
Settings	Callstack window

If you move these windows to another location then Visual Studio will remember that new location

There is also an X# Debugger toolbar with buttons for these windows
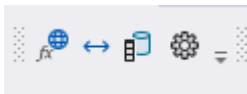
1.6.3.2 Globals Window

On the Globals window you can see a global variables in your app and the referenced class libraries that are loaded from your app.
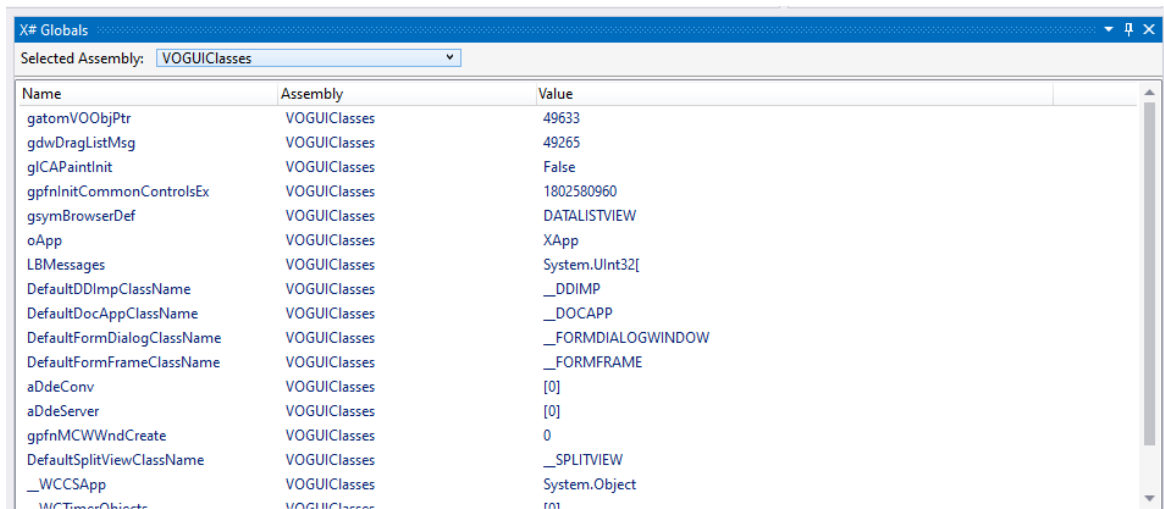
When your app does not include the X# runtime then a message will be shown that the runtime has not been loaded:
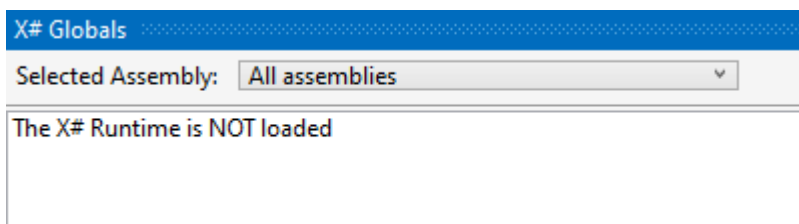
1.6.3.3   Publics and Privates Window

On this window you will see the publics and privates that are available at the current line of your source code:

You can use the checkboxes on the top to filter which type of variables you want to display.

When your app does not include the X# runtime then a message will be shown that the runtime has not been loaded:

1.6.3.4   Workareas window

On this window you will see the workareas are open at the current line of your source code, for the current thread.
The currently selected areas is marked with a checkbox.

In the future we may extend this window so you can see the properties of an area (such as BoF(), EoF() and RecNo()) and/or the field names and their values for the current record.

When your app does not include the X# runtime then a message will be shown that the runtime has not been loaded:

```
X# Workareas
Selected RDD:  All RDDs                    ▼
The X# Runtime is NOT loaded
```
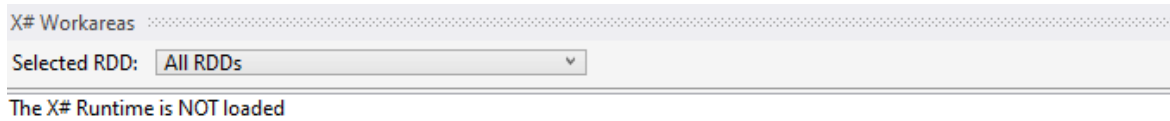
1.6.3.5 Settings Window

On this window you will see the various settings at the current line of your source code, and for the current thread.
You can sort these by name of by number.
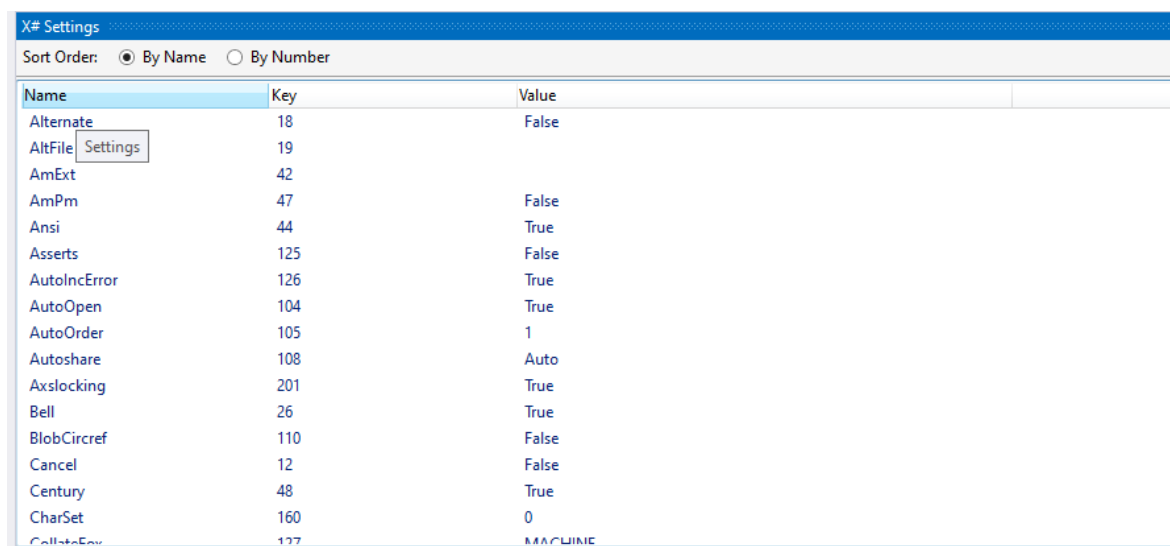
Name	Key	Value
Alternate	18	False
AltFile Settings	19	
AmExt	42	
AmPm	47	False
Ansi	44	True
Asserts	125	False
AutoIncError	126	True
AutoOpen	104	True
AutoOrder	105	1
Autoshare	108	Auto
Axslocking	201	True
Bell	26	True
BlobCircref	110	False
Cancel	12	False
Century	48	True
CharSet	160	0
CollateFox	127	MACHINE

When your app does not include the X# runtime then a message will be shown that the runtime has not been loaded:

```
X# Settings
Sort Order:    ● By Name   ○ By Number
The X# Runtime is NOT loaded
```
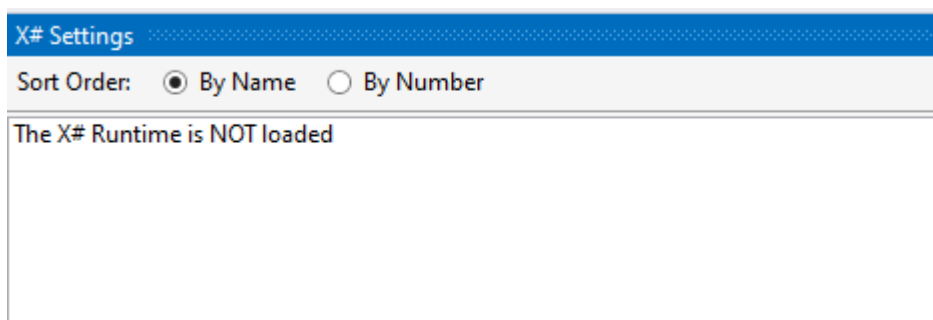
1.6.4 Other editors

We have added some custom editor to Visual Studio to allow you to edit resources that are migrated from Visual Objects.

- Form Editor
- DbServer Editor
- FieldSpec Editor
- Menu Editor

The Tools/Options dialog contains some settings that are used by these custom editors:
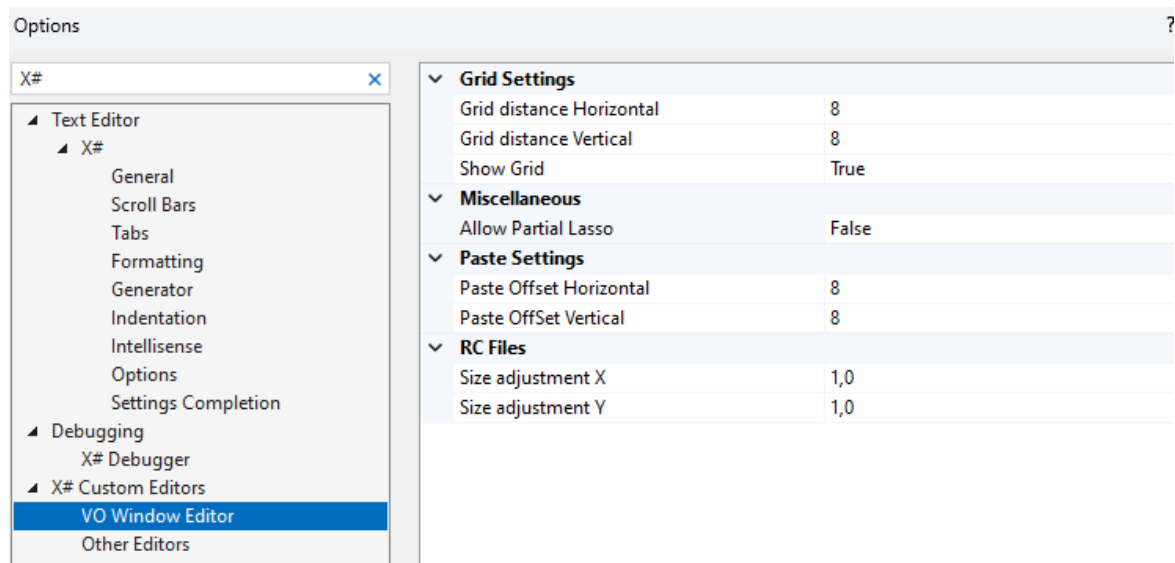
There is a second page with some settings for the other editor and for some other relevant settings:
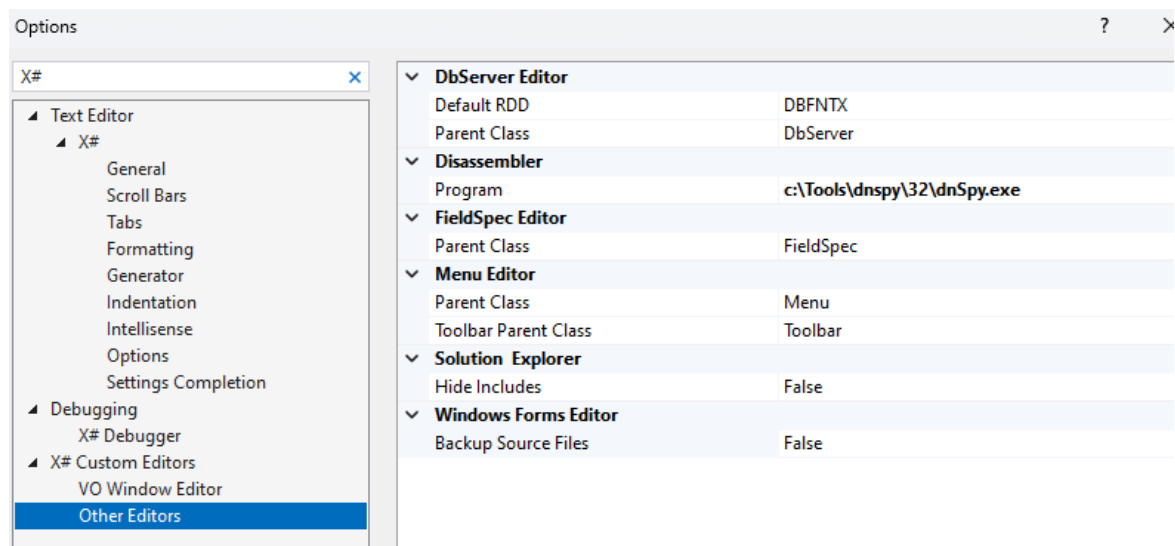
The values for the settings are stored in the file XSharp\ProjectSystemSettings.json in your roaming profile folder.

1.6.5 Templates

The Visual Studio integrations comes with a couple of predefined templates.
There are templates for:
- Projects
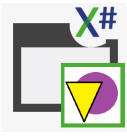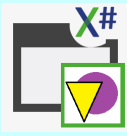- Items

1.6.5.1 Project Templates

The X# Visual Studio integration comes with the following project templates:

	Template	Description	Required Framework version	Default Dialect	References
	ClassLibrary	A project for creating a X# class library (.dll) for the Core dialect with no dependencies on the runtime	2.0	Core	System, and some extensions for frameworks > 3.5
	Core ClassLibrary	A project for creating a X# class library (.dll) for the Core dialect with a dependency on XSharp.Core	2.0	Core	System, X# Core and some extensions for frameworks > 3.5
	FoxPro ClassLibrary	A project for creating a X# class library (.dll) for the FoxPro Dialect	2.0	FoxPro	System, X# Core, VFP and some extensions for frameworks > 3.5
	VO ClassLibrary	A project for creating a X# class library (.dll) for the VO Dialect	2.0	Visual Objects	System, X# Core, VO and some extensions for frameworks > 3.5

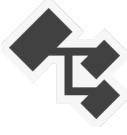|					
	Vulcan ClassLibrary	A project for creating a X# class library (.dll) for the Vulcan Dialect	2.0	Vulcan	System, X# Core, Vulcan and some extensions for frameworks > 3.5
	XPP ClassLibrary	A project for creating a X# class library (.dll) for the Xbase++ Dialect	2.0	Xbase++	System, X# Core, XPP and some extensions for frameworks > 3.5
	Console Application	A project for creating a X# command-line application in the core dialect.	2.0	Core	System, and some extensions for frameworks > 3.5
	FoxPro Console Application	A project for creating a X# command-line application in the FoxPro Dialect.	2.0	FoxPro	System, VFP, and some extensions for frameworks > 3.5
	VO Console Application	A project for creating a X# command-line application in the VO Dialect.	2.0	Visual Objects	System, VO, and some extensions for frameworks > 3.5
	Vulcan Console Application	A project for creating a X# command-line application in the Vulcan dialect with Vulcan	2.0	Vulcan	System, Vulcan, and some extensions for frameworks > 3.5

				Runtime Assemblies (BYOR = Bring Your Own Runtime).		
	XPP Console Application	A project for creating a X# class library (.dll) for the Xbase++ Dialect	2.0		Xbase++	System, XPP, and some extensions for frameworks > 3.5
	Windows Forms Application	A project for creating an application with a Windows Forms user interface.	2.0		Core	System, and some extensions for frameworks > 3.5 System.Drawing System.Windows.Forms
	WPF Application	Windows Presentation Foundation client application.	3.0		Core	System, and some extensions for frameworks > 3.5 WindowsBase, PresentationCore, PresentationFramework
	VO MDI Application	A project for creating a VO MDI application in the VO Dialect.	4.6		Visual Objects	System and some extensions for frameworks > 3.5
	VO SDI Application	A project for creating a VO SDI application in the VO Dialect.	4.6		Visual Objects	System and some extensions for frameworks > 3.5
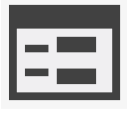
	Template	Description	Version	Type	Required Framework
	Vulcan Console Application	A project for creating a Vulcan command-line application.	2.0	Vulcan	System, and some extensions for frameworks > 3.5 VulcanRT and VulcanRTFuncs
	Ms Test Library	A Library with test support using the MsTest Framework	2.0	Core	System and several MsTest related assemblies
	NUnit Test Library	A library with test support using the open source NUnit Framework	2.0	Core	System and several NUnit related assemblies. The template uses a NuGet package to retrieve the correct assemblies
	XUnit Test Library	A library with test support using the open source XUnit Framework	2.0	Core	System and several XUnit related assemblies. The template uses a NuGet package to retrieve the correct assemblies

1.6.5.2 Item Templates

The X# Visual Studio integration comes with the following item templates:

Template	Description	Required Framework

			version
	Category: Code		
	Class	An empty class definition	2.0
	CodeFile	An empty Code File	2.0
	Header	An empty Header File	n/a
	TextFile	An empty text file	n/a
	Category: Forms		
	Windows Forms Form	A Windows form with separate designer.prg	2.0
	Windows Forms Simple Form	A simple Windows Forms Form without designer.prg	2.0
	Windows Forms User Control	Windows Forms User Control	2.0
	Category: Internal		
	AppConfigInternal	A hidden app.config	2.0
	AppManifestInternal	A hidden app.manifest	2.0
	AssemblyInfoInternal	A hidden assemblyinfo prg	2.0
	Managed Resource Internal	A hidden resource file	2.0
	Settings Internal	A hidden settings file	2.0

Category: Resources

	Native Resource File (.rc)	A file in which native resources can be defined	n/a
	Bitmap	A Bitmap file	n/a
	Cursor	A Cursor file	n/a
	Icon	An Icon file	n/a
	Managed resource file (.resx)	A file to store managed resources	2.0

Category: VO

	VODBServer	An empty DBServer	n/a
	VOFieldSpec	An empty FieldSpec container file	n/a
	VOMenu	An empty VO Menu	n/a
	VOWindow	An empty VO Window	n/a

Category: WCF

	WCF Service	A WCF Service	3.0

Category: WPF

	WPF Flow Document	Dynamically formatted XAML document	3.0
	WPF Page	Windows Presentation Foundation page	3.0

	WPF Page Function	Windows Presentation Foundation page function	3.0
	WPF Resource Dictionary	XAML Resource Dictionary	3.0
	WPF UserControl	Windows Presentation Foundation user control	3.0
	WPF Window	Windows Presentation Foundation window	3.0

1.6.6   VOXporter

Enter topic text here.

1.6.7 XPorter

Enter topic text here.

1.6.8 VFPXporter

The VFPXporter too is created to ease the move of your Visual FoxPro (VFP) Application in .NET, using a language you already know, with help of XSharp.
1. Minimum changes to your code during export
2. Keep the original code as comment if any change is made.
3. Move VFP Forms to Windows Forms, using an adaptation layer/library called XSharp.VFP.UI.Dll
4. Move VCX libraries as external projects containing Forms/Controls
5. Create real Windows Forms solution/project so you can use any existing 3rd party for .NET in your application

Start & Settings

First, open the Settings dialog box:

Folders settings

Exporter Settings ✕

Folders Export Project

Default VFP Items Folder

[   ] [...]

Default Output Folder

[|] [...]

Default XPorter ressources Folder

C:\Program Files (x86)\XSharp\VFPXPorter\.\Data [...]

[Reset Settings] [Open Settings Folder] [Cancel] [Ok]
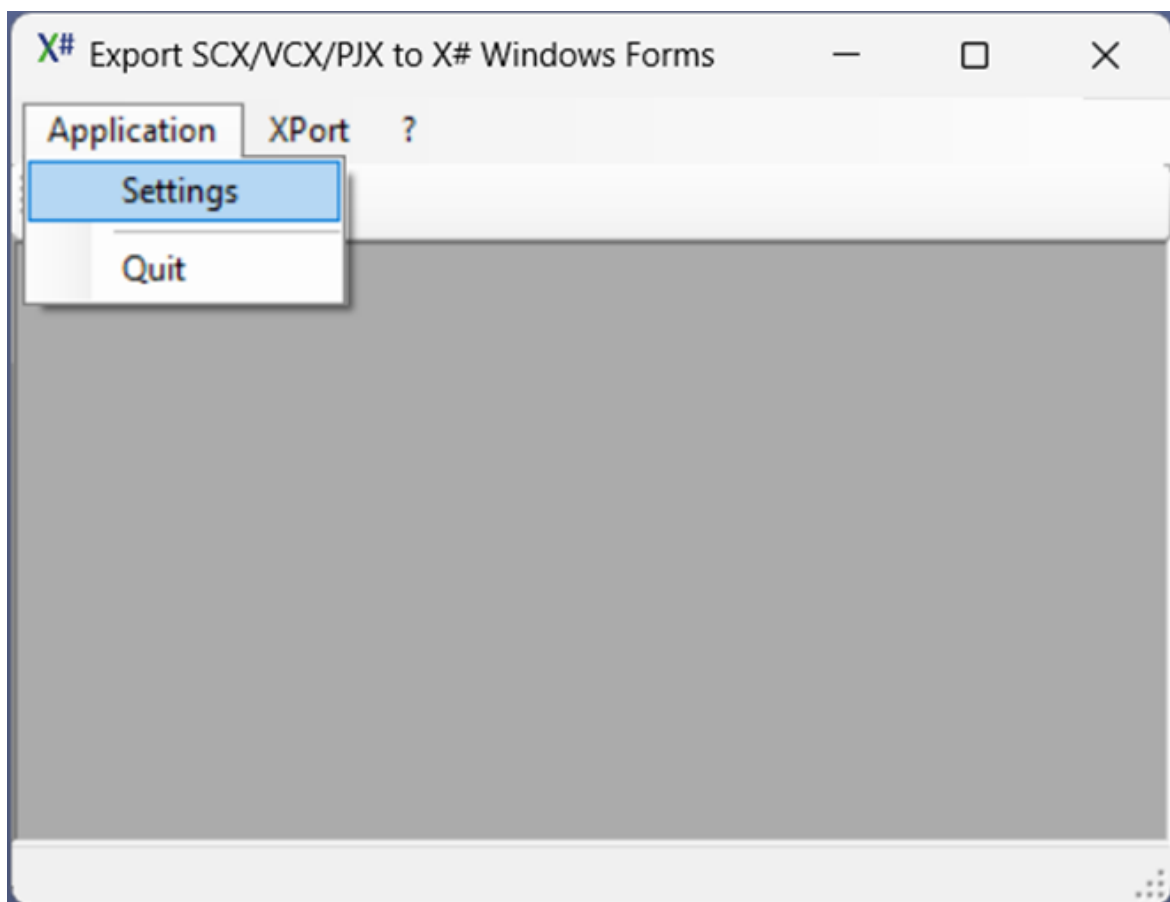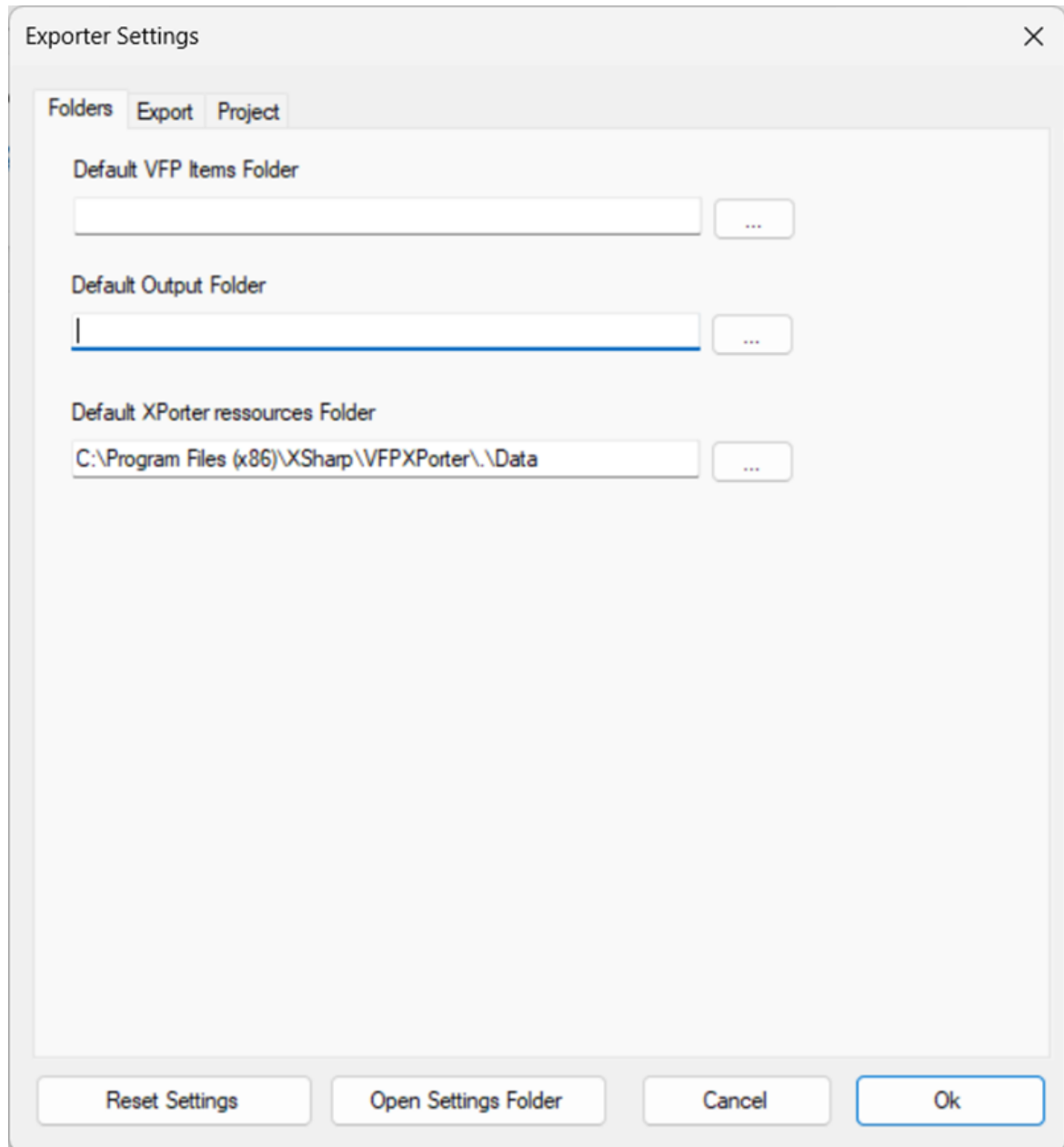
- Default VFP item Folder: The folder where VFPXPorter will start to search for elements to export per default.
- Default Output Folder: The folder where the VFPXPorter will create the generated export content
- Default exporter resources Folder: The folder where the VFPXPorter will look for extra templates/Libraries/Tools.etc

You can then:
- Reset Settings to their default values.
- Open Settings Folder where the .json settings file is stored (for backup for eg)

Export settings
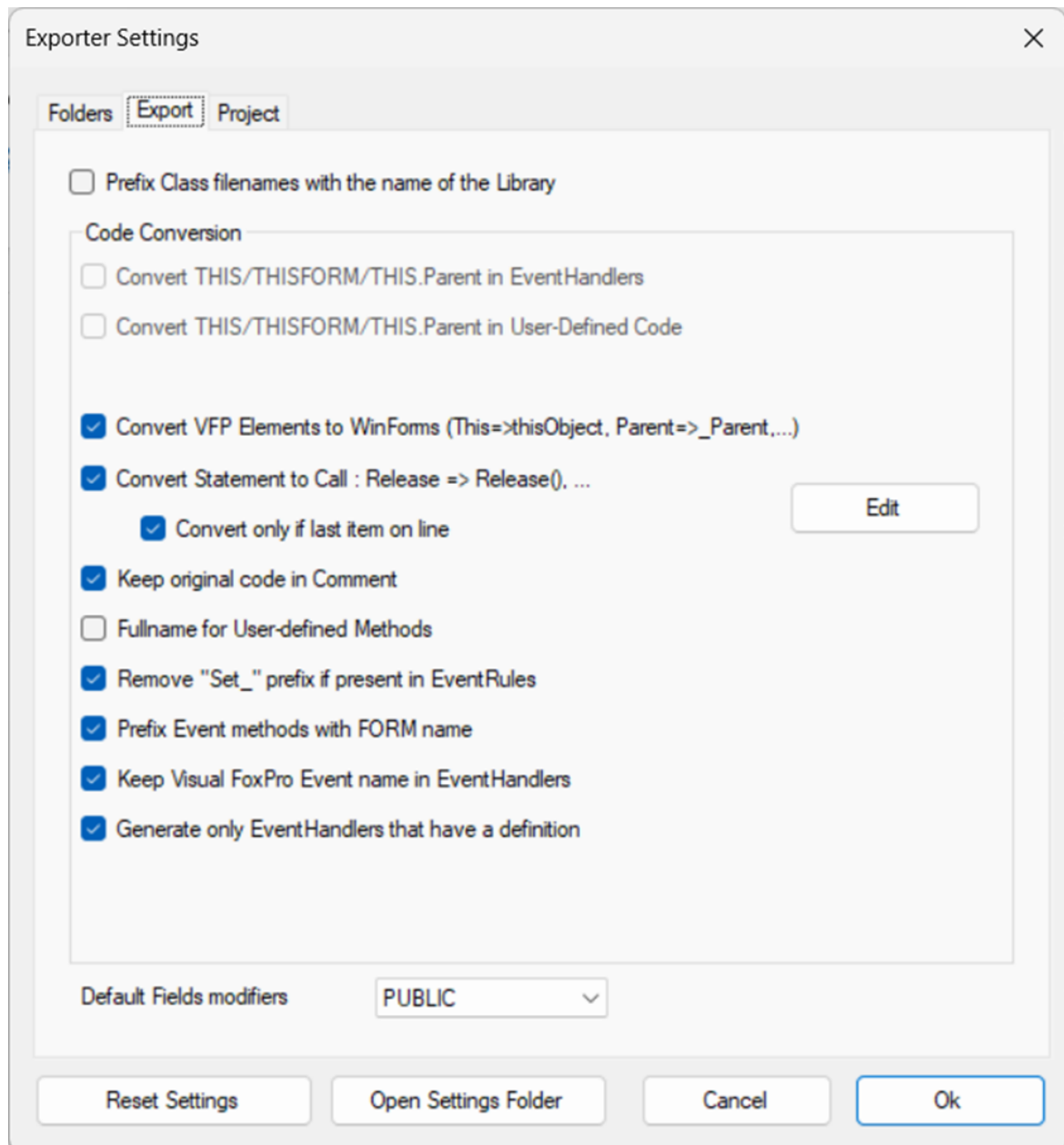
Check or Uncheck the export settings: Usually the default values are the best choice.

- Default Fields modifiers: When exporting a CLASS, set the default Field visibility.

Project settings

Exporter Settings   ✕

Folders Export **Project**

☑ Ignore export Errors

☑ Store Items in Folders by Items type   [<<]

☑ Each ClassLibrary is in a SubFolder

☑ Empty the destination Folder (if already exist)

Folder Name	Item Type	
Forms	Forms	
Libs	Libs	
Menus	Menus	
Code	Code	
Databases	Databases	
FreeTables	FreeTables	
Others	Others	

[Reset Settings]   [Open Settings Folder]   [Cancel]   [Ok]
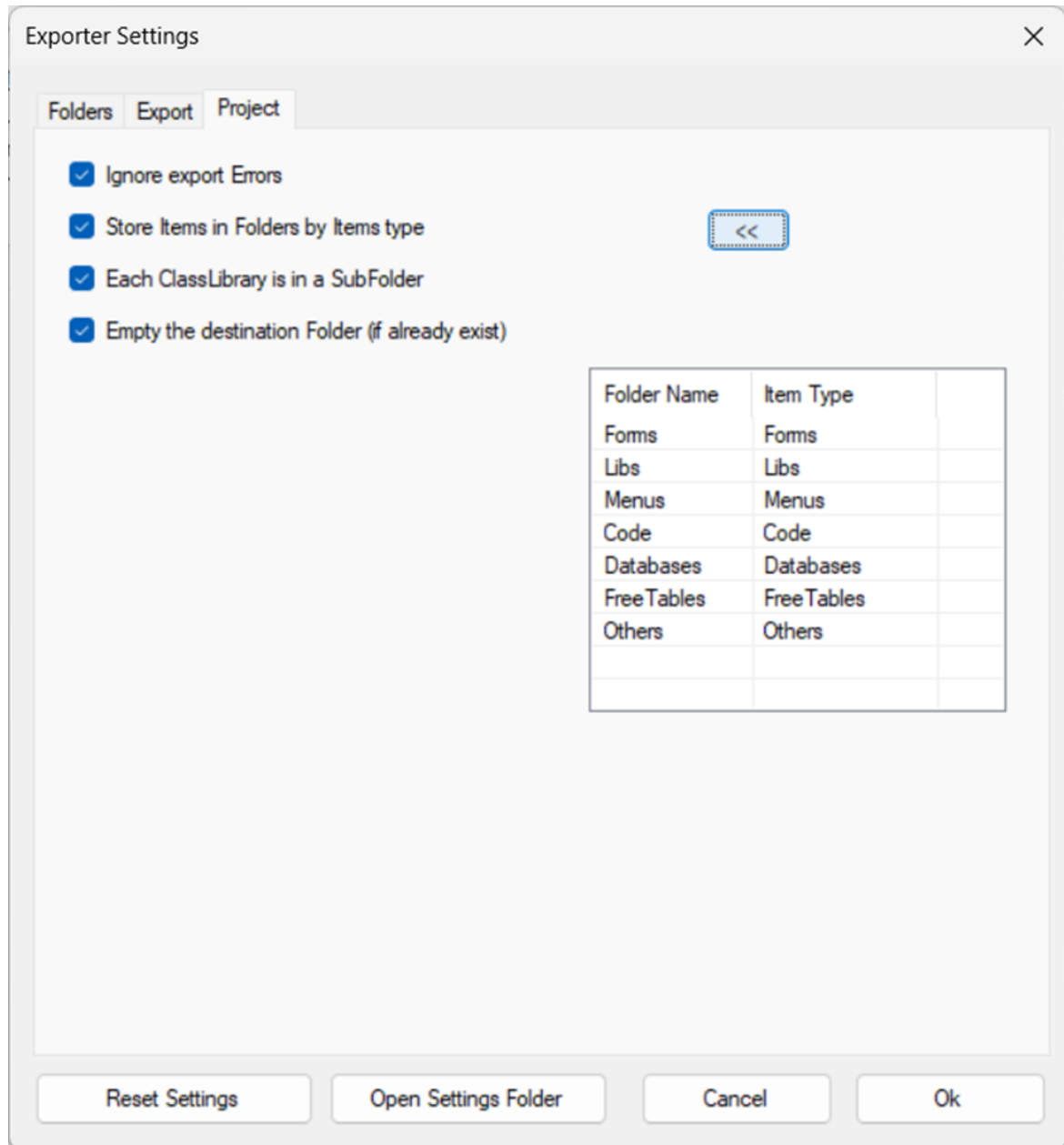
- Ignore export Errors: Allows to continue the export process in case of errors. All errors are logged in the destination folder.
- If needed, you can specify your own name for the Folders where elements are exported.
- Each ClassLibrary is in a SubFolder: If your application is using libraries, it is a good practice to export them separately.
- Empty the destination folder: Each time you export an element, the destination folder is emptied

Exporting

Project
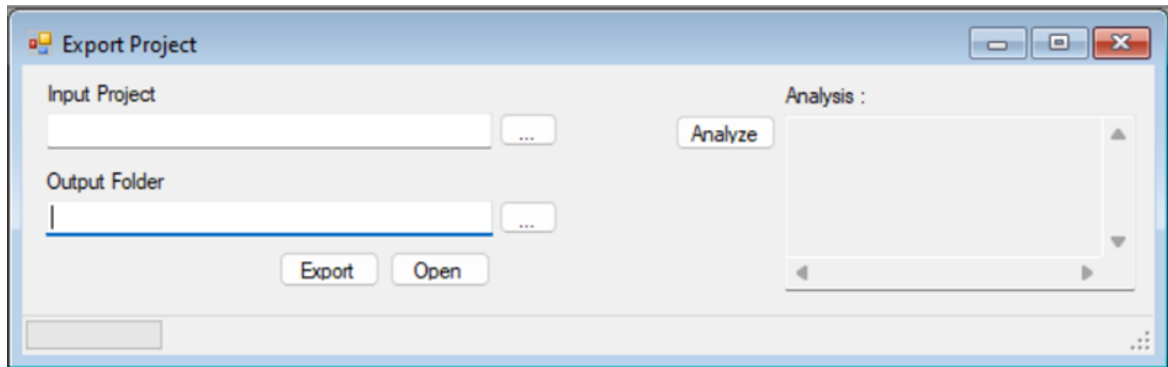
- Input Project: Search for the .PJX file to export.
- Output Folder: Indicate where the VFPXPorter will write the generated files.
- Analyze: Allow to look at the content that the VFPXPorter has detected in the Project.
- Export: Export the project
- Open : Open the folder where the files where written

The generated folder should contain several folders (One per Library if you have checked the setting, One for detected FreeTables, One for XSharp Tools, …)

You will find a file with the same name as the exported project with .SLN as extension.
This is the MS Visual Studio solution file that will group all exported top-level items (Project, Libraries, ...)
This is the file you should open first.

How does the VFPXPorter work?

The VFPXPorter will read you project and its elements. It will identify their type and generate some X# code.
In order to generate that code, it will look into the Data folder.
There, we will find several .JSON files that are used as conversion Rules:
The documentation for these rules is available in VFPXPorter repository on GitHub.
Then, to generate the files, it will use some templates : One per type of file. You can find these templates on GitHub : Templates

Converting from VFP

Common problems

Please find here a list of the common troubles we will face when converting your VFP GUI code to X# and WinForms.

Events & Methods naming

When converting your Application from VFP to X# & WinForms, you have to keep in mind that these two "worlds" are using a different point-of view to your Form.

In VFP, **events** are routed to the **Controls**.
So we could consider that, for eg, each button on the Form has its own class that INHERIT from the base-class CommandButton, and is an instance of that particular class: So you can override Events and Methods that that level, and it gives you a lot of control on what's happening there.
In *WinForms*, **events** are routed to the Owner of the Controls: The **Form**
So, the usage is to prefix the EventHandlers and Methods with the name of the Control.
If *button1* is raising a click event, the name of the handler is *button1_click*.
If a *Form* is raising a event, as the Form is the class, the name will be *nameOfTheForm_Event*.
You can change that via an exporter setting:

• Prefix Event methods with FORM name

Start() Function

The generated code contains a Start() Function, which is the Startup Object of the generated application. If you are also using a Start() Function in your VFP Code, you will have to rename it after export.

Init() Method

With VFP, the Init() method is called when an object is created.
With X#, calling the Constructor() of a class (with the MyClass{} syntax), won't automatically called the Init() Method.
This process is emulated by the *InitType.prg* template that is used during export.
With Controls, the Init() Method is called when the Handle is created. This code is generated by the *InitType.prg* template used during export.

Colors

Unfortunately, these cannot be handled automatically as it would break the Windows Forms designer in Visual Studio.
Setting a BackColor or ForeColor with an RGB, like :

```
this.Label1.ForeColor = RGB(255,0,0)
```

Should be converted to :

```
this.Label1.ForeColor =
System.Drawing.Color.FromARGB( 255,0,0) )
```

BUT.... If you had a function called RGB() that returns a System.Drawing.Color in a PRG file, in the Tools folder of the Exporter, it will be part of your Exported Solution, and it will supersede the XSharp Runtime definition, and that will do the trick !!

Method Call without parenthesis

In *X#*, a Property holds a Value that is set/get with the equal sign, while a Method contains code that is executed when called : And you can make the difference because Method call ends with parenthesis.
While this is allowed in VFP, you will have to correct that in *X#*

```
this.Refresh
```

Should be corrected as

```
this.Refresh()
```

These can be corrected at export time, if you check the "Convert Statement to Call" in settings.
The handled statements are in the *Statements.json* file

Form Properties casing

In order to modify a Form with the *Windows Form Designer* in Visual Studio, the Properties must respect the WinForm casing. So, you may have to add some properties to the *PropRules.json* in order to automatically convert them.
That's why *ShowInTaskBar* appears in the rules as *ShowInTaskbar*. (Note the lowercased b)

The following points have been addressed with the VFP2WinForms.json Rules

Access to Parent

The Parent Property is already defined in Windows Forms, and is strongly typed to a System.Windows.Forms.Control object. To avoid troubles, and ease the port of applications, the support library contains a property called _Parent that access the Control's parent, but as a weakly typed Object.
This will forced X# to a late-bound call, and could help in resolving access to ported code.
The call to _Parent is automatically generated by the XPorter.

Calling an Object Event in Your Code

In object with Parent, you may sometimes call directly Event handlers in your VFP code.
This might be problematic, and for example a direct call to the Click event of a button will fail.

```
This.Parent.cmd_prec.Click
```

During export, the This will be turned to ThisObject, the parent.

Array access

In *X#*, array elements are accessed using brackets [].
In VFP, it is possible to use parenthesis ().
But this can be a problem when accessing a .Net typed-array.

So, in order to ease the port of VFP applications, some array access have been turned to method call in the XSharp.VFP.UI, the Control layer that ease access to .NET Controls with VFP syntax.

```
x = __screen.FormCount
__screen.Forms(x-1).Release() // Here, Forms array will be
replaced by Forms Method call.
```

Access Fields in the current Area/Cursor

In VFP, it is possible to reference the current WorkArea/Cursor using its name, and to point to specific fields by specifying their name after a Dot selector, like with :

```
? movie.title
```

Unfortunately, in X# the Dot selector is used for Objects, so you will have to use an arrow as selector, like with:

```
? movie->title
```

Undeclared Vars

You can use an undeclared var anywhere in your code. This is something that X# will recognize and support.
In such case, X# will silently create a MemVar: a local, untyped, var.
But there are some cases, where this will not work. For eg:

```
IF SQLExec(nHandle,"SELECT *  FROM employees ORDER BY id ASC
LIMIT 1","CurResult") <=0 THEN &&& validate error first.
   Aerror(laErr)
   Messagebox("Check the following error: " + Chr(13) +
laErr[2],'Alert',3000)
   RETURN .F.
ENDIF
```

In the previous code, the var laErr has never been declared at this level, but it is used as a REF var by the AError() function.
This type of construction is not supported by X#. You will have to create a local variable laErr first.

Unsupported Functions

When building your VFP App, you may call some functions that are not (yet) supported by X#.
The compiler will express a warning (**XS0618**), but will generate an Exe, that will certainly crash at Runtime.
To avoid that, indicate these warnings as Errors.
To do so, with MS Visual Studio, follow these steps:
• Open the Properties of the Project

X#

- Go To Build
- In section Treat Warning as Errors, enter the Specific Warning : XS0618

XSharp.VFP.UI.Dll : adaptation layer

VFP GUI Control library is very different from .NET Windows Forms : Properties, Methods and Event Handling are very different.
In order to offer GUI export, VFPXPorter is using an adaptation layer : XSharp.VFP.GUI
That library contains some controls that micmics the way VFP is behaving. It exposes the originals properties of VFP and convert them to their .NET Windows Forms counterparts.

Some controls might be missing, same for some Properties/Methods/Events : Please report your troubles in the XSharp Forums.

1.6.9 UDC Tester

Enter topic text here.

1.7 X# Programming guide

1.7.1   Classes and Structures

In .Net, there are two kind of types that hold data, reference types (or classes) and value types (or structures). The two of them have different semantics in the way they are used, but both can contain themselves other reference and/or value types.

Classes

A class (or a **reference type)**, is the most common way of storing data. Its name comes from the fact that a variable of a reference type (commonly referred to as an **instance** of the type) does not contain the data directly, but instead points to (references) a memory location where the actual data is stored. A class in X# is defined through the CLASS...END CLASS statement and can INHERIT from another reference type, implement one or more INTERFACES and may contain fields, properties, constructors, methods, events and other items:

```
CLASS Customer
    EXPORT name AS INT // exported (public) field
    PROTECT age AS INT // protected field, not visible to code
outside the class
END CLASS
```

Typically classes and all their members are defined in a single file of code. If it's necessary for class members to be defined in multiple files (for example when the amount of class members is very big), then the class must be defined **in every file** as PARTIAL:

```
PARTIAL CLASS Customer
// class members
END CLASS
```

Since an instance of a class is only storing a pointer to the data, one or more variables can point to the exact same object in memory. Assigning a variable of a reference type to another variable of the same type results to both representing the same data. Updating data using one reference automatically updates the other reference as well:

```
FUNCTION Start() AS VOID
    LOCAL one, two AS Customer
    one := Customer{}
    two := one // now both vars point to the same object in memory
    two:name := "Robert"
    ? one:name // also "Robert"
```

Nested Classes

A class can even contain other types (classes or structures). In this case the type inside the main type is called a **nested type**. Nested types can be used by using the name of their container class and their own name, connected by a dot:

```
CLASS Customer
    CLASS NestedClass
        EXPORT FieldInNestedClass AS INT
    END CLASS
END CLASS

FUNCTION Start() AS VOID
    LOCAL oNested AS Customer.NestedClass
    oNested := Customer.NestedClass{}
    oNested:FieldInNestedClass := 100
    ? oNested:FieldInNestedClass
```

Nested classes are useful in particular for defining *helper* classes, a class that is only used in the context of the parent class, for holding information only relevant to that class. Creating a nested class for this data, instead of using a regular class results to better structured code.

Structures

A structure (or a **value type)**, in contrast to reference types, stores its data directly. It has some similarities to the Visual Objects STRUCTURE feature (renamed in X# to VOSTRUCT), but it is a lot more powerful than in VO, as it can contain most items that also reference types have, like properties, constructors, methods etc. Unlike reference types though, value types cannot inherit from other types or implement interfaces. They can contain nested classes or structures though. Value types can be defined with the STRUCTURE Statement:

```
STRUCTURE Vector2D
    EXPORT x AS INT
    EXPORT y AS INT
    METHOD Invert() AS VOID
        SELF:x := - SELF:x
        SELF:y := - SELF:y
END STRUCTURE
```

Since structures hold their data directly, instantiating them does not involve any additional memory consumption than the memory needed for their data itself (reference types need memory for the data and also for the pointer to the data) or garbage collector activity. They are mostly suitable as light weight data containers, usually hold a small amount of fields, typically 2-4, but can even contain a single element, like the System.Int32 (INT) or System.Boolean (LOGIC) data types which simply define the INT and LOGIC data types, including several methods for manipulating their data. Other very commonly used system defined structures are System.Drawing.Point, System.Drawing.Rectangle etc, all containing a small amount of data fields.

Structure semantics

Structures also have different semantics when using them compared to regular classes. It is not necessary to instantiate such a variable to use it, since declaring a var of a value type results to its data being allocated directly:

```
FUNCTION Start() AS VOID
    LOCAL vector AS Vector2D
    vector:x := 10
    vector:y := 20
```

Although, for convenience, it's possible to also define constructors in value types and instantiate them as with regular classes:

```
STRUCTURE Vector2D
    EXPORT x AS INT
    EXPORT y AS INT
    CONSTRUCTOR(vec_x AS INT, vec_y AS INT)
        SELF:x := vec_x
        SELF:y := vec_y
END STRUCTURE

FUNCTION Start() AS VOID
    LOCAL vector AS Vector2D
    vector := Vector2D{10,20}
    ? vector:x // 10
```

The most important difference that must always be taken under consideration, is that assigning a value type to another one results to the data of the source being **copied** to the destination, so unlike what happens with reference types, the data of the two variables are stored in separate memory locations and any changes to one variable will not affect the other:

```
FUNCTION Start() AS VOID
    LOCAL vec_1,vec_2 AS Vector2D
    vec_1:x := 10 ; vec_1:y := 20
    vec_2 := vec_1
    ? vec_2:x // 10, value was copied from first vector

    vec_2:x := 40 // put a new value to second vector
    ? vec_1:x // 10 again, first vector value still has its
original value
```

For this reason structures are not suitable for very large objects, since assigning one to another or passing one as an argument to a method involves copying all data from the source to the destination. On the other hand, with regular classes, only the pointer to the data is passed as an argument to a method.

Equals operator

Another important difference between reference and value types, is the behavior of the equals operator (==). For reference types, the equals operator between two variables only compares the pointers themselves, not the data of the objects. So it returns TRUE only when both variables point to the same object and in all other cases it returns FALSE, even if the data both objects contain is the same:

```
CLASS ReferenceType
    EXPORT data AS STRING
END CLASS

FUNCTION Start() AS VOID
    LOCAL o1,o2 AS ReferenceType
    o1 := ReferenceType{}
    o1:data := "test"

    o2 := ReferenceType{}
    o2:data := "test"

    ? o1 == o2 // FALSE, because o1 and o2 point to different
memory locations

    o2 := o1
    ? o1 == o2 // TRUE
```

On the other hand, by default the == operator cannot be used on structures and the compiler will report an error if you try to do so. It can be made possible to use it though, by defining an OPERATOR method in the structure that implements how the comparison with == should be done. In the sample below, the == operator is implemented to compare the actual data that the two compared structures hold, so that it returns TRUE, when the data is equal:

```
STRUCTURE ValueType
    EXPORT data AS STRING
    OPERATOR == (a AS ValueType, b AS ValueType) AS LOGIC
    RETURN a:data == b:data // let the equals == operator return
true when the data of the two arguments is the same
END STRUCTURE


FUNCTION Start() AS VOID
    LOCAL o1,o2 AS ValueType
    o1:data := "test"
    o2:data := "nothing"
    ? o1 == o2 // FALSE
```

```
    o2:data := "test"
    ? o1 == o2 // TRUE
```

Note that it is possible to compare values of most common system defined structures like System.Int32, System.Boolean, System.Double, because they also have defined equals operator methods like the one in the code above.

Which one to use

Weather to use a class or a structure for holding data depends on the specific needs related to the particular data. For data holding a lot of information (for example a customer object) you would typically use a reference type, as such objects usually don't get instantiated very often, but usually "live" long for the duration of the program. For smaller objects, that are being a created, manipulated and copied between variables a lot of times and in particular in tight loops (like for example an object representing a Complex number, consisting of a real and an imaginary part, which can be used in a lot of calculations), it is' more suitable to use a structure, as this will typically lead to faster execution, with a lot less memory consumption and garbage collector activity. In any case, it's very important to carefully consider their differences in semantics when using value vs reference types.

1.7.2 Codeblock, Lambda and Anonymous Method Expressions

X# contains 3 language constructs that are fairly similar yet different.

Codeblocks

Historically the XBase language has known the CodeBlock Type. A codeblock is specified as

```
{| params | expression }
```

The parameters inside a codeblock are of type USUAL and its return value is also of type USUAL.
To evaluate a codeblock you call the Eval() runtime function and pass this function the codeblock and parameters when needed

```
FUNCTION Start() AS VOID
LOCAL cb as CODEBLOCK
cb := { |x, y| x * y}
? Eval(cb, 2,3)  // shows 6
WAIT
RETURN
```

In stead of a single expression, you can also use an expression list. The value of the last expression in the list is returned as the result of the codeblock.

```
FUNCTION Start() AS VOID
LOCAL cb as CodeBlock
cb := { |x, y| x += 1, x * y}
? Eval(cb, 2,3)  // shows 9
WAIT
RETURN
```

X# has also added the possibility to the language to include a list of statements as "body" of the codeblock. The result of the last statement is returned to the calling code. When the last statement is "Void" then a NIL will be returned:
Note that the closing Curly must be on a line of its own and the first statement must appear on a new line as well.

```
FUNCTION Start() AS VOID
LOCAL cb as CodeBlock
```

```
cb := { |x, y|
        x += 1
          ? x
     RETURN x * y
   }
? Eval(cb, 2,3)  // prints 3 and shows the result 9
WAIT
RETURN
```

Lambda Expressions

Lamda expressions look a lot like Codeblocks. They are usually used in combination with Delegates.

```
DELEGATE MultiPlyInt( x as Int, y as Int) as Int

FUNCTION Start() AS VOID
LOCAL lambda as MultiPlyInt
lambda := { x, y => x * y}
? Lambda(2,3) // shows 6
RETURN
```

Parameters are optional and the return type can be VOID, so this works as well

```
DELEGATE DoSomething( ) as VOID

FUNCTION Start() AS VOID
LOCAL lambda as DoSomething
lambda := { => Console.WriteLine("This is a Lambda")}
Lambda() // prints the text
RETURN
```

The parameters of a Lambda expression may be typed. This can be convenient for documentation purposes but can also help the compiler to find the right overload for a method:

```
DELEGATE CalcInt( x AS INT, y AS INT) AS INT
```

```
DELEGATE CalcReal( x AS REAL8, y AS REAL8) AS REAL8

FUNCTION Start() AS VOID
TestLambda( { x AS INT, y AS INT => x * y} )
TestLambda( { x AS REAL8, y AS REAL8 => x + y} )
TestLambda( { x , y  => x - y } ) // Which one will be called ?
RETURN

FUNCTION TestLambda (lambda AS CalcInt) AS VOID
? "Int", lambda(2,3)
RETURN

FUNCTION TestLambda (lambda AS CalcReal) AS VOID
? "Real",lambda(2,3)
RETURN
```

The body of the Lambda may also be a single expression, expression list and a statement list.

Anonymous Method Expressions

These work almost the same as Lambda Expressions.
Take the example below:

```
FUNCTION TestAnonymous() AS VOID
   LOCAL oForm AS Form
   oForm := Form{}
   oForm:Click += DELEGATE(o AS System.Object, e AS
System.EventArgs ) {
                System.Windows.Forms.MessageBox.Show("Click from
AME 1!")
                System.Windows.Forms.MessageBox.Show("Click from
AME 2!")
             }
   oForm:Click += { o,e =>
System.Windows.Forms.MessageBox.Show("We can also do this with a
Lambda!") }
   oForm:ShowDialog()
   RETURN
```

The biggest difference between Lambda Expressions and Anonymous Method Expressions is that the parameters to Lambda Expressions do not have to be typed. They will be inferred from the usage. Parameters for Anonymous Method Expressions must always be typed.

1.7.3 Exceptions and Exception Handling

In X#, errors in the program at run time are propagated through the program by using a mechanism called exceptions. Exceptions are thrown by code that encounters an error and caught by code that can correct the error. Exceptions can be thrown by the .NET Framework common language runtime (CLR) or by code in a program. Once an exception is thrown, it propagates up the call stack until a catch statement for the exception is found. Uncaught exceptions are handled by a generic exception handler provided by the system that displays a dialog box.

Exceptions are represented by classes derived from Exception. This class identifies the type of exception and contains properties that have details about the exception. Throwing an exception involves creating an instance of an exception-derived class, optionally configuring properties of the exception, and then throwing the object by using the throw keyword. For example:

```
CLASS CustomException INHERIT Exception

    CONSTRUCTOR(message AS STRING)
    SUPER(message)

END CLASS


FUNCTION TestThrow as VOID
LOCAL ex AS CustomException
ex := CustomException{"Custom exception in TestThrow()"}
THROW ex
RETURN
```

After an exception is thrown, the runtime checks the current statement to see whether it is within a try block. If it is, any catch blocks associated with the try block are checked to see whether they can catch the exception. Catch blocks typically specify exception types; if the type of the catch block is the same type as the exception, or a base class of the exception, the catch block can handle the method. For example:

```
FUNCTION TestCatch as VOID

    TRY
        TestThrow()
    CATCH ex AS CustomException
        System.Console.WriteLine(ex.ToString())
    END TRY
RETURN
```

If the statement that throws an exception is not within a try block or if the try block that encloses it has no matching catch block, the runtime checks the calling method for a try statement and catch blocks. The runtime continues up the calling stack, searching for a compatible catch block. After the catch block is found and executed, control is passed to the next statement after that catch block.

A try statement can contain more than one catch block. The first catch statement that can handle the exception is executed; any following catch statements, even if they are compatible, are ignored. Therefore, catch blocks should always be ordered from most specific (or most-derived) to least specific.

1.7.4 Memory Variables

X# provides support for dynamically scoped variables that are created and maintained completely at runtime.

The term dynamically scoped refers to the fact that the scope of these variables is not limited by the entity in which the variable is created.

Warning! Dynamically scoped variables are NOT supported in the Core and Vulcan dialects. In other dialects they are supported only if the -memvar compiler option is enabled.

Variable Type	Lifetime	Visibility
PRIVATE	Until creator returns or until released	Creator and called routines
PUBLIC	Application or until released	Application

The data type of a dynamically scoped variable changes according to the contents of the variable. For this reason they are often described as dynamic or polymorphic. Dynamically scoped variables are provided mainly for Clipper/Xbase compatibility; however, they are very useful in certain circumstances. For instance, they let you develop rapid prototypes and have certain inheritance properties that you may find hard to resist. You must be aware, however, that using them comes at a cost. Consider these points:

- Because they are not resolved at compile time, these variables require overhead in the form of runtime code, making your application larger and slower than necessary.
- No compile time checking for type compatibility is possible with these variables.
- Using the inheritance properties of these variables defies one of the basic tenets of modular programming and may lead to maintenance and debugging problems down the line. Furthermore, this practice will make the transition to lexically scoped and typed variables more difficult.

This section explores dynamically scoped variables fully, but X# has several options for variable declarations that you will want to explore before choosing to use this variable class. The next two sections in this chapter introduce you to Lexically Scoped Variables and Strongly Typed Variables, which you may find useful.

Important! For the sake of illustration, some of the examples in this section use unorthodox programming practices. Using the inheritance properties of public and private variables instead of passing arguments and returning values is not recommended.

Private

Private is one of the two types of dynamically scoped variables, and there are several ways to create a private variable:

- List the variable name as part of a PRIVATE statement. If you do not make an assignment at this time, the variable takes on the NIL value and data type; otherwise, it takes on the data type of its assigned value. You can assign a new value (with a new datatype) to a variable at any time:

```
PRIVATE X := 10, y
```

This creates x as a numeric variable and y as an untyped variable with the value NIL. You can later change the values and their types by assigning other values to them:

```
X := "X# Is great"
Y := Today()
```

- List the variable name as part of a PARAMETERS statement within a FUNCTION, PROCEDURE or METHOD definition. The variable takes on the data type of its

associated argument when the routine is called, or NIL if the argument is omitted. You can assign a new value (and a new data type) to the variable at any time.

- Assign a value to a non-existent variable name (for example, x := 10). The variable takes on the data type of its assigned value until you assign a new value to it. (x is numeric, but the assignment x := "Ms. Jones" changes it to a string.) This will only work if you have used the -underclared as well as the -memvar commandline options.

Private variables have these properties:

- You can access them within the creating routine and any routines called by the creator. In other words, private variables are automatically inherited by called routines without having to pass them as arguments.
- You can hide them from a called routine by explicitly creating a private (using PRIVATE or PARAMETERS) or declaring a local (using LOCAL) variable with the same name in the called routine.
- They are automatically released from memory when the creator returns to its calling routine, or you can release them explicitly using RELEASE, CLEAR ALL, or CLEAR MEMORY.

In this example, the function Volume() expects three arguments, or parameters, to be passed. When the function is called, it creates three private variables, nLength, nWidth, and nHeight to accept the arguments. Because they are created with the PARAMETERS statement, any higher-level variables (either public or private) created with these names are temporarily hidden, preventing their values from being overwritten in memory:

```
FUNCTION Volume()
    PARAMETERS nLength, nWidth, nHeight
    RETURN nLength * nWidth * nHeight
```

In the next example, a modified version of Volume() creates a private variable (assuming no other variable name nVolume is visible) to store its return value. If the variable nVolume exists prior to calling Volume() and is visible to Volume() (for example, nVolume may be public or private to the routine that called Volume()), its value is overwritten in memory and will remain changed when the function returns to its calling routine:

```
FUNCTION Volume()
    PARAMETERS nLength, nWidth, nHeight
    nVolume := nLength * nWidth * nHeight
    RETURN nVolume
```

In this version, Volume() specifies the nVolume variable as PRIVATE. Doing this temporarily hides any higher-level variable (either public or private) with the same name, preventing its value from being overwritten in memory:

```
FUNCTION Volume()
    PARAMETERS nLength, nWidth, nHeight
    PRIVATE nVolume := nLength * nWidth * nHeight
    RETURN nVolume
```

Public

The second category of undeclared variable is public. Public variables have application-wide lifetime and visibility, and you can define them in only one way:

- List the variable name as part of a PUBLIC statement. If you do not make an assignment at this time, the variable takes on a value of FALSE (or NIL for array elements); otherwise, it takes on the data type of its assigned value. You can assign a new value (and a new data type) to the variable at any time.

Public variables have these properties:

- Once they are created, you can access them anywhere in the application. In other words, public variables are automatically inherited by all routines in the application without having to pass them as arguments or post them as return values.
- You can hide them from a routine by explicitly creating a private (using PRIVATE or PARAMETERS) or declaring a local (using LOCAL) variable with the same name.
- They are not released from memory until you explicitly release them using RELEASE, CLEAR ALL, or CLEAR MEMORY.

In this example, the function Volume() is defined without arguments. Instead, the calling routine, Compute(), creates three public variables, nLength, nWidth, and nHeight that are automatically visible to Volume():

```
PROCEDURE Compute()
PUBLIC nLength := 5, nWidth := 2, nHeight := 4
? Volume() // Result: 40


FUNCTION Volume()
RETURN nLength * nWidth * nHeight
```

In the next example, a modified version of Volume() creates a public variable to store the computed volume, getting around having to return a value to the calling routine. Since nVolume is public, it is not released from memory when Volume() returns:

```
PROCEDURE Compute()
PUBLIC nLength := 5, nWidth := 2, nHeight := 4
Volume()
? nVolume // Result: 40 , this will only compile with -undeclared
RETURN

PROCEDURE Volume()
PUBLIC nVolume
nVolume := nLength * nWidth * nHeight
RETURN
```

A better solution for the use of the nVolume variable from last example that will not require the -undeclared commandline option is:

```
PROCEDURE Compute()
PUBLIC nLength := 5, nWidth := 2, nHeight := 4
```

```
MEMVAR nVolume     // tell the compiler that nVolume is a Public or
private
Volume()
? nVolume // Result: 40
RETURN
```

or

```
PROCEDURE Compute()
PUBLIC nLength := 5, nWidth := 2, nHeight := 4
Volume()
? _MEMVAR->nVolume // Result: 40
RETURN
```

Please note that this kind of programming is NOT recommended.

Variable References

Once a public or private variable is created as demonstrated in the previous two sections, you obtain its value by referring to its name. You might display the value of a variable using a built-in command or function:

```
? nVolume
QOut(nVolume)
```

or use its value as part of an expression:

```
Str(nVolume, 10, 2) + " cubic feet"
```

For dynamically scoped variables, you can use the _MEMVAR alias to qualify a variable reference. In some cases, you may have to do this in order to help the compiler resolve what might otherwise be an ambiguous reference (for example, if you have a field variable with the same name as a memory variable and want to use the memory variable in an expression).

Note: MEMVAR is an abbreviation for memory variable, a term that is synonymous with dynamically scoped variable.

Assuming that the database file Measures has fields named nLength, nWidth, and nHeight, this example calls Volume() using the field variable values:

```
FUNCTION Calculate()
PRIVATE nLength := 5, nWidth := 2, nHeight := 3
USE measures
? Volume(nLength, nWidth, nHeight)
...
```

To force the function to use the private variables instead of the field variables, you could use the _MEMVAR-> (or, more simply, M->) alias to qualify the variable names:

```
FUNCTION Calculate()
PRIVATE nLength := 5, nWidth := 2, nHeight := 3
USE measures
? Volume(_MEMVAR->nLength, _MEMVAR->nWidth, _MEMVAR->nHeight)
...
```

Of course, it is better to avoid ambiguous situations like the one described above by taking care to have unique field and variable names, but the point is that the compiler has certain default rules for handling ambiguous references. If you do not want to be at the mercy of those defaults, it is best to qualify variable names in all cases.

MEMVAR Declarations

Although you may hear them referred to as such, the statements mentioned so far in the discussion of dynamically scoped variables are not declarations. The term declaration refers to a statement whose purpose is to inform the compiler of something—PRIVATE, PARAMETERS, and PUBLIC are statements that generate memory variables at runtime. In fact you never have to declare a dynamically scoped variable to the compiler, which is the reason for their inefficiency. Because they are not created using compile-time declaration statements, the compiler has to generate runtime code for handling such issues as type translation, memory management, and resolving ambiguous references to variable names since it is possible for several variables with the same name to be visible at one time.

You can, however, declare dynamically scoped variables with the MEMVAR statement and they will be created as PRIVATE variables:

```
FUNCTION Calculate()
MEMVAR nLength, nWidth, nHeight
nLength := 5
nWidth := 2
nHeight := 3
USE measures
? Volume(nLength, nWidth, nHeight)
```

In this case, the MEMVAR statement causes memory variables to take precedence over field variables with the same names, causing Volume() to be called with the private variables.

Using MEMVAR to declare dynamically scoped variable names to the compiler may make your programs slightly more efficient (especially if you have lots of ambiguous references); however, it will not eliminate the runtime overhead of these variables.

1.7.5   Modifiers

Modifiers are language keywords that are used to modify the visibility of a program element (function, global, class, property etc), or the way they operate within a type hierarchy.

There are several groups of modifiers that will be discussed in these subtopics.

[Visibility Modifiers](#)	PUBLIC, EXPORT, PROTECT, PROTECTED, PRIVATE, HIDDEN, INTERNAL
[Class Hierarchy Modifiers](#)	VIRTUAL, OVERRIDE, ABSTRACT, SEALED
[STATIC modifier](#)	STATIC
[ASYNC / AWAIT](#)	ASYNC, AWAIT
[Other Modifiers](#)	PARTIAL, EXTERN, UNSAFE

1.7.5.1   Access/Visibility modifiers

Access / Visibility modifiers

Access modifiers restrict the visibility of program elements:

- PUBLIC/EXPORT: A public function, global, type (class, structure, delegate etc) or type member (field, method, property, event etc) is visible to all code within the same assembly (exe or dll) where it is declared and assemblies referenced by the declaring assembly. This is the default in X#.
- PROTECT/PROTECTED: The PROTECT(ED) keyword can be applied only to type members and restrict their visibility to code within the same type and classes that inherit from that type
- INSTANCE: A unique modifier keyword to X# (and Visual Objects), very similar to PROTECT, but applies only to class fields and has different semantics to PROTECT, on the way it is used by code within the same class (see sample below)
- PRIVATE/HIDDEN: Also applies to type members and is more restrictive than PROTECT, as it limits the member visibility to only code within the type where it is declared (not to classes inheriting from that type). If you declare a field or method as PRIVATE then the compiler may produce a compiler warning xs0169 when it detects that that field or method is never called inside the class.
- INTERNAL: Restricts the visibility of a type or member only to code from the same assembly (dll or exe). An internal member is not visible to any code outside the defining assembly, not even to classes inheriting from this class declared in other assemblies. Can be combined with PROTECTED, so that an INTERNAL PROTECTED member can be seen only form within the same class and also from inheriting classes that are defined in the same assembly, but not from inheriting classes that are defined in other assemblies.
- The **XBase**++ dialect also has the modifier STATIC which means that the class is only visible in the source file where it is used.

Note: It is still possible to explicitly specify certain assemblies where internal types and members of an assembly are visible, by using the InternalsVisibleTo attribute. See

documentation for System.Runtime.CompilerServices.InternalsVisibleToAttribute for more information.

```
INTERNAL PROCEDURE InternalProc()          // procedure
accessible only from within the same assembly
INTERNAL GLOBAL InternalGlobal AS INT       // global accessible
only from within the same assembly

PUBLIC CLASS TestClass                       // accessible from
everywhere
    EXPORT Export_field AS INT               // accessible from
everywhere
    PROTECT Protected_field AS INT           // accessible to this
class and classes inherited from it
    INTERNAL PROTECT Internal_Protected_field AS INT // accessible
to this class and classes inherited from it, defined in this same
assembly only
    PRIVATE METHOD Private_Method() AS VOID   // accessible only
from code inside this particular class
END CLASS
```

PROTECTED vs HIDDEN

A compatibility feature for code derived from Visual Objects, INSTANCE applies to fields and is similar to PROTECT, but has different semantics when a property (or ACCESS) is defined with the same name. When using a PROTECT field and an ACCESS/ASSIGN pair (or PROPERTY) is defined with the same name in the type, then any code within that class that tries to access this member, always refers to the field. But when the member is declared with the INSTANCE keyword, then the compiler binds the name to the ACCESS instead:

```
FUNCTION Start() AS VOID
    TestClass{}:DoTest()

CLASS TestClass
    PROTECT Protected_member := "field" AS STRING
    INSTANCE Instance_member := "field" AS STRING
    ACCESS Protected_member AS STRING
    RETURN "access"
    ACCESS Instance_member AS STRING
    RETURN "access"
    METHOD DoTest() AS VOID
        ? SELF:Protected_member // field
        ? SELF:Instance_member // access, because the field is
defined with the INSTANCE keyword
END CLASS
```

This feature is a left over from old Visual Objects versions that were emitting INSTANCE members for every control in the Window Editor and is depreciated in X#, but still available for compatibility reasons

1.7.5.2 Class hierarchy modifiers

Hierarchy modifiers control the way elements of a class behave within their class hierarchy.

VIRTUAL/OVERRIDE/NEW

By default, all methods in .Net are non-VIRTUAL, meaning they cannot be overridden by same named methods (and signature - parameter and return types) in descendant classes. In the following example, calling a method from code inside the Parent class, results to calling he version of the method defined in this particular class in the class hierarchy, not the method with the same name and signature defined in the Child class (even though the object we are testing with is an instance of the Child class):

```
FUNCTION Start() AS VOID
    LOCAL oTest AS Child
    oTest := Child{}
    oTest:DoTest()

CLASS Parent
    METHOD SomeMethod() AS VOID
        ? "Parent method was called"
    METHOD DoTest() AS VOID
        SELF:SomeMethod() // calls the method of this Parent class,
not the same named one defined in the Child class
    RETURN
END CLASS


CLASS Child INHERIT Parent
    METHOD SomeMethod() AS VOID
        ? "Child method was called"
END CLASS
```

This behavior is different to that of Visual Objects, FoxPro and Xbase++, where ALL methods are always considered VIRTUAL. In order to make a method overridable from descendant classes, it needs to be defined as VIRTUAL. In the following code, calling a VIRTUAL method from code in the Parent class, results to calling the version of the method defined in the Child class, since it has overridden the parent method:

```
CLASS Parent
    METHOD NonVirtualMethod() AS VOID // cannot be overriden in
child class
        ? "Parent non virtual method was called"
    VIRTUAL METHOD VirtualMethod() AS VOID // can be overriden
        ? "Parent virtual method was called"
```

```
    METHOD DoTest() AS VOID
        SELF:NonVirtualMethod() // calls method in parent
        SELF:VirtualMethod() // calls method in child
    RETURN
END CLASS

CLASS Child INHERIT Parent
    METHOD NonVirtualMethod() AS VOID // this is completely
different method to the parent one, even though it has the same
name
        ? "Child non virtual method was called"
    OVERRIDE METHOD VirtualMethod() AS VOID // overrides the same
named method of the parent class
        ? "Child virtual method was called"
END CLASS
```

Note that VirtualMethod() in the child class is declared with the OVERRIDE modifier, this tells the compiler that we have on purpose overridden a parent method with the same name. The OVERRIDE modifier is not mandatory in X# and can be omitted, but using it makes the code more self-explanatory and allows the compiler to make additional compile time checks. If OVERRIDE is used on a method to override a parent method that is not VIRTUAL, or it does not even exist (or is spelled with a different name or has a different signature in the parent class), then a compiler error will be reported. For this reason, it is recommended to explicitly declare methods overriding parent methods with the OVERRIDE keyword. It is also possible to enforce the use of the OVERRIDE keyword in the compiler, by enabling the compiler option "Enforce OVERRIDE" (/enforceoverride).

In the above code, the compiler does report a warning on the NonVirtualMethod() defined in the child class: "warning XS0108: 'Child.NonVirtualMethod()' hides inherited member 'Parent.NonVirtualMethod()', use the new keyword if hiding was intended". This warns about the definition of a method in the child class that has the same name with a non-virtual method in the parent class, as this could had been done by accident (either accidentally using the same name, or forgetting to define the base method as virtual). In order to tell the compiler that the use of the same name in a child method was intentional (and prevent the warning), the NEW modifier keyword can be used, which explicitly marks the method as one that does override the base one:

```
CLASS Child INHERIT Parent
    NEW METHOD NonVirtualMethod() AS VOID // explicitly mark the
method as new one, different to the parent method
        ? "Child non virtual method was called"
    OVERRIDE METHOD VirtualMethod() AS VOID // overrides the same
named method of the parent class
        ? "Child virtual method was called"
END CLASS
```

For already existing code ported from older systems like Visual Objects where methods are always VIRTUAL, it can be tiresome to manually add the VIRTUAL modifier keyword in all methods that need it. For this reason, it is possible for convenience to instruct the

compiler to automatically treat ALL methods as virtual, with the use of the "All instance methods virtual" (/vo3) compiler option. But is highly recommended to instead review the code and manually add the VIRTUAL modifier only where it is really needed.

Note that a method in a child class can override a parent class method, only if it has the exact same signature with it, meaning it has the exact same name (even the exact same casing of the name, if the Case sensitive type names (/cs) compiler option is enabled), parameter count and types and return type with the base method. If the two methods are different on any of those aspects, then they are considered as completely different methods and one cannot override the other:

```
CLASS Parent
    VIRTUAL METHOD VirtualMethod(n AS INT) AS VOID // child class
has same named method but with different signature
END CLASS

CLASS Child INHERIT Parent
    OVERRIDE METHOD VirtualMethod(c AS STRING) AS INT // compiler
error XS0115: 'VirtualMethod': no suitable method found to
override
    RETURN 0
END CLASS
```

The VIRTUAL/OVERRIDE/NEW modifiers do not apply to methods only, but also to properties or ACCESS/ASSIGN pairs (and also to events, but it makes little sense in declaring/overriding virtual events):

```
FUNCTION Start() AS VOID
    Child{}:DoTest()

CLASS Parent
    VIRTUAL PROPERTY TestProp AS STRING GET "parent"
    ACCESS TestAccess AS STRING
    RETURN "parent"

    METHOD DoTest() AS VOID
        ? SELF:TestProp // child, because the property is overridden
in the child class
        ? SELF:TestAccess // parent, because the access is not
virtual
    RETURN
END CLASS

CLASS Child INHERIT Parent
    OVERRIDE PROPERTY TestProp AS STRING GET "child"
    NEW ACCESS TestAccess AS STRING
    RETURN "child"
END CLASS
```

Finally, also class fields can be declared as NEW (but not as VIRTUAL), in order to differentiate them from fields with the same name in a parent class:

```
FUNCTION Start() AS VOID
    Child{}

CLASS Parent
    EXPORT cField := "parent" AS STRING
    CONSTRUCTOR()
        ? SELF:cField // parent, refers to the filed defined in the
parent class
END CLASS

CLASS Child INHERIT Parent
    NEW EXPORT cField := "child" AS STRING
    CONSTRUCTOR()
        SUPER()
        ? SELF:cField // child, refers to the filed defined in the
parent class
END CLASS
```

Please note that the *XBase*++ dialect also has the modifiers:

DEFERRED	Synonym for ABSTRACT
INTRODUCE	Synonym for NEW
SYNC	Code inside the method is protected from running simultaneously in different threads

ABSTRACT/SEALED

Using the SEALED modifier on a class, prevents it from being inherited from a child class.
This can be useful in order to prevent users of a class/library to alter its functionality in a subclass and possibly introduce problems in it.
It also allows the compiler to emit more efficient code, because it knows that there will be no subclasses of a type.

```
SEALED CLASS Parent
END CLASS

CLASS Child INHERIT Parent // compiler error XS0509: cannot derive
from sealed type 'Parent'
END CLASS
```

SEALED can be used also on specific (virtual) methods/properties to prevent overriding only them in a subclass:

```
FUNCTION Start() AS VOID
   Child{}:DoTest()

CLASS Base
   VIRTUAL METHOD VirtualMethod() AS VOID
      ? "base"
   METHOD DoTest() AS VOID
      SELF:VirtualMethod() // parent, becase method is overriden
in the parent class, but cannot be overriden in the subclass of
parent
END CLASS

CLASS Parent INHERIT Base
   SEALED OVERRIDE METHOD VirtualMethod() AS VOID
      ? "parent"
END CLASS

CLASS Child INHERIT Parent // compiler error
//   OVERRIDE METHOD VirtualMethod() AS VOID // compiler error
XS0239: cannot override method because it is sealed
   NEW METHOD VirtualMethod() AS VOID // can only define a NEW
method with the same name and signature, which does not override
the parent method
      ? "child"
END CLASS
```

The ABSTRACT modifier can be use on a class in order to prevent it from being accidentally directly instantiated. Only classes inheriting (and implementing additional functionality) from an abstract class can be instantiated. This is useful for classes that implement base functionality, but does not make sense creating an instance of them directly (like the Control class in System.Windows.Forms or the same named class in the VOSDK):

```
FUNCTION Start() AS VOID
   LOCAL o AS OBJECT
   o := Child{} // OK
   o := Parent{} // Compiler error XS0144: cannot create an
instance of the abstract type

ABSTRACT CLASS Parent
END CLASS

CLASS Child INHERIT Parent
END CLASS
```

Abstract classes can also define abstract methods or properties, which dictates that it is mandatory to implement them in classes inheriting from the abstract class. Abstract

methods are implicitly virtual so they can be overridden, and it is a compiler error not providing an implementation in a child class:

```
ABSTRACT CLASS Parent
    METHOD BasicFunctionality() AS VOID // does not need to be
overriden in subclass
    ABSTRACT METHOD MustImplementInChild() AS VOID //
implementation must be defined in subclasses
END CLASS

CLASS Child INHERIT Parent
//  Omitting this would result to a compiler error that Child
class does not implement inherited abstract method
    OVERRIDE METHOD MustImplementInChild() AS VOID
END CLASS
```

Please note that XBase++ dialect used the modifier FINAL, which is equivalent to SEALED. The FREEZE modifier is not supported by X#

1.7.5.3 STATIC modifier

The STATIC modifier has a different meaning when used on a class or its members, on a FUNCTION/GLOBAL or on a LOCAL variable.

Static classes and members

Static members

The STATIC modifier on a member of a class (method, constructor, property, field, event) declares a member which belongs to the class itself, instead of to an instance of the class. Normal (instance) methods and other members can be called by using an instance of the class (for example through SELF, or a local variable holding such an instantiated object) with the colon (":") operator (also with the dot "." operator if the compiler option /allowdot is enabled), while static members can be accessed directly through the class itself, using the dot operator:

```
FUNCTION Start() AS VOID
    LOCAL oInstance AS TestClass
    oInstance := TestClass{}

    ? oInstance:instance_field
    ? oInstance:InstanceMethod()
    ? oInstance:InstanceProperty

    ? TestClass.static_field
    ? TestClass.StaticMethod()
    ? TestClass.StaticProperty
```

X#

```
CLASS TestClass
    EXPORT instance_field := "instance field" AS STRING
    STATIC EXPORT static_field := "static field" AS STRING

    METHOD InstanceMethod() AS STRING
    RETURN "Instance method"
    PROPERTY InstanceProperty() AS STRING GET "Instance property"

    STATIC METHOD StaticMethod() AS STRING
    RETURN "Static method"
    STATIC PROPERTY StaticProperty() AS STRING GET "Instance
property"
END CLASS
```

A static method is very similar to a common function and can be used in the same way, but has the advantage that it is encapsulated within a class that may contain other related static or non static members. Like functions, static methods can not use instance members of the class with SELF:, but can access other static members. Methods with related functionality can be grouped together under the same class, offering much better intellisense support in the editor (typing a dot after a class name will show a list of all its static members) and makes the code more structured. For example, the System.Math class offers a lot of mathematical functions, all grouped together under a single class, which makes it easier to find and use, rather that having several standalone functions offering the same functionality. Similarly, a static field can be regarded as a GLOBAL, but again structured under a class. Any method of a class that is not using SELF in its body (so is not accessing any instance members) is a good candidate to be declared as static.

Static constructor

It is also possible to define a single static constructor per class, which is guaranteed to be called automatically just before any static member (method, field, etc) of the class is accessed for the first time. A static constructor must have no parameters and cannot be overloaded. Typical use of a static constructor is to initialize static fields:

```
FUNCTION Start() AS VOID
    ? TestClass.static_field // adjusted by static constructor

CLASS TestClass
    STATIC EXPORT static_field := "initial value" AS STRING

    STATIC CONSTRUCTOR()
        ? TestClass.static_field // initial value
        TestClass.static_field := "adjusted by static constructor"
END CLASS
```

Note that it is not allowed to define a static destructor. If you need to cleanup data stored inside a static class, then it is recommended to register a ProcessExit event handler in the AppDomain class.This will be called when the application ends.

```
FUNCTION Start() AS VOID

CLASS TestClass

    STATIC CONSTRUCTOR()
        AppDomain.CurrentDomain:ProcessExit +=
EventHandler{CurrentDomain_ProcessExit}

static method CurrentDomain_ProcessExit(sender as object, e as
EventArgs) as void
    // clean up
    return
END CLASS
```

Static class

When a class is designed to hold only static members, then it can be declared itself as static. Doing so allows the compiler to check every member of the class and make sure none is accidentally declared as non-static:

```
STATIC CLASS StaticClass
    STATIC EXPORT static_field AS STRING
    EXPORT accidental_instance AS STRING // compiler error XS0708:
cannot declare instance members in a static class

    STATIC METHOD Static_method() AS VOID
    METHOD Accidental_Instance_method() AS VOID // compiler error
XS0708: cannot declare instance members in a static class
END CLASS
```

Static functions and globals

Declaring a function or global as static, restricts its visibility to only the code file where it is declared. This way, multiple files can declared functions and globals with the same name, each one being visible only in their respective file:

```
STATIC GLOBAL GlobalVisibleOnlyInThisFile AS INT
STATIC FUNCTION FunctionVisibleOnlyInThisFile() AS VOID
```

Static local

Declaring a LOCAL variable as STATIC, causes it to be initialized with a value only once, at the first time that the method, function or other entity where it is declared is called. The local variable retains its previous value the next time the code is called:

```
FUNCTION Start() AS VOID
    ? TestStaticLocal() // 2
    ? TestStaticLocal() // 3
    ? TestStaticLocal() // 4


FUNCTION TestStaticLocal() AS INT
    STATIC LOCAL nStaticLocal := 1 AS INT // gets initialized to 1
only the first time the function is called
    nStaticLocal ++ // the current value will be retained the next
time the function is called
    RETURN nStaticLocal
```

1.7.5.4   ASYNC/AWAIT

1.7.5.5   Other modifiers

PARTIAL

Using the PARTIAL modifier on a class declaration, instructs the compiler that the class definition and its members (methods, properties etc) may span in multiple program files in the current application/library. By default (when not using the PARTIAL modifier), all members of a class are expected to be defined in a single file, and if they span in multiple files, then all of the CLASS...END CLASS definitions of that class need to be marked as PARTIAL:

```
// file code1.prg
PARTIAL CLASS ClassSpanningInMultipleFiles // omitting PARTIAL
would result to compiler error on missing partial modifier on
declaration of type
    METHOD MethodInFile1() AS VOID
END CLASS

// file code2.prg
PARTIAL CLASS ClassSpanningInMultipleFiles
    PROPERTY PropertyInFile2() AS LOGIC
        GET
            RETURN TRUE
        END GET
    END PROPERTY
END CLASS
```

Note that all members of a partial class need to be defined in the same assembly. It is NOT possible to use the PARTIAL modifier to define members of a class in separate assemblies/libraries.

EXTERN

The EXTERN modifier tells the compiler that a method is implemented externally, so it has no body in the code itself. Most commonly EXTERN is used together with the DllImport attribute (System.Runtime.InteropServices.DllImportAttribute), which specifies exactly where the method is implemented (usually in a Win32 API dll) and how it needs to be called. When using that, the method needs to be declared also as static:

```
USING System.Runtime.InteropServices
FUNCTION Start() AS VOID
    WinAPICalls.MessageBox(IntPtr.Zero, "Calling unmanaged code
through an EXTERN method", "EXTERN modifier sample", 0)

STATIC CLASS WinAPICalls
    [DllImport("user32.dll", CharSet := CharSet.Unicode)];
    STATIC EXTERN METHOD MessageBox(hWnd AS IntPtr, text AS STRING
, caption AS STRING , type AS DWORD ) AS INT
END CLASS
```

Note that it is a compiler error for an EXTERN method to directly return a value, since its complete implementation is provided externally.

UNSAFE

The UNSAFE modifier on a method specifies that its entire body will run in an unsafe context, meaning it can use pointers to data and manipulate memory directly. The unsafe modifier can be used only when the compiler option Allow unsafe code (/unsafe) is enabled:

```
FUNCTION Start() AS VOID
    ? UnsafeMethods.GetByte2of4(0x10203040) // 32 (hex 20)

CLASS UnsafeMethods
    UNSAFE STATIC METHOD GetByte2of4(d AS DWORD) AS BYTE
        LOCAL p AS BYTE PTR
        p := (BYTE PTR) @d // get direct pointer to the data
        RETURN p[3]
END CLASS
```

When only a small part of the method body involves pointers or other potentially unsafe operations, it is more common to declare a BEGIN UNSAFE block for it, instead of marking the whole method as unsafe:

```
CLASS UnsafeMethods
    STATIC METHOD GetByte2of4(d AS DWORD) AS BYTE
        LOCAL b AS BYTE
        ? "Begin of unsafe possibly code executing"
```

```
        BEGIN UNSAFE // code inside this block uses pointer
operations
            LOCAL p AS BYTE PTR
            p := (BYTE PTR) @d // get direct pointer to the data
            b := p[3]
        END UNSAFE
        ? "End of unsafe code"

        RETURN b
END CLASS
```

Note that writing code that uses pointers is strongly discouraged (as the code cannot be verified by the .Net runtime for correct usage and can destabilize the running application) and needs to be used with extreme caution and only when it is absolutely necessary to do so.

1.7.6   Namespaces

A Namespace is a prefix to item names that provides a logical grouping of types and other
.Net elements, making it easier to structure together items that have a common purpose
and avoid naming conflicts. Due to the vast amount of libraries and classes available in
.Net (in System and 3rd party libraries and in every application written in .Net), using
simple small names without a namespace would had led to a lot of naming conflicts,
making it impossible to distinguish between each other. For example, there's a type
named "Button" (representing a Button control) in the Windows Forms system library,
another for WPF, also one in the VOSDK GUI classes and of course the new typed
VOSDK GUI classes library has one, too. Furthermore, it's very likely that also dozens of
other custom control libraries include a same named type, too! In order to distinguish
among all them, an additional prefix name (namespace) has been added to each version
of the type:

- System.Windows.Forms.Button // Windows forms
- System.Windows.Controls.Button // WPF
- VO.Button // VOGUI library
- XSharp.VO.Button // Typed SDK GUI library

For the full class name System.Windows.Forms.Button, everything before the final dot
(System.Windows.Forms) is called the **namespace** of the class, while the rest is called
the **short class name**. Note that a namespace is not something concrete, it does not
exist as a separate entity in .Net assembiles, it is only a convention to use a descriptive
name prefix for all class names and is not mandatory. Using the dot as name part
separator is also a convention in .Net, it could had been equally valid to choose another
character like an underscore for the same purpose, resulting to the name
System_Windows_Forms_Button for the winforms class name, where
"System_Windows_Forms" would had been the namespace part of the name.

Also by convention, usually namespace and type names are structured in the following
format:

```
<Company name>.<Library name>.<Optional further group
names>.<Short type name>
```

This has been followed in the name XSharp.VO.Button, where "XSharp" is the company
name, "VO" represents the library and "Button" is the actual class (short) name. The
optional additional group names provide further better structuring of the type names, often
used in large libraries where it is important to logically group together the large amount of
available items.

Specifying namespaces to type names

In X#, there are several ways to provide a namespace to types/classes defined in the
code. The most common way is to use the BEGIN..END NAMESPACE block statement:

```
BEGIN NAMESPACE OurCompany.CommonLibrary
   CLASS GeneralUseType
   // ...
```

```
        END CLASS
    END NAMESPACE
```

Every typed defined inside the block will automatically have its name prefixed by the compiler with the namespace provided, so the class in the above sample will become OurCompany.CommonLibrary.GeneralUseType. Also BEGIN NAMESPACE blocks can be nested (to any level) and the above could had been written in an equal way like this:

```
BEGIN NAMESPACE OurCompany
    BEGIN NAMESPACE CommonLibrary
        CLASS GeneralUseType
        // ...
        END CLASS
    END NAMESPACE
END NAMESPACE
```

Another option is to provide the namespace part of the name in the class declaration directly:

```
CLASS OurCompany.CommonLibrary.GeneralUseType
// ...
END CLASS
```

Finally, it is possible to define a **Default Namespace** in the project properties (which maps to the -ns compiler option). When defining this, then all class names in the code that have not been explicitly assigned a namespace, automatically get the one provided in the project option. This is particularly useful for applications and libraries ported from Visual Objects or other systems that did not support the concept of namespaces so all types in them used simple class names. In order to avoid having to explicitly provide namespace names in potentially 100s or 1000s of files of code, the project option can be used instead.

Using types with namespaces

Normally, in order to use a type in code, its full class name (including the namespace part) needs to be used:

```
FUNCTION Start() AS VOID
    LOCAL oUse AS OurCompany.CommonLibrary.GeneralUseType
    oUse := OurCompany.CommonLibrary.GeneralUseType{}
```

If the namespace part was not included, then the compiler would not had been able to resolve the class name. But because always using such long names can be tedious and may result to all of code bloat, it is common practice to specify the commonly used namespace in each file of code in the beginning of the file, in USING statements:

```
USING OurCompany.CommonLibrary

FUNCTION Start() AS VOID
   LOCAL oUse AS GeneralUseType
   oUse := GeneralUseType{}
```

The USING statement instructs the compiler every time it finds a class name that it cannot resolve, to try resolving it by prefixing it with all the namespace names provided in USING statements in the given file. Of course if two or more types used in the code have the same short name with different namespaces, then it is not possible to have a USING statement for each one of them, as that would lead to a name conflict, making it impossible for the compiler to distinguish with exact class is used in each case.

Also note that any code between a BEGIN...END NAMESPACE statement, additionally automatically resolves short type names defined in it to full type names using the namespace provided in the statement. So it is not necessary to provide a namespace in the class name in this code:

```
BEGIN NAMESPACE OurCompany.CommonLibrary
   CLASS GeneralUseType
   END CLASS
   CLASS AnotherClass
      METHOD Test() AS VOID
         LOCAL oUse AS GeneralUseType // not necessary to provide
the namespace
         oUse := GeneralUseType{}
   END CLASS
END NAMESPACE
```

Implicit namespace lookup

Especially for libraries with the **Default namespace** option (see above) provided, it is possible for applications that reference them to use their classes with their short names without needing to provide USING statements. This can be done by enabling the **Enable Implicit Namespace lookup** project option (which maps to the /ins compiler option). Every such library includes its default namespace as information in an assembly attribute and when that option is used, the compiler automatically resolves type names also by using that default namespace in the library. This is used for example in the VOSDK classes, so all the classes defined in those libraries can be used without the need to provide **USING VO** statements, in all files of the applications that reference those libraries.

1.7.7 Types

1.7.8 Tuples

Tuple generic data type

A tuple is a data structure that groups multiple data elements in a lightweight data structure. Tuples can be used as an easy way to represent multiple data elements, pass them as argument(s) or receive them as a return value, without the need to declare a complete new class or structure for holding the data set.

For example, the following code uses the generic System.Collections.Generic.Tuple type to receive multiple values from a function, without the need to declare multiple REF/OUT parameters or to create a new dedicated type for holding them:

```
USING System.Collections.Generic

FUNCTION GetCustomerData() AS Tuple<STRING, INT, LOGIC>
   LOCAL oCustomer AS Tuple<STRING, INT, LOGIC>
   oCustomer := Tuple<STRING, INT, LOGIC>{"Nikos", 47, TRUE}
RETURN oCustomer

FUNCTION Start() AS VOID
   LOCAL oCustomer AS Tuple<STRING, INT, LOGIC>
   oCustomer := GetCustomerData()
   ? "Customer name:", oCustomer:Item1
   ? "Age:", oCustomer:Item2
   ? "Is active:", oCustomer:Item3
```

TUPLE keyword

X# also supports the dedicated keyword TUPLE, which offers an easier syntax for declaring and working with tuples. The above code can be written in a simpler way with the TUPLE keyword syntax:

```
FUNCTION GetCustomerData() AS TUPLE(STRING, INT, LOGIC)
   LOCAL oCustomer AS TUPLE(STRING, INT, LOGIC) // defining the
tuple
    // tuple item types must match the types in the Tuple
definition
   oCustomer := TUPLE{"Nikos", 47, TRUE}
RETURN oCustomer
```

X#

```
FUNCTION Start() AS VOID
    LOCAL oCustomer AS (STRING, INT, LOGIC)
    // the TUPLE keyword in the tuple definition can even be
omitted!
    oCustomer := GetCustomerData()
    ? "Customer name:", oCustomer:Item1
    ? "Age:", oCustomer:Item2
    ? "Is active:", oCustomer:Item3
```

Note that when using the TUPLE keyword, the compiler internally uses the type System.ValueTuple, instead of the generic type System.Collections.Generic.Tuple. Also when defining the tuple, the TUPLE keyword can be omitted, thus using the more c#-like syntax (type1, type2, ...)

Named tuples

The dedicated TUPLE keyword syntax supports also specifying custom names for the tuple items, instead of using the generic names Item1, Item2 etc. So the code can become much more readable and close to what it would be like if using a separate new class for the data:

```
FUNCTION Start() AS VOID
    LOCAL oCustomer AS TUPLE(Name AS STRING, Age AS INT, IsActive
AS LOGIC)
    // provide names for each item of the tuple
    oCustomer := GetCustomerData()
    ? "Customer name:", oCustomer:Name
    ? "Age:", oCustomer:Age
    ? "Is active:", oCustomer:IsActive
```

to make code even more self-explanatory, a named tuple can also be instantiated by using named arguments:

```
FUNCTION GetCustomerData() AS Tuple(STRING, INT, LOGIC)
    // defining the named tuple with custom item names
    LOCAL oCustomer AS Tuple(Name AS STRING, Age AS INT, IsActive
AS LOGIC)
    // including tuple item names in the tuple instantiation
    oCustomer := TUPLE{Name := "Nikos", Age := 47, IsActive :=
TRUE}
RETURN oCustomer
```

Defining tuples with VAR

Tuples can also be defined and instantiated at the same time using the VAR keyword:

```
FUNCTION Start() AS VOID
    VAR oTuple := TUPLE{"Nikos", 47, FALSE}
    ? oTuple:Item1, oTuple:Item2, oTuple:Item3 // "Nikos", 47, FALSE
```

In this case, the data type of each tuple item is inferred from the item types supplied in the code. In the above sample, the item types are (STRING, INT, LOGIC).

Tuples with the same item types can be assigned to each other (thus copying the item values from the source tuple to the destination tuple), and the same can be done for tuples defined with either LOCAL or VAR:

```
FUNCTION Start() AS VOID
    VAR oTuple := TUPLE{"Nikos", 47, FALSE}
    LOCAL oNew AS TUPLE(STRING, INT, LOGIC)

    oNew := oTuple
    ? oNew:Item1, oNew:Item2, oNew:Item3 // "Nikos", 47, FALSE
```

When defining a tuple with the VAR keyword, item names can be provided as well:

```
FUNCTION Start() AS VOID
    VAR oTuple := TUPLE{Name := "Nikos", Age := 47, IsActive :=
FALSE}
    ? oTuple:Name, oTuple:Age, oTuple:IsActive // "Nikos", 47,
FALSE
```

Finally, when defining a tuple with VAR by using identifiers for the item values, instead of using literal values, then each item automatically gets a name by the identifier name used for it:

```
CLASS CustomerInfo
    EXPORT Description := "Customer description" AS STRING
END CLASS

FUNCTION Start() AS VOID
    LOCAL name := "Unknown Customer" AS STRING
    LOCAL oInfo := CustomerInfo{} AS CustomerInfo

    VAR oTuple := TUPLE{name, oInfo:Description}
    ? oTuple:name // "Unknown Customer"
    ? oTuple:description // "Customer description"
```

Tuple deconstruction

A tuple can be deconstructed to multiple plain variables in one line of code, using the (var1, var2, ...) syntax:

```
FUNCTION Start() AS VOID
    LOCAL oCustomer AS TUPLE(Name AS STRING, Age AS INT)
    oCustomer := TUPLE{"Nikos", 47}

    LOCAL name AS STRING
    LOCAL age AS INT

    (name, age) := oCustomer
    ? name, age // "Nikos", 47
```

The local variables can also be defined and assigned to the tuple item values in a single line, with a special LOCAL syntax for tuples:

```
FUNCTION Start() AS VOID
    LOCAL oCustomer AS TUPLE(Name AS STRING, Age AS INT)
    oCustomer := TUPLE{"Nikos", 47}

    LOCAL (name AS STRING, age AS INT) := oCustomer
    ? name, age // "Nikos", 47
    // You can also deconstruct into existing local variables
without the LOCAL keyword:
    (name, age) := oCustomer
```

Also the VAR keyword can be used for the tuple deconstruction, in which case the variable types are inferred from the tuple item types:

```
FUNCTION Start() AS VOID
    LOCAL oCustomer AS TUPLE(Name AS STRING, Age AS INT)
    oCustomer := TUPLE{"Nikos", 47}

    VAR (name, age) := oCustomer
    ? name, age // "Nikos", 47
```

1.7.9   XML Documentation Comments

In X#, you can create documentation for your code by including XML elements in special comment fields (indicated by triple slashes) in the source code directly before the code block to which the comments refer, for example.

```
/// <summary>
///  This class performs an important function.
/// </summary>
CLASS MyClass
.
.
END CLASS
```

When you compile with the -doc option, the compiler will search for all XML tags in the source code and create an XML documentation file. To create the final documentation based on the compiler-generated file, you can create a custom tool or use a tool such Sandcastle.

To refer to XML elements (for example, your function processes specific XML elements that you want to describe in an XML documentation comment), you can use the standard quoting mechanism (< and >). To refer to generic identifiers in code reference (cref) elements, you can use either the escape characters (for example, cref="List<T>") or braces (cref="List{T}"). As a special case, the compiler parses the braces as angle brackets to make the documentation comment less cumbersome to author when referring to generic identifiers.

Since we use the same documentation engine that C# does [we refer to the C# documentation](#) for the documentation tags that are allowed:

X#

1.7.10   Strong Typing

1.7.11 Runtime Scripting

XSharp 2.8 adds support for Runtime scripting through the ExecScript() function.
Scripting was added for the FoxPro dialect but also works for other dialects.
Inside scripts you will have the full X# language at your disposal.
The first time a script runs it will be compiled. If you run the same script a second time then the script compiler can reuse the already compiled version of the first script.

An example (FoxPro dialect, MessageBox() is a FoxPro functon)

```
VAR cScript := 'MessageBox("Hello ExecScript")'
ExecScript(cScript)
```

Of course scripts can also be multiple lines and can call any function in the runtime and in your code.
PUBLIC and PRIVATE variables that are visible when your script is started are accessible by the script.
You can also declare new local variables in the script and use any statement and user defined command that you would normally use.

We did an online session about scripting in Jan 2021 that shows several example scripts.
See https://www.youtube.com/watch?v=88crZsEiAOg&t=5s for the recording.
One sample script from that demo is

```
LPARAMETERS oForm
USE employees.dbf
PrintOut(oForm, 'private',MyPrivate, Used(), Alias())
GO TOP
DO WHILE ! Eof()
    PrintOut(oForm, RecNo(), FieldGet(1), FieldGet(2), FieldGet(3))

    SKIP
ENDDO
USE
```

That script calls a PrintOut function and passes it the form that was received as parameter. The PrintOut function then adds a line of text to the terminal window that was passed to the script. The USE, GO TOP and SKIP commands are all UDCs that are handled by the script compiler without problems.

1.7.12 Calling conventions

Calling conventions are something from the unmanaged world. They describe how parameters should be passed when you call a function or method and they also describe who is responsible for adjusting the stack when the called function / method returns. Different compilers have different default strategies for passing parameters to functions.

Convention	Description
STRICT	This is the most common calling convention in the C/C++ world. With this convention parameters are pushed on the stack. Value types are pushed completely and for reference types the address of the variable is pushed. When a method is called then also the address of the "this" object is pushed on the stack. After the function / method returns then the calling method adjusts the stack. This calling convention allows for functions or methods with a variable number of arguments, like printf(). The caller knows the # of parameters that were passed, so the calling is the best candidate for adjusting the stack. In C/C++ this is also called __cdecl In VO (and X#) there is also a synonym "ASPEN" for this.
PASCAL	This calling convention is used a lot in the Pascal world. It looks a lot like the STRICT calling convention, but now the function / method that gets called adjusts the stack when it returns. Of course, this also means that there cannot be a variable number of arguments. If and when that is necessary, then usually the last parameter becomes an array of values, so there still is some flexibility. In C/C++ this is called the __stdcall calling convention. This calling convention is used by most functions in the windows API. In VO this is also called WINCALL or CALLBACK. In 16bits windows WINCALL was different from PASCAL but 32 bits windows and later dropped that difference.
THISCALL	This is a special variant of the PASCAL calling convention, where the "this" pointer is not pushed on the stack but passed in a register (usually the ECX register). Passing the "this" in the register can be faster, especially when the register is not used for something else, so repeated calls for the same object to not have to push the "this" pointer. In C/C++ this is called __thisccall
FASTCALL	This calling convention tries to pass as many parameters in registers as possible. In C/C++ this is called __fastcall.
CLIPPER	This is a special calling convention in the Xbase world, where parameters to a function are (technically) all optional and where you can also pass more values than you have declared parameters. Originally in the Xbase languages the calling code would push the values on the stack and would also pass the parameter count, so the function that is called "knows" how many parameters are passed. In .Net there is no real equivalent for that. To emulate the CLIPPER calling convention we generate a special PARAMS parameter that contains an array of USUAL values. Parameters of type PARAMS must

Convention	Description

be the last (or only) parameter in the list of parameters. The Roslyn compiler (that we use for x#) will automatically wrap all values that are passed to a function / method with clipper calling convention in an array. Of course, when you declare a function like this

```
FUNCTION Foo(a,b,c)
```

Then you expect that you will have 3 local variables in your code with the names "a", "b" and "c".
The compiler however generates a function with a params argument. Something like this:

```
FUNCTION Foo(args PARAMS USUAL[])
```

Inside the function we then generate local variables with the name of the parameters that you have declared

```
LOCAL a := args[1] as USUAL
LOCAL b := args[2] as USUAL
LOCAL c := args[3] as USUAL
```

In reality, the code is a bit more complex, because you may decide to all the function with less parameters than were declared. We have to take that into account.
It looks like this then:

```
LOCAL numParams := args:Length
LOCAL a := iif(numParams > 0, args[1], NIL) AS USUAL
```

The names for "numParams" and "args" are generated by the complier with a special character in them, to make sure that we do not introduce variable names that conflict with names in your code.
The X# debugger support layer also hides these special variables.

For "normal" managed code, you really only have to deal with 2 calling conventions:
- For untyped methods the compiler uses the CLIPPER calling convention
- For typed methods the compiler there is no difference between STRICT and PASCAL. They both produce the same code

Only when you call unmanaged code in other DLLs then you need to use one of the other calling conventions. You have to "know" what the DLL uses. One problem is that quite often the calling convention in C/C++ code is hidden in a compiler macro.
As a rule of thumb you should use STRICT for C/C++ code and PASCAL for windows api funtions.
If it does not work (for example, the .Net runtime complains about stack problems), then switch to the oher calling convention.

X#

1.8   X# Language Reference

This section is a reference to the X# language

X# Language elements

The following table shows reference topics that provide tables of keywords, symbols and literals used as tokens in X#.

Title	Description
Keywords	Contains links to information about all X# language keywords.
Expression	A list of possible expressions
Symbol and Operators	Contains a table of symbols and operators that are used in the X# language.
Literals (X#)	Describes the syntax for literal values in X# and how to specify type information for X# literals.
Commands and Statements	A List of commands and statements

1.8.1 Keywords

The table below has the keywords that are available in the X# language.

- The Keywords in the **VO** column find their origin in the Visual Objects language. When the compiler dialect is set to VO then these keywords may be abbreviated (4 letter minimum)
- The Keywords in the **VN** column were introduced in Vulcan.NET. These keywords may never be abbreviated, and most of these keywords are positional, so are only recognized in certain positions in the language. That also means that these keywords may be used as Variable Names or Method names
- The Keywords in the **X#** column were introduced in X#. Just like the VN keywords they may never be abbreviated and they are also positional.
- The Keywords in the **VFP** column are FoxPro specific. Please note that many FoxPro commands are implemented as "User Defined Commands" and their tokens are not strictly Keywords inside X#.
- The Keywords in the **Xb++** column are Xbase++ specific. Please note that many FoxPro commands are implemented as "User Defined Commands" and their tokens are not strictly Keywords inside X#.
- Keywords that are listed in the Id column may also be used as an identifier. These are so called "context sensitive" keywords. You may see in the Visual Studio editor that these keywords will change color depending on the position in the source.
  For example if you start typing PROPERTY the word will be shown in the IDENTIFIER color:

```
BEGIN NAMESPACE ClassLibrary1
    CLASS Class1
    CONSTRUCTOR()
        RETURN
    PROPERTY ▮
    END CLASS
END NAMESPACE
```

But as soon as you continue to complete the PROPERTY definition it will get the KEYWORD color:

```
BEGIN NAMESPACE ClassLibrary1
    CLASS Class1
    CONSTRUCTOR()
        RETURN
    PROPERTY Foo as STRING AUTO
    END CLASS
END NAMESPACE
```

Keyword	VO	VN	X#	VFP	Xb++	Id
ABSTRACT		Y				Y
ACCESS	Y					
ADD			Y			Y
ALIGN	Y					Y

Keyword	VO	VN	X#	VFP	Xb++	Id
AND				Y		Y
ANNOUNCE					Y[2]	Y
ANSI		Y				Y
ARRAY	Y					Y[1]
AS	Y					
ASCENDING			Y			Y
ASPEN	Y					Y
ASSEMBLY						Y
ASSIGN	Y					
ASSIGNMENT					Y	Y
ASYNC			Y			Y
AUTO		Y				Y
AWAIT			Y			Y
BEGIN	Y					
BREAK	Y					
BY			Y			Y
BYTE	Y					Y[1]
CALLBACK	Y					Y
CASE	Y					
CATCH		Y				
CCALL	Y					
CCALLNATIVE		Y				
CHAR		Y				Y
CHECKED			Y			Y
CLASS	Y					
CLIPPER	Y					Y

Keyword	VO	VN	X#	VFP	Xb++	Id
CODEBLOCK	Y					Y[1]
CONST		Y				Y
CONSTRUCTOR		Y				
DATE	Y					Y[1]
DECLARE	Y					Y
DEFAULT		Y				Y
DEFERRED					Y	Y
DEFINE	Y					Y
DELEGATE		Y				Y
DESCENDING			Y			Y
DESTRUCTOR		Y				
DIM	Y					Y
DIMENSION				Y		Y
DLLEXPORT	Y					Y
DO	Y					
DOWNTO	Y					Y
DWORD	Y					Y[1]
DYNAMIC			Y			Y
EACH				Y		Y
ELSE	Y					
ELSEIF	Y					
END	Y					
ENDCASE	Y					
ENDDO	Y					
ENDCLASS					Y	Y

Keyword	VO	VN	X#	VFP	Xb++	Id
ENDDEFINE				Y		Y
ENDIF	Y					
ENDFOR				Y[2]	Y[2]	Y
ENDFUNC				Y[2]		Y
ENDPROC				Y[2]		Y
ENDSCAN				Y[2]		Y
ENDSEQUENCE					Y[2]	Y
ENDTRY				Y[2]		Y
ENDWITH				Y[2]		Y
ENUM		Y				Y
EQUALS			Y			Y
EVENT		Y				Y
EXIT	Y					
EXCLUDE				Y		Y
EXPLICIT		Y				Y
EXPORT	Y					
EXPORTED					Y	Y
EXTERN			Y			Y
FALSE	Y					
FASTCALL	Y					Y
FIELD	Y					Y
FINAL					Y	Y
FINALLY		Y				
FIXED			Y			Y
FLOAT	Y					
FOR	Y					
FOREACH		Y				Y

Keyword	VO	VN	X#	VFP	Xb++	Id
FOR EACH				Y[2]	Y[2]	Y
FREEZE		`			Y	Y
FROM			Y			Y
FUNC	Y					Y
FUNCTION	Y					
GET		Y				Y
GLOBAL	Y					Y
GROUP			Y			Y
HELPSTRING				Y		Y
HIDDEN	Y					
IF	Y					
IIF	Y					
IMPLEMENTS		Y				Y
IMPLICIT		Y				Y
IMPLIED		Y				Y
IN	Y					Y
INHERIT	Y					Y
INITONLY		Y				Y
INLINE					Y	Y
INSTANCE	Y					Y
INT	Y					Y[1]
INT64		Y				Y[1]
INTERFACE		Y				Y
INTO			Y			Y
INTERNAL		Y				Y
INTRODUCE					Y	Y
IS	Y					

Keyword	VO	VN	X#	VFP	Xb++	Id
JOIN			Y			Y
LET			Y			Y
LOCAL	Y					
LOCK		Y				Y
LONG	Y					Y[1]
LONGINT	Y					Y[1]
LOOP	Y					
LPARAMETERS				Y		Y
MEMBER	Y					
MEMVAR	Y					
METHOD	Y					
MODULE			Y			Y
NAMEOF			Y			Y
NAMESPACE		Y				Y
NEW		Y				Y
NEXT	Y					
NIL	Y					
NOINIT				Y		Y
NOP			Y			Y
NOSAVE					Y	Y
NOT				Y		Y
NULL	Y					
NULL_ARRAY	Y					
NULL_CODEBLOCK	Y					
NULL_DATE	Y					
NULL_OBJECT	Y					
NULL_PSZ	Y					

Keyword	VO	VN	X#	VFP	Xb++	Id
NULL_PTR	Y					
NULL_STRING	Y					
NULL_SYMBOL	Y					
OBJECT	Y					Y[1]
OFF		Y				
OLEPUBLIC				Y		Y
ON		Y				Y
OPERATOR		Y				Y
OPTIONS		Y				
OR				Y		Y
ORDERBY			Y			Y
OTHERWISE	Y					
OVERRIDE			Y			Y
OUT						Y
PARAMETERS	Y					
PARAMS			Y			Y
PARTIAL		Y				Y
PASCAL	Y					Y
PCALL	Y					
PCALLNATIVE		Y				
PCOUNT	Y					
POP		Y				
PRIVATE	Y					
PROC	Y					Y

Keyword	VO	VN	X#	VFP	Xb++	Id
PROCEDURE	Y					
PROPERTY		Y				Y
PROTECTED	Y					
PSZ	Y					Y[1]
PTR	Y					Y[1]
PUBLIC	Y					
PUSH		Y				
REAL4	Y					Y[1]
REAL8	Y					Y[1]
RECOVER	Y					
REF	Y					
REMOVE			Y			Y
REPEAT		Y				
REQUEST					Y[2]	Y
RETURN	Y					
SCOPE		Y				Y
SEALED		Y				Y
SELECT			Y			Y
SELF	Y					
SEQUENCE	Y					Y
SET		Y				Y
SHARED					Y	Y
SHARING					Y	Y
SHORT	Y					Y[1]
SHORTINT	Y					Y[1]
SIZEOF		Y				
STATIC	Y					

Keyword	VO	VN	X#	VFP	Xb++	Id
STEP	Y					Y
STRICT	Y					Y
STRING	Y					Y[1]
STRUCT	Y					
STRUCTURE		Y				Y
SUPER	Y					
SWITCH			Y			Y
SYMBOL	Y					Y[1]
SYNC					Y	Y
THEN				Y		Y
THISCALL	Y					Y
TO	Y					
THROW		Y				
TRUE	Y					
TRY		Y				Y
TUPLE			Y			Y
TYPEOF		Y				
UINT64		Y				Y[1]
UNCHECKED			Y			Y
UNICODE		Y				Y
UNION	Y					Y
UNSAFE			Y			Y
UNTIL		Y				Y
UPTO	Y					Y
USING	Y					Y
USUAL	Y					Y[1]
VALUE		Y				Y
VAR			Y			Y
VIRTUAL		Y				Y
VOID	Y					Y[1]

Keyword	VO	VN	X#	VFP	Xb++	Id
VOLATILE			Y			Y
VOSTRUCT		Y				Y
WARNINGS		Y				
_WINCALL	Y					Y
WHEN			Y			Y
WHERE			Y			Y
WHILE	Y					
WORD	Y					Y[1]
XOR				Y		Y
YIELD			Y			Y
__ARGLIST			Y			
_AND	Y					
_CAST	Y					
_CODEBLOCK	Y					
_DLL	Y					
_FIELD	Y					
_GETFPARAM						
_GETMPARAM						
_INIT1, _INIT2, _INIT3	Y					
_NOT	Y					
_OR	Y					
_SIZEOF	Y					
_TYPEOF	Y					
_XOR	Y					
.AND.	Y					
.F.	Y					

Keyword	VO	VN	X#	VFP	Xb++	Id
.NOT.	Y					
.OR.	Y					
.T.	Y					
.XOR.	Y					
...	Y					
#command		Y				
#define		Y				
#else		Y				
#endif		Y				
#endregion		Y				
#ifdef		Y				
#ifndef		Y				
#include		Y				
#line		Y				
#pragma		Y				
#region		Y				
#translate		Y				
#undef		Y				
#using		Y				
#warning		Y				
#xcommand		Y				
#xtranslate		Y				
#ycommand			Y			
#ytranslate			Y			

1   These type names can only be used as identifiers when the dialect is not Core, VO or Vulcan.
2   This keyword is actually defined by a preprocessor command

1.8.2 Types

The X# language knows the following data types

Type	Description
[Native Types](#)	
[xBase Specific Types](#)	
[User Defined Types](#)	

1.8.2.1 Simple (Native) Types

By simple data types we mean data types that are not primarily used to hold other data, e.g. Objects, Structures, Arrays, etc., that we will see later.

Most data types are identical across all .Net languages. This contributes to the ease with which one can use assemblies written in different .Net languages within one application, one of the main The simple data types come in various categories; some include several types. The following table just groups data types by category:

Type	Category	.Net Name	Size in Bits
[BYTE](#)	Unsigned Integer	Byte	8
[CHAR](#)	Character	Char	16
[DWORD](#)	Unsigned Integer	UInt32	32
[DECIMAL](#)	Numeric	Decimal	96
[DYNAMIC](#)	Multi purpose	Dynamic	Reference (32 or 64 bits)
[INT](#)	Signed Integer	Int32	32
[INT64](#)	Signed Integer	Int64	64
[LOGIC](#)	Logic	Boolean	8
[LONGINT](#)	Signed Integer	Int32	32
[OBJECT](#)	Multi purpose	Object	Reference (32 or 64 bits)
[PTR](#)	Multi purpose	Intptr	Reference (32 or 64 bits)
[REAL4](#)	Floating Point	Single	32
[REAL8](#)	Floating Point	Double	64
[SBYTE](#)	Signed Integer	SByte	8

Type	Category	.Net Name	Size in Bits
SHORT	Signed Integer	Int16	16
STRING	String	String	Reference (32 or 64 bits)
UINT64	Unsigned Integer	Uint64	64
VOID	Not a type	Void	0
WORD	Unsigned Integer	UInt16	16

1.8.2.1.1 BYTE

The BYTE keyword denotes an integral type that stores unsigned 8-bit values with values that range from 0 to 255.

1.8.2.1.2 CHAR

The CHAR keyword denotes an type to represent a Unicode character. The Unicode Standard identifies each Unicode character with a unique 21-bit scalar number called a code point, and defines the UTF-16 encoding form that specifies how a code point is encoded into a sequence of one or more 16-bit values. Each 16-bit value ranges from hexadecimal 0x0000 through 0xFFFF and is stored in a Char structure. The value of a Char object is its 16-bit numeric (ordinal) value.

1.8.2.1.3 DECIMAL

The DECIMAL keyword describes the .Net System.Decimal type.
The Decimal value type represents decimal numbers ranging from positive 79,228,162,514,264,337,593,543,950,335 to negative 79,228,162,514,264,337,593,543,950,335. The Decimal value type is appropriate for financial calculations that require large numbers of significant integral and fractional digits and no round-off errors. The Decimal type does not eliminate the need for rounding. Rather, it minimizes errors due to rounding. For example, the following code produces a result of 0.9999999999999999999999999999 instead of 1.

1.8.2.1.4 DWORD

The DWORD keyword denotes an integral type that stores unsigned 32-bit values with values ranging from 0 to 4,294,967,295.

1.8.2.1.5 DYNAMIC

The dynamic type enables the operations in which it occurs to bypass compile-time type checking. Instead, these operations are resolved at run time. The dynamic type simplifies access to COM APIs such as the Office Automation APIs, and also to dynamic APIs such as IronPython libraries, and to the HTML Document Object Model (DOM).

Type dynamic behaves like type object in most circumstances. However, operations that contain expressions of type dynamic are not resolved or type checked by the compiler. The compiler packages together information about the operation, and that information is later used to evaluate the operation at run time. As part of the process, variables of type dynamic are compiled into variables of type object. Therefore, type dynamic exists only at compile time, not at run time.

Please not that to use the DYNAMIC type in your app you have to include the Microsoft.CSharp DLL at this moment, and the property names and method names are case sensitive at this moment.

1.8.2.1.6 INT

The INT or LONG keyword denotes an integral type that stores signed 32-bit values with values that range from negative 2,147,483,648 (which is represented by the Int32.MinValue constant) through positive 2,147,483,647 (which is represented by the Int32.MaxValue constant) The .NET Framework also includes an unsigned 32-bit integer value type, UInt32 or DWORD, which represents values that range from 0 to 4,294,967,295.

1.8.2.1.7 INT64

The INT64 keyword denotes an integral type that stores signed 64-bit values with values that range from negative 9,223,372,036,854,775,808 through positive 9,223,372,036,854,775,807

1.8.2.1.8 LOGIC

The LOGIC keyword represents the .Net Boolean type. This type can have either of two values: true, or false.

If you have members of type LOGIC in VOSTRUCT or UNION types then these will not be represented with .Net Boolean types because the size of these Boolean is 1 byte but in the Windows API LOGIC values are represented with 4 bytes. Therefore the compiler will replace these with a special type __WinBool which has 4 bytes and has implicit converters between Logic and __WinBool.

1.8.2.1.9 OBJECT

The OBJECT keyword is a generic type from which all objects in .Net automatically inherit.

X# does not require a class to declare inheritance from Object because the inheritance is implicit.

Because all classes in the .NET Framework are derived from Object, every method defined in the Object class is available in all objects in the system. Derived classes can and do override some of these methods, including:

- Equals - Supports comparisons between objects.

- Finalize - Performs cleanup operations before an object is automatically reclaimed.

- GetHashCode - Generates a number corresponding to the value of the object to support the use of a hash table.

- ToString - Manufactures a human-readable text string that describes an instance of the class.

1.8.2.1.10 PTR

The PTR keyword denotes an integral type that stores a pointer to a memory location. It is usually compiled to the System.IntPtr .Net Type.
Please note that the size of the PTR depends on the underlying operating system and it will also be different when your application runs in x86 or x64 mode.
In Visual Objects PTR is always 32 bits. Many people have written code that CAST INT values to PTR and back. This works because both values are 32 bit in VO.
Safe code in .Net cannot do this, since the size of PTR is not fixed.

1.8.2.1.11 REAL4

The REAL4 keyword denotes a single precision 32 bit number with values ranging from negative 3.402823e38 to positive 3.402823e38, as well as positive or negative zero, PositiveInfinity, NegativeInfinity, and not a number (NaN). It is intended to represent values that are extremely large (such as distances between planets or galaxies) or extremely small (such as the molecular mass of a substance in kilograms) and that often are imprecise (such as the distance from earth to another solar system). The REAL4 type complies with the IEC 60559:1989 (IEEE 754) standard for binary floating-point arithmetic.

1.8.2.1.12 REAL8

The REAL8 keyword denotes an double precision 64-bit number with values ranging from negative 1.79769313486232e308 to positive 1.79769313486232e308, as well as positive or negative zero, PositiveInfinity, NegativeInfinity, and not a number (NaN). It is intended to represent values that are extremely large (such as distances between planets or galaxies) or extremely small (the molecular mass of a substance in kilograms) and that often are imprecise (such as the distance from earth to another solar system), The Double type complies with the IEC 60559:1989 (IEEE 754) standard for binary floating-point arithmetic.

1.8.2.1.13 SBYTE

The SBYTE keyword denotes an integral type that stores signed 8-bit values with values ranging from negative 128 to positive 127.

1.8.2.1.14 SHORT

The SHORT keyword denotes an integral type that stores signed 16-bit values with values ranging from negative 32768 through positive 32767.

1.8.2.1.15 STRING

A STRING is a sequential collection of Unicode characters that is used to represent text. A String object is a sequential collection of System.Char objects that represent a string. The value of theString object is the content of the sequential collection, and that value is immutable (that is, it is read-only). For more information about the immutability of strings, see the Immutability and the StringBuilder class section later in this topic. The maximum size of a String object in memory is 2 GB, or about 1 billion characters.

Please note that you cannot CAST strings to PSZ like you can in Visual Objects. The PSZ type consists of 8 bits per character where the STRING type has 16 bits per character.

1.8.2.1.16 UINT64

The UINT64 keyword denotes an integral type that stores unsigned 64-bit values with values ranging from 0 to 18,446,744,073,709,551,615.

1.8.2.1.17 VOID

VOID specifies a return value type for a function or method that does not return a value. Procedures implicitly define a return type of VOID

1.8.2.1.18 WORD

The WORD keyword denotes an integral type that stores unsigned 16-bit values with values ranging from 0 to 65535.

1.8.2.2 xBase Specific Types

Type	Description
ARRAY	
BINARY	
CODEBLOCK	
CURRENCY	
DATE	
FLOAT	
PSZ	
SYMBOL	
USUAL	

1.8.2.2.1 ARRAY

The ARRAY type is a dynamic array of USUAL values. Each element of the array may contain another array, so arrays can be multidimensional.

Implementation

The ARRAY type is implemented in the class XSharp.__Array.
The Usualtype of ARRAY has the value 5

1.8.2.2.2 ARRAY (FoxPro)

The FoxPro dialect in X# has its own Array type. This type is not declared with an AS keyword, but the array type is derived from the context.
The following lines of code all generate a FoxPro compatible array:

```
LOCAL ARRAY aTest(1,2)        // LOCAL ARRAY
PUBLIC ARRAY aPublicArray[10]   // PUBLIC ARRAY
DIMENSION AnotherArray(3,4)   // DIMENSION with parentheses , but
angled brackets are supported too
DECLARE ThirdArray[10]        // DIMENSION with angled brackets, but
parentheses are supported too
```

The elements of a Foxpro compatible array are all USUAL.
FoxPro arrays cannot be dynamically sized with AAdd(). To resize them you need to add a DIMENSION statement with new dimensions.
Internally FoxPro arrays are single dimensional arrays. But you can also (re)dimension them as two dimensional.
So the 3rd array in this example can also be treated as a single dimensional array of 12 elements.

We advise to use angled brackets to access elements of a FoxPro array. This is not ambiguous and the compiler can resolve that at compile time.
If you want to use parentheses to access FoxPro array elements you need to enable the /fox compiler option. This compiler option also enables the behavior that assigning a single value to a FoxPro array will result in assigning that value to all elements in the array.

Internally FoxPro arrays are implemented as a class that derives from the generic XBase array type.
So all functions in the X# runtime that take an array as parameter will also accept a FoxPro array.
When there is different behavior between the FoxPro implementation of a function or the Xbase implementation then this will be handled at runtime.

Implementation

The ARRAY type is implemented in the class XSharp.__FoxArray.
The Usualtype of ARRAY has the value 5

1.8.2.2.3 BINARY

The BINARY type is represented a series of bytes.

- Binary literals are written as 0h12345678abcdef
- The value behind 0h is a sequence of hex numbers. Each pair of hex numbers (nibble) represents 1 byte. There must be an even number of 'nibbles'.
- The binary literals are encoded in an array of bytes. In the Core dialect the binary literals are represented as a byte[]. In the other dialects the binary literals are a new type (XSharp.__Binary) which can be specified as the new BINARY keyword.
- The UsualType() of BINARY is 29.
- The XSharp.__Binary type has operators to add a string to a binary and add a binary to a string.
  Binary + String will return a Binary
  String + Binary will return a String
  Binary + Binary will return a Binary.
  There are also comparison operators on the Binary type (>, <, >=, <=). These will use the string comparison routines that are defined with SetCollation() with the exception that an = comparison with a single equals operator does not return TRUE when the Right hand side is shorter than the Left hand side and the first bytes match.
- Conversions from Binary to String are done with the Encoding.GetBytes() and Encoding.GetString() functions for the current Windows Encoding.
  That means that on single byte code pages each character in the string will result in one byte and each byte will result in one character.
  For multibyte code pages (Chinese, Japanese, Korean etc) some characters will result in more than one byte and some byte pairs will result in a single character.
- There are implicit operators that convert a BINARY to a byte[] and back. There are also implicit operators that convert a Binary to a String and back.
- When compiling with the Vulcan Runtime then the byte[] array is stored in a USUAL value for the non core dialects.

Implementation

The BINARY type is implemented in the class XSharp.__Binary
The Usualtype of BINARY has the value 29

1.8.2.2.4 CODEBLOCK

The **codeblock** type was introduced in the XBase language in the Clipper days.
They can be seen like unnamed functions. They can have 0 or more parameters and return a value.
The most simple codeblock that returns a string literal looks like this

```
FUNCTION Start() AS VOID
LOCAL cb AS CODEBLOCK
cb := {|| "Hello World"}
? Eval(cb)
WAIT

RETURN
```

To use a codeblock you call the Eval() runtime function

Codeblocks are not restricted to fixed expressions, because they can use parameters.
The following codeblock adds 2 parameters.

```
FUNCTION Start() AS VOID
LOCAL cb AS CODEBLOCK
cb := {|a,b| a + b}
? Eval(cb, 1,2)
? Eval(cb, "Helllo ", "World")
WAIT
RETURN
```

As you can see in the example, we can both use numeric parameters here or string
parameters. Both work. That is because the parameters to a codeblock are of the so
called USUAL type. They can contain any value. Of course the following will fail because
the USUAL type does not support multiplying strings:

```
FUNCTION Start() AS VOID
LOCAL cb AS CODEBLOCK
cb := {|a,b| a * b}
? Eval(cb, 1,2)
? Eval(cb, "Helllo ", "World")
WAIT
RETURN
```

More complicated codeblocks

Codeblocks are not restricted to single expressions.
They may also contain a (comma seperated) **list of expressions**. The value of the last
expression is the return value of the codeblock:

```
FUNCTION Start() AS VOID
LOCAL cb AS CODEBLOCK
cb := {|a,b,c| QOut("value 1", a) , QOut("value 2", b),
QOut("value 3", c), a*b*c}
? Eval(cb,10,20,30)
WAIT
RETURN
```

XSharp has also introduced codeblocks that contain of **(lists of) statements:**

```
FUNCTION Start() AS VOID
LOCAL cb AS CODEBLOCK
cb := {| a,b,c|
    ? "value 1" ,a
    ? "value 2" ,b
```

```
         ? "value 3" ,c
         RETURN a*b*c
         }
     ? Eval(cb,10,20,30)
     WAIT
     RETURN
```

Please note

- The first statement should start on a new line after the parameter list
- There should be NO semi colon after the parameter list.
- The statement list should end with a RETURN statement.

Implementation

The CODEBLOCK type is implemented in the abstract class XSharp.Codeblock
The Usualtype of CODEBLOCK has the value 9.

In your code you will never have objects of type XSharp.Codeblock.
Compile time codeblocks are translated into a subclass of XSharp.Codeblock
Runtime (macro compiled) codeblocks are translated into a subclass of the class
XSharp._Codeblock which inherits from Codeblock.
Depending on the type of the runtime codeblock this is either an instance of the
MacroCodeblock class or of the MacroMemVarCodeblock class (when the macro creates
dynamic memory variables)

1.8.2.2.5 CURRENCY

- The currency type stored numbers with a precision of 4 decimals. Internally it contains a .Net decimal value, rounded to 4 decimals.
-

Implementation

The CURRENCY type is implemented in the structure XSharp.__Currency
The Usual type of CURRENCY is 28.

1.8.2.2.6 DATE

The DATE type is an integral type that stores a date value.
The DATE is internally stored in 3 fields (DAY, MONTH and YEAR) that occupy a total of 32 bits in memory.

Implementation

The DATE type is implemented in the structure XSharp.__Date
The Usual type of DATE is 2.

1.8.2.2.7 FLOAT

The FLOAT type is a type that stores a 64-bit floating point value, along with formatting information. The precision and range of a FLOAT are the same as that from a [REAL8](#), since the value of the float is stored in a REAL8.

Implementation

The FLOAT type is implemented in the structure XSharp.__Float
The Usual type of FLOAT is 3.

1.8.2.2.8 PSZ

The PSZ type is a pointer type that points to a null terminated sequence of zero or more bytes, typically representing a printable character string. This type is for backward compatibility only. Don't use this type in new code unless you have to.

Implementation

The PSZ type is implemented in the class XSharp.__Psz
The Usual type of PSZ is 17.

1.8.2.2.9 SYMBOL

The SYMBOL type is a 32-bit integer that represents an index into an array of strings.

Since a SYMBOL represents a string, there is a built-in implicit conversion from SYMBOL to STRING, and from STRING to SYMBOL.

Since the underlying value of a SYMBOL is an integer, there is a built-in explicit conversion from SYMBOL to DWORD and from DWORD to SYMBOL. A cast is necessary in order to perform explicit conversions.

Unlike with Visual Objects, the number of symbols is not limited by available memory or symbols that are declared in another library.

Literal symbols consist of the '#' character followed by one or more alphanumeric character. The first character must be a letter or an underscore.

Some examples of literal symbols are shown below:
#XSharp
#XSHARP

Note that although literal symbols can be specified with lower or upper case letters, the strings they represent are converted to uppercase at compile time, for compatibility with Visual Objects. It is not possible to specify a literal symbol that contains lower case letters, the StringToAtom() function must be used.

The compiler-defined constant NULL_SYMBOL can be used to express a null symbol, i.e. a symbol that has no associated string value.

Implementation

The SYMBOL type is implemented in the structure XSharp.__Symbol
The Usual type of SYMBOL is 10.

1.8.2.2.10 USUAL

The USUAL type is datatype that can contain any data type. It consists internally of a type flag and a value. This type can store any value.
The compiler treats this type in a special way. The compiler will not warn you when you assign a value of type USUAL to another type, but will automatically generate the necessary conversion operation/

USUAL is provided primarily for compatibility with untyped code. It is not recommended for use in new code because the compiler cannot perform any type checking on expressions where one or more operands are USUAL. Any data type errors will only be discovered at runtime.

Locals, parameters and fields declared as USUAL also incur considerably more runtime overhead than strongly typed variables.

The literal value NIL may be assigned into any storage location typed as USUAL. The value NIL indicates the absence of any other data type or value, and is conceptually equivalent to storing NULL into a reference type. NIL is the default value for a local USUAL variable that has not been initialized.

When the left operand of the ':' operator is a USUAL, the compiler will generate a late bound call to the method, field or property specified as the right operand. This call may fail if the value contained in the USUAL at runtime does not have such a member, the member type is incorrect or inaccessible, or if the name evaluates to a method and the number of parameters or their types is incorrect. The /lb compiler option must be enabled in order to use a USUAL as the left operand of the ':' operator, otherwise a compile-time error will be raised.

Numeric operations and USUAL variables of mixed types.

When you combine 2 USUAL variables in a numeric operation then the type of the result is derived from the types of operands.
The leading principle has been that we try not to loose decimals.

The generic rule is:
- When the Left Hand Side is fractional then the result is also fractional of the type of the LHS
- When the LHS is NOT fractional and the Right Hand Side (RHS) is fractional then the result is the type of the RHS
- When both sides are integral then the result has the type of the largest of the two.

LHS	\RHS	LONG	INT64	FLOAT	CURRENCY	DECIMAL
LONG		LONG	INT64	FLOAT	CURRENCY	DECIMAL
INT64		INT64	INT64	FLOAT	CURRENCY	DECIMAL

FLOAT	FLOAT	FLOAT	FLOAT	FLOAT	FLOAT
CURRENCY	CURRENCY	CURRENCY	CURRENCY	CURRENCY	CURRENCY
DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL

Implementation

The USUAL type is implemented in the structure XSharp.__Usual

1.8.2.3   **User defined Types**

The X# language allows you to define your own types (in fact the XBase Specific types have all been defined that way).
The language has the following keywords that define User defined Types.

Type	Description
CLASS	Defines a class, using a generic syntax that is available in all dialect.
CLASS (FoxPro syntax)	Defines a class, using the FoxPro specific syntax.
CLASS (Xbase++ syntax)	Defines a class, using the Xbase++ specific syntax.
DELEGATE	Defines a delegate, a description of a method or a function.
ENUM	Defines an enum, a list of possible options.
INTERFACE	Defines an interface.
STRUCTURE	Defines a structure, a value type
UNION	Defines a Union, a special kind of structure. Only available in the VO and Vulcan dialects.
VOSTRUCT	Defines a VOStruct, a special kind of structure. Only available in the VO and Vulcan dialects.

1.8.3   Literals

The X# language knows the following literal types

Keyword	Description
Char Literals	
Date Literals	
Logic Literals	
Macros	
NULL Literals	
Numeric Literals	
String Literals	
Symbol Literals	

1.8.3.1   Char Literals

There are different notations for a char literal depending on the dialect selected.
In the many dialects the single quotes are string delimiters and therefore the 'c' prefix is
needed for character literals.
Only in the Core and Vulcan dialects single quotes are always reserved for character
literals

Literal	VO, XPP, FoxPro & Harbour	Vulcan & Core
c'<char>' or **c**"<char>"	Char literal*	Char literal*
'<char>'	**String literal**	Char literal*

The <char> literals in the table that are marked with an asterisk (*) may contain a special
escape code

Character	Description
Character that does NOT start with a backslash	Normal character
\\	Backslash
\"	Double quote
\'	Single quote
\0	0 character
\a	Alert

\b	Backspace
\f	Form feed
\n	Line feed
\r	New Line
\t	Tab
\v	Vertical tab
\x HEXDIGIT(1-4)	Hex number of character. 1-4 hex digits
\u HEXGDIGIT (4 or 8)	Unicode number of character. 4 or 8 hex digits

1.8.3.2 String Literals

There are a couple of different string types inside X#. For normal strings the notation can be different for different dialects:

Literal	VO, FoxPro & Harbour	Vulcan & Core
'<char>'	String literal	**Char literal***
'<char>...<char>'	String literal	**Not Supported**
"<char>...<char>"	String literal	String literal
e"<char>...<char>"	Extended string literal*	Extended string literal*
i"<char>..<char>{expression}"	Interpolated string literal	Interpolated string literal
ei"<char>..<char>{expression}" or **ie**"<char>..<char>{expression}"	Extended interpolated string literal*	Extended interpolated string literal*

The <char> literals in the table that are marked with an asterisk (*) may contain a special escape code

Character	Description
Character that does NOT start with a backslash	Normal character
\\	Backslash
\"	Double quote
\'	Single quote
\0	0 character
\a	Alert
\b	Backspace

\f	Form feed
\n	Line feed
\r	New Line
\t	Tab
\v	Vertical tab
\x HEXDIGIT(1-4)	Hex number of character. 1-4 hex digits
\u HEXGDIGIT (4 or 8)	Unicode number of character. 4 or 8 hex digits

1.8.3.3 Date Literals

Date literals are formatted

```
YYYY.MM.DD
```

or alternatively
```
{^YYYY-MM-DD}
```

DateTime Literals can be constructed with the followin syntax
```
{^YYYY-MM-DD HH:MM:SS}.
```

Examples:

```
2000.01.01
2012.02.29
2015.09.25
// foxpro date and datetime literals
{^2019.02.28}
{^2020.02.29 23:59:59}
```

Date and DateTime literals are supported in all dialects. In the Core dialect both date and datetime literals will be translated to a DateTime value. In the other dialects the Date literals are translated to a Date value.
Of course you can also use runtime functions to construct a literal, such as STOD("20150925") or ConDate(2019,9,25) but a literal is more efficient.

1.8.3.4 Logic Literals

X# uses the following Logical literals.

Value	Description
TRUE or .T. or .Y.	Logical TRUE
FALSE or .F. or .N.	Logical FALSE

X#

1.8.3.5 Null Literals

X# uses the following NULL literals. They indicate the absense of a value.
Some of these literals require the runtime.

Value	Description	Requires Runtime
NULL	"Multi purpose" NULL. Can be used for OBJECT type and Pointer types.	N
NULL_ARRAY	Null reference to the xBase ARRAY type	Y
NULL_CODEBLOCK	Null reference to the xBase CODEBLOCK type	Y
NULL_DATE	Empty date (0000-00-00)	Y
NULL_OBJECT	Null reference to an OBJECT value	N
NULL_PSZ	Null PSZ value	Y
NULL_PTR	Null PTR value	N
NULL_STRING	Null reference to a string. When the compiler option /vo2 is used then this will become the equivalent of String.Empty	N
NULL_SYMBOL	NULL reference to the xBase SYMBOL type	Y
.NULL.	This FoxPro literal null is translated to a NULL reference	N

1.8.3.6 Numeric Literals

The X# language knows the following numeric literal types

Keyword	Description
Integer Literals	
Real Liteals	

1.8.3.6.1 Integer Literals

Literal INT values may be specified in decimal, hexadecimal or binary notation:

```
12345
0x1234ABCD
0b010010101
```

Integer literals may have one of the following suffixes:

Suffix	
L or l	Signed 32 bits integer
U or u	Unsigned integer

1.8.3.6.2 Floating point Literals

Single or double precision literals may be used to store numeric values in your code.

```
123.456
123.0
0.123
.123
123.456e+7
123.456e-7
$123.45
```

Real literals may have one of the following suffixes:

Suffix	
S or s	Single precision (REAL4)
D or d	Double precision (REAL8)
M or m	Decimal

Floating point literals without suffix are stored as REAL8, unless you have specified the /vo14 compiler option, in which case they are stored as FLOAT literals.

Prefix	
$	Currency Literal

1.8.3.7 Symbol Literals

Symbol literals consist of the '#' character followed by one of more alphanumeric characters

```
#XSharp
#AnExample
```

Even though the examples above are written in mixed case, the compiler will treat them as all uppercase.
Symbol literals are compiled only once by the X# compiler for performance reasons.
Each .Net assembly created by the X# compiler will have a special, compiler generated, class that contains all the literal symbols found in that assembly.
These literals are initialized at startup and stored in the symbol table in the runtime.

1.8.3.8 Escape codes

The <char> literals in the table that are marked with an asterisk (*) may contain a special escape code

Character	Description
Character that does NOT start with a backslash	Normal character
\\	Backslash
\"	Double quote
\'	Single quote
\0	0 character
\a	Alert
\b	Backspace
\f	Form feed
\n	Line feed
\r	New Line
\t	Tab
\v	Vertical tab
\x HEXDIGIT(1-4)	Hex number of character. 1-4 hex digits
\u HEXGDIGIT (4 or 8)	Unicode number of character. 4 or 8 hex digits

1.8.3.9   Binary Literals

Binary literals are written as 0h12345678abcdef
The value behind 0h is a sequence of hex numbers. Each pair of hex numbers (nibble) represents 1 byte. There must be an even number of 'nibbles'.
The compiler compiles these literals into a byte[] array which is stored into a variable of the type BINARY

1.8.4   Commands and Statements

The X# language knows commands and statements. The difference between these two is:

- Statements are defined by the compiler. Examples are the declarations of functions,local variables and control structures such as IF statements and FOR statements
- Commands are defined in a header file and are preprocessed by the compiler into function calls. Examples of a Command are the USE command and APPEND BLANK command

Statements are available in almost all dialects, with only a few exceptions.
The available commands are very much dependent on the dialect. For example in the Core dialect no commands are available at all. Most commands have optional parameters and depend on the X# runtime types such as USUAL

This reference lists both the commands that are 'built-in' into the compiler and commands that are defined as default 'User Defined Commands' in STD.UDC.
Some commands are marked with an asterisk as 'deprecated'. They may not be available in future versions of %APP%

Built-in Statements	Commands in header file	Commands in header file	Grouped by category
#ifdef #else #endif	ACCEPT	SAVE*	Comment
#ifndef #else #endif	APPEND BLANK	SEEK	Concurrency Control
?|??	APPEND FROM	SELECT	Database
|\\	AVERAGE	SET ANSI	Date
_DLL	CANCEL	SET CENTURY	Entity Declaration
ACCESS	CLEAR ALL*	SET COLLATION	Environment
ASSIGN	CLEAR MEMORY*	SET COLOR	File
BEGIN SEQUENCE	CLOSE	SET DATE	Index/Order
BREAK	COMMIT	SET DATE FORMAT	International
CLASS	CONTINUE	SET DECIMALS	Memory Variable
DECLARE	COPY FILE	SET DEFAULT	Numeric
DELEGATE	COPY STRUCTURE	SET DELETED	Program Control
DEFINE	COPY STRUCTURE	SET DESCENDING	Runtime Declaration
DO*	EXTENDED	SET DIGITFIXED	Terminal Window
DO CASE	COPY TO	SET DIGITS	Variable Declaration
DO WHILE	COUNT	SET DRIVER	
ENUM	CREATE	SET EPOCH	
EXIT	CREATE FROM	SET EXCLUSIVE	
FIELD	DEFAULT	SET FILTER	
FOR	DELETE	SET FIXED	
FOREACH	DELETE FILE	SET INDEX	
FUNCTION	DELETE TAG	SET INTERNATIONAL	
GLOBAL	DIR	SET MEMOBLOCK	
IF	ERASE	SET OPTIMIZE	
LOCAL	EXTERNAL*	SET ORDER	
LOOP	FIND	SET PATH	
MEMVAR	GO	SET RELATION	
METHOD	INDEX	SET SCOPE	
PARAMETERS	JOIN	SET SCOPEBOTTOM	
PRIVATE	LOCATE	SET SCOPETOP	

PROCEDURE	**NOTE**	**SET SOFTSEEK**
PUBLIC	**PACK**	**SET UNIQUE**
REPEAT	**QUIT**	**SKIP**
RETURN	**RECALL**	**SORT**
STATIC	**REINDEX**	**STORE**
STRUCTURE	**RELEASE***	**SUM**
TEXT	**RENAME**	**TOTAL**
UNION	**REPLACE**	**UNLOCK**
VOSTRUCT	**RESTORE***	**UPDATE**
	RUN	**USE**
		WAIT
		ZAP

1.8.4.1   Identifiers

Identifiers in X# appear on many places in the language. They consist of a character followed by one or more characters, numbers or digits. The lexer definition for identifiers is below. As you can see we also allow 'special characters' and 'unicode characters' , but in general that is not recommended.

New keywords that are introduced in Vulcan and X# (see the keyword table) can also be used as Identifier
When an identifier <u>must</u> be used that has the same value as a keyword, then you can prefix the identifier with a double @@ sign, like in the following example.
This is not recommended. But it can happen that code in an external DLL has properties or method names that are keywords in X#. In that case using the @@ prefix can work too.

```
LOCAL @@Class as STRING
LOCAL @@Local as LOGIC
```

```
ID                  : IDStartChar IDChar*
                    ;

fragment IDStartChar: 'A'..'Z' | 'a'..'z'
                    | '_'
                    | '\u00C0'..'\u00D6'
                    | '\u00D8'..'\u00F6'
                    | '\u00F8'..'\u02FF'
                    | '\u0370'..'\u037D'
                    | '\u037F'..'\u1FFF'
                    | '\u200C'..'\u200D'
                    ;

fragment IDChar     : IDStartChar
                    | '0'..'9'
                    | '\u00B7'
                    | '\u0300'..'\u036F'
                    | '\u203F'..'\u2040'
```

;

1.8.4.2   Blocks and Namespaces

There are several block statements that allow you to group a block of code. Variables declared inside a block are only visible for the duration of the block.

BEGIN CHECKED
BEGIN FIXED
BEGIN LOCK
BEGIN SCOPE
BEGIN UNCHECKED
BEGIN UNSAFE
BEGIN USING

1.8.4.2.1   BEGIN (UN)CHECKED

Purpose

The **BEGIN** (**UN**)**CHECKED** and **END** (**UN**)**CHECKED** keywords mark a block of statements that are compiled with overflow checking enabled or disabled

Syntax

```
BEGIN CHECKED
    statements
END CHECKED

BEGIN UNCHECKED
    statements
END CHECKED
```

Arguments

statements   One or more statements or expressions that are compiled with the specified overflow checking

Remarks

BEGIN CHECKED ... END CHECKED ensures that a block of code is compiled with a clear overflow checking option, regardless of the compiler option -ovf.

```
BEGIN CHECKED
    LOCAL intValue as INT
    LOCAL dwordValue as DWORD
    intValue := -1
    dwordValue := (DWORD) intValue    // Overflow error
END CHECKED

BEGIN UNCHECKED
    LOCAL intValue as INT
    LOCAL dwordValue as DWORD
    intValue := -1
    dwordValue := (DWORD) intValue    // NO Overflow error,
dwordValue now has the value UInt32.MaxValue
END UNCHECKED
```

1.8.4.2.2 BEGIN FIXED

Purpose

The **BEGIN FIXED** and **END FIXED** keyword prevent the garbage collector from relocating a movable variable. The **BEGIN FIXED** statement is only allowed in an unsafe context.

The fixed statement sets a pointer to a managed variable and "pins" that variable during the execution of the statement. Pointers to movable managed variables are useful only in a fixed context. Without a fixed context, garbage collection could relocate the variables unpredictably. The X# compiler only lets you assign a pointer to a managed variable in a fixed statement.
You can initialize a pointer by using an array, a string, a fixed-size buffer, or the address of a variable.

Syntax

```
BEGIN FIXED declaration
    statements
END FIXED
```

Arguments

declaration   Declaration of a variable and assignment that

statements	Code including one or more statements that may contain unsafe code.

Example

```
UNSAFE FUNCTION Start AS VOID
    VAR s := "SDRS"
    BEGIN FIXED LOCAL p := s AS CHAR PTR
        VAR i := 0
        WHILE p[i] != 0
            p[i++]++
        END
    END FIXED
    Console.WriteLine(s)
    Console.Read()
    RETURN
```

1.8.4.2.3 BEGIN LOCK

Purpose

The BEGIN LOCK and END LOCK keywords mark a block of statements as a critical section.

Syntax

```
BEGIN LOCK object
    statements
END LOCK
```

Arguments

object	An expression that evaluates to an object reference that is used as a locking object.
statements	One or more statements or expressions that are guarded by a mutual exclusion lock on the object specified in object.

Remarks

BEGIN LOCK ... END LOCK insures that multiple threads cannot execute the statements within the block at the same time. If one thread is executing code within the

block, any other thread that attempts to enter the block will be suspended until the thread that is executing leaves the block.

The object used as the locking object must be a reference type, it cannot be a value type and the expression cannot evaluate to NULL or a runtime error will occur.

BEGIN LOCK ... END LOCK uses Monitor.Enter() and Monitor.Exit() to acquire and release a lock on the specified object. The following example:

```
BEGIN LOCK lockObj
? "In guarded block"
END LOCK
```

is equivalent to:

```
System.Threading.Monitor.Enter( lockObj )
TRY
? "In guarded block"
FINALLY
System.Threading.Monitor.Exit( lockObj )
END TRY
```

Using **BEGIN LOCK ... END LOCK** is recommended over using the Monitor class directly because the code is more concise and insures that the monitor object is released even if an exception occurs within the guarded block.

BEGIN LOCK ... END LOCK provides functionality similar to the Windows API functions EnterCriticalSection() and LeaveCriticalSection(). However, instead of using an object created by InitializeCriticalSection(), any instance of a reference type may be used for the locking object.

Please see the documentation for the System.Threading.Monitor class for more information.

1.8.4.2.4 BEGIN NAMESPACE

Purpose

The BEGIN NAMESPACE and END NAMESPACE keyword pairs declare a scope and add a namespace prefix to all types declared inside this scope

Syntax

```
BEGIN NAMESPACE namespaceName
     typeDeclarations
```

```
        namespaceDeclarations
END NAMESPACE
```

Arguments

namespaceName	The name of the namespace being declared.
typeDeclarations	One or more type declarations (CLASS, STRUCTURE, etc.).
namespaceDeclarations	Zero or more namespace declarations.

Any types declared within a namespace scope have the namespace name prepended to the type name. For example, a class named "MyClass" that is declared within a namespace named "MyNamespace" will have a type name of "MyNamespace.MyClass".

Types declared outside any namespace scope are declared in the "global" or "unnamed" namespace.

BEGIN NAMESPACE ... END NAMESPACE blocks can be nested to any depth. Nested namespace names have the enclosing namespace name prepended to it, separated by a period.

The same namespace name may be declared multiple times in the same or different files. The **BEGIN NAMESPACE** and **END NAMESPACE** statements do not cause any code to be generated, they simply affect the name of any types declared within the namespace block.

using directives that appear within a namespace are only in effect within the enclosing namespace block, and any nested namespace blocks.

Compatibility Note:

Code migrated from Visual Objects to Vulcan.NET using the Transporter is not placed within any BEGIN NAMESPACE ... END NAMESPACE blocks, because Visual Objects has no concept of namespaces. Therefore, all classes in transported code are in the global or "unnamed" namespace and do not have a namespace name prepended to them.

Example

```
BEGIN NAMESPACE a

    CLASS one          // actual type name is 'a.one'
        ...
    END CLASS
```

```
        BEGIN NAMESPACE b    // the namespace name is 'a.b'

            CLASS two        // actual type name is 'a.b.two'
                ...
            END CLASS

        END NAMESPACE

    END NAMESPACE

    CLASS three              // actual type name is 'three'
        ...
    END CLASS
```

1.8.4.2.5 BEGIN SCOPE

Purpose

The **BEGIN SCOPE** and **END SCOPE** keyword pairs declare a scope of visibility and lifetime of LOCAL variables

Syntax

```
BEGIN SCOPE
    statements
END SCOPE
```

Arguments

statements   Code including one or more LOCAL declarations.

Remarks

BEGIN SCOPE...**END SCOPE** are used within a function/member body to define an area of restricted scope for LOCAL variables. Attempt to use a LOCAL variable that is declared with a BEGIN **SCOPE...END SCOPE** block outside the scope results in a compiler error.

Example

```
FUNCTION Test() AS VOID
BEGIN SCOPE
LOCAL n AS INT
n++
? n
END SCOPE
// n does not exist here
```

1.8.4.2.6 BEGIN SEQUENCE

Purpose

The **BEGIN SEQUENCE** keyword declares the beginning of an exception handling block.

Syntax

```
BEGIN SEQUENCE
    tryStatements
[RECOVER [USING localVariable]
    recoveryStatements
]
[FINALLY
    finallyStatements
]
END [SEQUENCE]
```

where:

tryStatements	One or more statements or expressions that may cause an exception to be thrown.
localVariable	A local variable that will receive the exception thrown by any code between the BEGIN SEQUENCE and RECOVER statements. This must be a variable typed as USUAL.
recoveryStatements	One or more statements or expressions that will execute if an exception is thrown by any of the tryStatements.
finallyStatements	One or more statements or expressions that will always execute regardless of whether an exception is thrown or not.

Remarks

BEGIN SEQUENCE ... END SEQUENCE is a control structure used for exception and runtime error handling. It delimits a block of statements defining a discrete operation, including invoked procedures and functions.

When an exception is thrown anywhere anywhere in the block of statements following the BEGIN SEQUENCE statement up to the corresponding RECOVER statement, control branches to the program statement immediately following the RECOVER statement. If a RECOVER statement is not specified, control branches to the statement following the FINALLY statement, terminating the sequence. If a FINALLY statement is not specified, control branches to the statement following the END SEQUENCE statement, terminating the sequence.

If control reaches a RECOVER statement without an exception being thrown, control branches to the statement following the FINALLY statement. If a FINALLY statement is not specified, control branches to the statement following the END SEQUENCE statement, terminating the sequence.

The RECOVER statement optionally receives an exception thrown by a statement in the tryStatements block. This is usually an error object, generated and returned by the current error handling block defined by ErrorBlock(). If an error object is returned, it can be sent messages to query information about the error. With this information, a runtime error can be handled within the context of the operation rather than in the current runtime error handler.

The FINALLY statement block is useful for cleaning up any resources allocated in the BEGIN SEQUENCE block. Control is always passed to the FINALLY block (if present) regardless of how the BEGIN SEQUENCE block exits.

You cannot RETURN, LOOP, or EXIT between a BEGIN SEQUENCE and RECOVER statement. From within the RECOVER and FINALLY statement blocks however, you can LOOP, EXIT, BREAK, or RETURN since the sequence is essentially completed at that point. Using LOOP from within the RECOVER statement block is useful for re-executing the sequence statement block.

BEGIN SEQUENCE ... END SEQUENCE control structures can be nested to any depth. The CanBreak() function returns TRUE if execution is within any BEGIN SEQUENCE block.

Example

```
FUNCTION Start() AS VOID
LOCAL x := 4, y := 0 AS INT
BEGIN SEQUENCE
? x / y
RECOVER
? "oops"
FINALLY
? "in finally block"
END SEQUENCE
RETURN
```

See Also

[BREAK](#)
[THROW](#)
[TRY-CATCH-FINALLY](#)

1.8.4.2.7 BEGIN UNSAFE

Purpose

The **BEGIN UNSAFE** and **END UNSAFE** keyword pairs declare a scope of code that contains unsafe statements, such as typed pointers.

Syntax

```
BEGIN UNSAFE
    statements
END UNSAFE
```

Arguments

statements   Code including one or more statements that may contain unsafe code.

Example

```
FUNCTION Start() AS VOID
LOCAL a AS INT[]
a := <INT>{1,2,3,4,5}

BEGIN UNSAFE
   LOCAL p AS INT PTR
   p := @a
   FOR VAR i := 1 to 5
      ? p[i]
   NEXT
END UNSAFE
RETURN
```

1.8.4.2.8 BEGIN USING

Purpose

The **BEGIN USING** and **END USING** keyword declare a block of code that ensures the correct use of disposable objects.

Syntax

```
BEGIN USING declaration
    statements
END USING
```

Arguments

declaration   Declaration of a variable and assignment that
statements   Code including one or more statements that may contain
                               unsafe code.

Description

When the lifetime of an IDisposable object is limited to a single method, you should declare and instantiate it in the using statement. The using statement calls the Dispose method on the object in the correct way, and (when you use it as shown earlier) it also causes the object itself to go out of scope as soon as Dispose is called. Within the using block, the object is read-only and cannot be modified or reassigned.

Example

```
BEGIN USING VAR oTest := Test{}
    oTest:DoSomething()
END USING
```

this is the equivalent of

```
VAR oTest := Test{}
TRY
    oTest:DoSomething()
```

```
FINALLY
   IF oTest != NULL_OBJECT
      ((IDisposable)oTest):Dispose()
   ENDIF
END TRY
```

1.8.4.2.9 LOCAL FUNCTION

Purpose

Declare a local function

Syntax

[Modifiers] **LOCAL FUNCTION** *<idFunction>*
[Typeparameters]
[([*<idParam>* [AS | REF|OUT|IN *<idType>*] [, ...])]
[AS *<idType>*]
[TypeparameterConstraints]
[=> <expression>]
CRLF
[*<Body>*]
END FUNCTION

Arguments

[Modifiers]	The only valid modifiers for a local function are UNSAFE and/or ASYNC
<idFunction>	A valid identifier name for the function. A function is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a function and a class, for example, with the same name.
TypeParameters	This is supported for methods with generic type arguments. This something like <T> for a method with one type parameter named T. Usually one of the parameters in the parameter list is then also of type T.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS	REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be

passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references.
The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters.
Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]

AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
TypeParameterConstraints	Here you can specify constraints for the Type parameters, such as WHERE T IS SomeName or WHERE T IS New
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body
END FUNCTION	These **mandatory** keywords indicate the logical end of the function.

Description

A local function is defined as a nested function inside a containing member. The END FUNCTION is mandatory so the compiler knows where the function ends and its surrounding container continues.
In the example below the WAIT command is part of the Start() function and will be executed after result of the call to Fact() is shown.

Example

```
FUNCTION Start AS VOID
    ? Fact(10)
    LOCAL FUNCTION Fact(nNum AS LONG) AS LONG
        IF nNum == 1
            RETURN 1
        ENDIF
        RETURN nNum * Fact(nNum-1)
    END FUNCTION
    WAIT
    RETURN
```

See Also

FIELD, LOCAL, MEMVAR, METHOD, PROCEDURE, RETURN, FUNCTION

1.8.4.2.10 LOCAL PROCEDURE

Purpose

Declare a local procedure

Syntax

[Modifiers] **LOCAL PROCEDURE** *<idFunction>*
[Typeparameters]
[([*<idParam>* [AS | REF|OUT|IN *<idType>*] [, ...])]
[TypeparameterConstraints]
[=> <expression>]
CRLF
[*<Body>*]
END PROCEDURE

Arguments

[Modifiers]	The only valid modifiers for a local function are UNSAFE and/or ASYNC
<idFunction>	A valid identifier name for the function. A function is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a function and a class, for example, with the same name.

TypeParameters	This is supported for methods with generic type arguments. This something like <T> for a method with one type parameter named T. Usually one of the parameters in the parameter list is then also of type T.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references.
	The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters.
	Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
TypeParameterConstraints	Here you can specify constraints for the Type parameters, such as WHERE T IS SomeName or WHERE T IS New
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity.
	The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed).
	CANNOT be combined with an Expression Body
END PROCEDURE	These **mandatory** keywords indicate the logical end of the function.

Description

A local function is defined as a nested function inside a containing member. The END PROCEDURE is mandatory so the compiler knows where the function ends and its surrounding container continues.
In the example below the WAIT command is part of the Start() function and will be executed after the call to Log(3)

Example

```
FUNCTION Start AS VOID
    Log(1)
    Log(2)
    Log(3)

    LOCAL PROCEDURE Log(nNum AS LONG)
        ? nNum
        RETURN
    END PROCEDURE
    WAIT


    RETURN
```

See Also

[FIELD](), [LOCAL](), [MEMVAR](), [METHOD](), [PROCEDURE](), [RETURN](), [PROCEDURE]()

1.8.4.2.11 USING

The **USING** statement allows the types in a namespace to be used without having to specify the fully-qualified type name.

```
using namespaceName
using alias := namespaceName
using static typeName
```

Arguments

namespaceName	Specifies a fully-qualified namespace name that the compiler will use when attempting to resolve type names.
alias	An alias for the namespace.

typeName	Specifies the name of the type whose static members and nested types can be referenced without specifying a type na

All types in .NET have fully-qualified names consisting of a namespace name and a type name. For example, "System.Windows.Forms.MessageBox" refers to a type named "MessageBox" (a class, in this case) in the "System.Windows.Forms" namespace.

The use of namespaces in .NET helps prevent naming conflicts between class libraries. However, it can be cumbersome to type the fully qualified name. The using statement instructs the compiler to treat the specified namespace as if it were part of the current namespace, for purposes of type resolution. This allows you to specify partial type names in your source code, rather than fully-qualified names. You can, of course, always use a fully-qualified name regardless of any using statements present.

Note that the using statement only imports the type names in the specified namespace; it does not import names in nested or parent namespaces.

Also note that every source file has an implied **using System** and **using XSharp** directive, since the System and XSharp namespace contain classes that virtually every application will use. Explicitly specifying **using System** or **using XSharp** is allowed, but unnecessary.

Examples

```
USING System.Windows.Forms

FUNCTION Start() AS VOID
    LOCAL dlg1 AS OpenFileDialog // error, without the using
statement
    LOCAL dlg2 AS System.Windows.Forms.OpenFileDialog // always ok,
but cumbersome to type

    dlg1 := OpenFileDialog{} // error, without the using statement
    dlg2 := System.Windows.Forms.OpenFileDialog{} // always ok, but
cumbersome to type
    ...
    RETURN
```

or

```
USING SWF := System.Windows.Forms

FUNCTION Start() AS VOID
    LOCAL dlg1 AS SWF.OpenFileDialog // use the alias in stead of
the full name

    dlg1 := SWF.OpenFileDialog{}
```

```
        ...
        RETURN
```

or

```
    USING STATIC System.Console
    FUNCTION Start as VOID
        WriteLine("Hello world")        // this calls
    System.Console.WriteLine()
        ReadLine()                      // this calls
    System.Console.ReadLine()
        RETURN
```

1.8.4.3   Comment

NOTE

1.8.4.3.1 Comments

Purpose

X# has many comment formats.

Syntax

```
/*  this is a multiline
    comment */
// this is a single line comment
&& this is a single line comment. It may appear after other
statements on a line
* this is also a comment. the * must be the first non whitespace
character on the line
```

Description

X# has many different comment formats

Examples

These examples show the various comment symbols:

```
// This is a comment
/* This is a comment */
* This is a comment
&& This is a comment
```

1.8.4.4 Concurrency Control

COMMIT
SET EXCLUSIVE
UNLOCK
USE

1.8.4.4.1 COMMIT Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Flush all pending updates in all work areas.

Syntax

```
COMMIT [ALL]
```

Description

COMMIT causes all pending updates to all work areas to be written to disk. It is functionally equivalent to DBCommit() for every occupied work area.
COMMIT **ALL** is functionally equivalent to DBCommitAll().
All updated database and order buffers are written to disk, and a OS Commit request is issued for all files associated with all work areas.

Notes

Shared mode: COMMIT makes database updates visible to other processes. To insure data integrity, issue DBCommit() before an unlock operation.

Examples

In this example, COMMIT forces a write to disk after a series of memory variables are assigned to field variables:

```
USE sales EXCLUSIVE NEW

APPEND BLANK
REPLACE Sales->Name WITH "Jones"
REPLACE Sales->Amount WITH 123.45
COMMIT
```

Assembly

XSharp.RT.DLL

See Also

DBCommit(), DBCommitAll(), GO, SKIP

1.8.4.4.2 SET EXCLUSIVE Command

Note This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines whether to open database files in exclusive or shared mode.

Syntax

```
SET EXCLUSIVE ON | OFF | (<lToggle>)
```

Arguments

ON	Limits accessibility of a table opened on a network to the user who opened it. The table isn't accessible to other users on the network. Unlike FLOCK( ), SET EXCLUSIVE ON also prevents all other users from having read-only access. A file can also be opened on a network for exclusive use by including the EXCLUSIVE clause with the USE command. It

isn't necessary to perform record or file locking on a table opened for exclusive use.

Opening a table for exclusive use ensures that the file can't be changed by other users. For some commands, execution isn't possible until a table is opened for exclusive use. These commands are PACK, REINDEX, and ZAP.

OFF	Allows a table opened on a network to be shared and modified by any user on the network.
lToggle	A logical expression which must appear in parentheses. True is equivalent to ON, False to OFF

Description

SET EXCLUSIVE is functionally equivalent to SetExclusive().

Changing the setting of SET EXCLUSIVE doesn't change the status of previously opened tables. For example, if a table is opened with SET EXCLUSIVE set to ON, and SET EXCLUSIVE is later changed to OFF, the table retains its exclusive-use status.

Assembly

XSharp.RT.DLL

See Also

FLock(), NetErr(), RLock(), SetExclusive(), USE

1.8.4.4.3 UNLOCK Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Release all locks for a work area.

Syntax

```
UNLOCK    [RECORD nRecordNumber]    [[IN|ALIAS] workarea]    [ALL]
```

Arguments

RECORD nRecordNumber	Releases the record lock on record number nRecordNumber. Issuing UNLOCK RECORD nRecordNumber for a record in a table with a file lock releases the file lock.
IN|ALIAS <workarea>	Specifies the work area for which the operation must be performed

ALL   Releases all record and file locks in all work areas. If not specified, only the lock in the current work area is released. ALL cannot be combined with the RECORD or IN|ALIAS clause.

Description

UNLOCK is functionally equivalent to DBUnlock() and UNLOCK ALL is functionally equivalent to DBUnlockAll().

Examples

This example attempts an update operation that requires a record lock. If the RLock() is successful, the record is updated with a function and the RLock() is released with UNLOCK:

```
USE sales INDEX salesman SHARED NEW
IF RLock()
    UpdateRecord()
    UNLOCK
ELSE
    ? "Record update failed"
    BREAK
ENDIF
```

Assembly

XSharp.RT.DLL

See Also

DbRLock(), DbUnlock(), DbUnlockAll(), FLock(), RLock(), SetExclusive(), USE

1.8.4.5   Database

Most Database commands are implemented as User Defined command in the standard header files, and call Runtime Functions under the hood.
Make sure that you include the standard header file during compilation (so do NOT use the option -nostddefs).
If you use the compiler option -stddefs to use an alternate standard header file then it is your responsibilty to make sure that these commands are implemented.

APPEND BLANK
APPEND FROM
AVERAGE
CLEAR ALL
CLOSE

1.8.4.5.1 APPEND BLANK Command

Purpose

Add a new record to the end of the current database file and make it the current record.

Syntax

```
APPEND BLANK  [[IN|ALIAS] <workarea>]
```

Arguments

IN|ALIAS <workarea>   Specifies the work area for which the operation must be performed

Description

The new field values are initialized to the empty values for each data type: character fields are assigned with spaces; numeric fields are assigned 0; logical fields are assigned FALSE; date fields are assigned NULL_DATE; and memo fields are left empty.

For a shared database, APPEND BLANK automatically places a record lock on the new record. If the record cannot be locked, NetErr() is set to TRUE, indicating that the record was not added, and execution continues.

Note: APPEND BLANK will not release any file locks set by the current process.
If NetErr() returns FALSE, the record was successfully added and locked, you can begin updating it. The newly appended record remains locked until you explicitly release the lock (for example, with UNLOCK), close the database file, or attempt another lock.

Examples

This example attempts to add a record to a shared database file and uses NetErr() to test if the operation succeeded:

```
USE sales SHARED NEW
<Statements>...
APPEND BLANK
IF !NetErr()
   <Update EMPTY record>...
ELSE
   ? "Append operation failed"
```

```
        BREAK
    ENDIF
```

Assembly

XSharp.RT.DLL

See Also

[APPEND FROM](), DBAppend(), DBRLock(), FLock(), NetErr(), RLock()

1.8.4.5.2 APPEND FROM ARRAY Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header ([-nostddefs]) files then this command will not be available. If you tell the compiler to use a different standard header file ([-stddef]) then this command may also be not available

Purpose

Adds one record to the currently selected table for each row in an array and fills each record with data from the corresponding array row.

Syntax

```
APPEND FROM ARRAY <ArrayName> [FOR <lCondition>]
 [FIELDS <idFieldList> | FIELDS LIKE <Skeleton> | FIELDS EXCEPT
<Skeleton>]
```

Arguments

<ArrayName>	Specifies the name of the array that contains the data to be copied to the new records. New records are added to the table until all rows in the array are appended.
FIELDS <idFieldList>	The list of fields to process. The default is all fields with the exception of memo fields, unless the command supports the MEMO clause. Only fields with the same names and types in both files are appended. If fields with the same name do not match in data type, a runtime error is raised.
FIELDS LIKE <Skeleton>	You can specify field names with a wild card, such as FIELDS LIKE *name
FIELDS EXCEPT <Skeleton>	You can exclude fields, such as for example the primary keys: FIELDS EXCEPT Id

<Skeleton> supports wildcards (* and ?). For example, to replace all fields that begin with the letters A and P, use:
`FIELDS LIKE A*,P*`

Please note that you can combine FIELDS LIKE and FIELDS EXCEPT but you cannot combine a fields list with the LIKE and EXCEPT clauses.

FOR <lCondition>   A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.

See Also

[COPY TO ARRAY](#), [GATHER](#), [SCATTER](#)

1.8.4.5.3   APPEND FROM Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header ([-nostddefs](#)) files then this command will not be available. If you tell the compiler to use a different standard header file ([-stddef](#)) then this command may also be not available

Purpose

Import records from a database or text file.

Syntax

```
APPEND FROM <xcSourceFile> [FIELDS <idFieldList>]
[<Scope>] [WHILE <lCondition>] [FOR <lCondition>]
[SDF] | [DELIMITED [WITH <xcDelimiter> | BLANK]] |
[VIA <cDriver>] [INHERIT FROM <acRDDs>]
```

Arguments

FROM <xcSourceFile>   The name of the source file from which to add records, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension for database files is determined by the RDD. For text files, it is .TXT.
This command attempts to open <xcSourceFile> in shared mode. If the file does not exist, a runtime error is raised. If the file is successfully opened, the operation proceeds. If access is denied because, for example, another process has exclusive use of the file, NetErr() is set to TRUE.

FIELDS <idFieldList>	The list of fields to append from <xcSourceFile>. The default is all fields. Only fields with the same names and types in both files are appended. If fields with the same name do not match in data type, a runtime error is raised.
<Scope>	The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of:

[NEXT <NEXT>]　　　　　　Optionally specifies the number of records to process starting

　　　　　　　　　　　　　　　with the first record of the source file.

[RECORD <rec>]　　　　　An optional record ID If specified, the processing begins

　　　　　　　　　　　　　　　with this data record in the source file.

[<rest:REST>]　　　　　　The option REST specifies whether records are sequentially

　　　　　　　　　　　　　　　searched only from the current up to the last record.

　　　　　　　　　　　　　　　If a condition is specified, the option ALL is the default value.

[ALL]　　　　　　　　　　　The option ALL specifies that all records from the source file are imported.

　　　　　　　　　　　　　　　This is the default setting.

WHILE <lCondition>	A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.
FOR <lCondition>	A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.
SDF	A System Data Format file with format specifications as shown in the table below. Records and fields are fixed length.

File Element	Format
Character fields	Padded with trailing blanks
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Padded with leading blanks or zeros
Field separator	None

Record separator	Carriage return/linefeed
End of file marker	1A hex or Chr(26)

DELIMITED [WITH <xcDelimiter>]

A text file in which character fields are enclosed in double quote marks (the default delimiter) or the specified <xcDelimiter>. Fields and records are variable length, and the format specifications are shown in the table below:

File Element	Format
Character fields	Can be delimited, with trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros can be truncated
Field separator	Comma
Record separator	Carriage return/linefeed
End of file marker	1A hex or Chr(26)

Note: Delimiters are not required and APPEND FROM correctly handles character fields not enclosed with them.

DELIMITED WITH BLANK

A text file in which fields are separated by one space and character fields are not enclosed in delimiters. The format specifications are shown in the table below:

File Element	Format
Character fields	Not delimited, trailing blanks can be truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros can be truncated
Field separator	Single blank space
Record separator	Carriage return/linefeed
End of file marker	1A hex or Chr(26)

Warning! If the DELIMITED WITH clause is specified on an APPEND FROM command line, it must be the last clause specified.

VIA <cDriver>   The name of the RDD that will service the work area. If not specified, the default RDD as determined by RDDSetDefault() is used.

INHERIT FROM <acRDDs>

A one-dimensional array with the names of RDDs from which the main RDD inherits special functionality. This allows you to use RDDs with special capabilities, like encryption or decryption, in different work areas with different database drivers. These RDDs overlay special functions of the main RDD (specified with the VIA clause). If multiple RDDs (specified with this INHERIT FROM clause) implement the same function, the function associated with the last RDD in the list takes precedence.

Notes

Deleted records: If SetDeleted() is FALSE, deleted records in *<xcSourceFile>* are appended to the current database file and retain their deleted status. If SetDeleted() is TRUE, however, deleted records are not visible and are, therefore, not processed.

Unmatched field widths: If a field in the current database file is character type and has a field length greater than the incoming *<xcSourceFile>* data, APPEND FROM pads the source data with blanks. If the current field is character data type and its field length is less than the incoming source data, the source data is truncated to fit. If the current field is numeric type and the incoming source data has more digits than the current field length, a runtime error is raised.

Examples

This example demonstrates an APPEND FROM command using a fields list and a condition:

```
USE sales NEW
APPEND FROM branchfile FIELDS Branch, Salesman, Amount FOR Branch
= 100
```

The next example demonstrates how a *<Scope>* can be specified to import a particular record from another database file:

```
APPEND RECORD 5 FROM temp
```

Assembly

XSharp.RT.DLL

See Also

COPY TO, DbApp(), DbAppDelim(), DbAppSDF(), RDDSetDefault(), SetDefault(), SetPath(), SetDeleted()

1.8.4.5.4 APPEND FROM Command (FoxPro)

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Import records from a database or text file.

Syntax

```
APPEND FROM <xcSourceFile>
   [FIELDS <idFieldList> | FIELDS LIKE <Skeleton> | FIELDS EXCEPT
<Skeleton>]
   [<Scope>] [WHILE <lCondition>] [FOR <lCondition>]
   [[TYPE] [DELIMITED [WITH <Delim> | WITH BLANK | WITH TAB
       | WITH CHARACTER <cDelim>] | DIF | FW2 | MOD | PDOX | RPD |
      SDF | SYLK | WK1 | WK3 | WKS | WR1 | WRK | CSV | XLS | XL5
      [SHEET <cSheetName>] | XL8 [SHEET <cSheetName>]]] [AS
<nCodePage>]
```

Arguments

FROM <xcSourceFile>	The name of the source file from which to add records, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension for database files is determined by the RDD. For text files, it is .TXT. This command attempts to open <xcSourceFile> in shared mode. If the file does not exist, a runtime error is raised. If the file is successfully opened, the operation proceeds. If access is denied because, for example, another process has exclusive use of the file, NetErr() is set to TRUE.
FIELDS <idFieldList>	The list of fields to process. The default is all fields with the exception of memo fields, unless the command supports the MEMO clause. Only fields with the same names and types in both files are appended. If fields with the same name do not match in data type, a runtime error is raised.
FIELDS LIKE <Skeleton>	You can specify field names with a wild card, such as FIELDS LIKE *name
FIELDS EXCEPT <Skeleton>	You can exclude fields, such as for example the primary keys: FIELDS EXCEPT Id <Skeleton> supports wildcards (* and ?). For example, to replace all fields that begin with the letters A and P, use: FIELDS LIKE A*,P*

Please note that you can combine FIELDS LIKE and FIELDS EXCEPT but you cannot combine a fields list with the LIKE and EXCEPT clauses.

<Scope>	The portion of the current database file to process. The

default is all visible records. Scope is one or more clauses of:

[NEXT <NEXT>]   Optionally specifies the number of records to process starting

        with the first record of the source file.

[RECORD <rec>]   An optional record ID If specified, the processing begins

        with this data record in the source file.

[<rest:REST>]   The option REST specifies whether records are sequentially

        searched only from the current up to the last record.

        If a condition is specified, the option ALL is the default value.

[ALL]   The option ALL specifies that all records from the source file are imported.

        This is the default setting.

WHILE <lCondition>	A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.
FOR <lCondition>	A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.
DELIMITED WITH <Delim>	Indicates that character fields are separated by a character other than the quotation mark.
DELIMITED WITH BLANK	Specifies files that contain fields separated by spaces instead of commas.
DELIMITED WITH TAB	Specifies files that contain fields separated by tabs rather than commas.
WITH CHARACTER <cDelim>	Specifies files that contain fields all enclosed by the character specified with Delimiter. If Delimiter is a semicolon (the character used in Visual FoxPro to indicate command line continuation), enclose the semicolon in quotation marks. You can also specify the BLANK and TAB keywords for Delimiter. The WITH Delimiter clause can be combined with the WITH CHARACTER clause.
TYPE	From the various types that FoxPro allows only the following ones are supported in X# at this moment: SDF An SDF file is an ASCII text file in which records have a fixed length and end with a carriage return and line

feed. Fields are not delimited.
    The file name extension is assumed to be .txt for SDF files.

CSV   A comma separated value file. A CSV file has field names as the first line in the file; the field names are ignored when the file is imported.
    The file name extension is assumed to be .csv for CSV files.

OTHER   **NOT SUPPORTED AT THIS MOMENT**

AS <nCodePage>   Specifies the codepage to use for the source file. **NOT SUPPORTED AT THIS MOMENT**

Notes

Deleted records: If SetDeleted() is FALSE, deleted records in <*xcSourceFile*> are appended to the current database file and retain their deleted status. If SetDeleted() is TRUE, however, deleted records are not visible and are, therefore, not processed.

Unmatched field widths: If a field in the current database file is character type and has a field length greater than the incoming <*xcSourceFile*> data, APPEND FROM pads the source data with blanks. If the current field is character data type and its field length is less than the incoming source data, the source data is truncated to fit. If the current field is numeric type and the incoming source data has more digits than the current field length, a runtime error is raised.

Examples

This example demonstrates an APPEND FROM command using a fields list and a condition:

```
USE sales NEW
APPEND FROM branchfile FIELDS Branch, Salesman, Amount FOR Branch
= 100
```

The next example demonstrates how a <*Scope*> can be specified to import a particular record from another database file:

```
APPEND RECORD 5 FROM temp
```

Assembly

XSharp.RT.DLL

See Also

<u>COPY TO</u>, DbApp(), DbAppDelim(), DbAppSDF(), RDDSetDefault(), SetDefault(), SetPath(), SetDeleted()

1.8.4.5.5 AVERAGE Command

Purpose

Calculate the average of one or more numeric expressions to variables for a range of records in the current database file.

Syntax

```
AVERAGE <nValueList> TO <idVarList> [<Scope>] [WHILE <lCondition>]
[FOR <lCondition>]
```

Arguments

<nValueList>	A list of the numeric values to average for each record processed.
TO *<idVarList>*	A list of receiving variable or field names which will contain the average results. Variables that either do not exist or are not visible are created as private variables. *<idVarList>* must contain the same number of elements as *<nValueList>*.

<Scope>	The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of:
	[NEXT <NEXT>] Optionally specifies the number of records to process starting
	                                          with the first record of the source file.
	[RECORD <rec>] An optional record ID If specified, the processing begins
	                                          with this data record in the source file.
	[<rest:REST>] The option REST specifies whether records are sequentially
	                                          searched only from the current

up to the last record.

If a condition is specified, the option ALL is the default value.

[ALL] The option ALL specifies that all records from the source file are imported. This is the default setting.

WHILE <lCondition> A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.

FOR <lCondition> A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.

Description

Zero (0) values are counted in the AVERAGE unless explicitly ruled out with a FOR condition.

Examples

This example averages a single numeric field using a condition to select a subset of records from the database file:

```
USE sales NEW
AVERAGE Amount TO nAvgAmount FOR Branch = "100"
```

The next example finds the average date for a range of dates:

```
AVERAGE (SaleDate - CToD("00/00/00")) ;

   TO nAvgDays FOR !Empty(SaleDate)
   dAvgDate := CToD("00/00/00") + nAvgDays
```

Assembly

XSharp.RT.DLL

See Also

<u>COUNT</u>, DBEval(), <u>SUM</u>, <u>TOTAL</u>

1.8.4.5.6 CLEAR ALL Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (<u>-nostddefs</u>) files then this command will not be available. If you tell the compiler to use a different standard header file (<u>-stddef</u>) then this command may also be not available

Purpose

Release all public and private variables, close all open database, index, alternate, and memo files in all active work areas, and select work area 1.

Note: CLEAR ALL is a compatibility command and is no longer recommended. It is superseded by the command or function that performs the specific action you need. See the list below.

Syntax

```
CLEAR ALL
```

Notes

Declared Variables: CLEAR ALL does not release declared variables or constants.
You can close files associated with work areas with one of the various forms of the CLOSE command. You can release private and public variables using the RELEASE command although explicitly releasing variables is discouraged in most cases.

Assembly

XSharp.RT.DLL

See Also

<u>CLEAR MEMORY</u>, <u>CLOSE</u>, DBCloseArea(), FClose(), <u>RELEASE</u>, Select(),

1.8.4.5.7 CLOSE Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (<u>-nostddefs</u>) files then this command will not be available. If you tell the compiler to use a different standard header file (<u>-stddef</u>) then this command may also be not available

Purpose

Close specified file type.

Syntax

```
CLOSE [<xcAlias> | ALL | ALTERNATE | DATABASES | INDEXES]
```

Arguments

<xcAlias>	The alias identifier for the work area where all files are closed.
ALL	Closes database and index files in all work areas, releasing all active filters, relations, and format definitions.
ALTERNATE	Closes the currently open alternate file.
DATABASES	Closes all database, memo, and associated index files in all work areas, and releases all active filters and relations. It does not, however, have any effect on the active format.
INDEXES	Closes all index files and clears the order list in the current work area.

Description

CLOSE with no option closes the current database and associated files. A number of other operations close files as a side effect, but it is always a good idea to explicitly close files when you are finished using them.

Assembly

XSharp.RT.DLL

See Also

DBCloseArea(), FClose(), SET INDEX, USE

1.8.4.5.8 CONTINUE Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Resume a pending locate.

Syntax

```
CONTINUE
```

Description

CONTINUE searches from the current record position for the next record meeting the most recent locate condition executed in the current work area. (You can set the locate condition with the LOCATE command or with the VODBLocate() or VODBSetLocate() functions.)

The search terminates when a match is found or end of file is encountered. If CONTINUE is successful, the matching record becomes the current record and Found() returns TRUE; if unsuccessful, Found() returns FALSE.
Each work area can have an active locate condition. A locate condition remains pending until a new one is specified.

Notes

Scope and WHILE condition: The scope and WHILE condition of the locate condition are ignored; only the for condition is used with CONTINUE. If you are using a LOCATE command with a while condition and want to continue the search for a matching record, use SKIP and then repeat the original LOCATE command, adding REST as the scope.

Examples

This example scans records in SALES.DBF for a particular salesman and displays a running total sales amount:

```
LOCAL nRunTotal := 0
USE sales NEW
LOCATE FOR Sales->Salesman = "1002"
DO WHILE Found()
   ? Sales->SalesName, nRunTotal += Sales->Amount
   CONTINUE
ENDDO
```

This example demonstrates how to continue if the pending LOCATE scope contains a WHILE condition:

```
LOCAL nRunTotal := 0
USE sales INDEX salesman NEW
SEEK "1002"
LOCATE REST WHILE Sales->Salesman = "1002";
        FOR Sales->Amount > 5000
DO WHILE Found()
        ? Sales->Salesname, nRunTotal += Sales->Amount
        SKIP
        LOCATE REST WHILE Sales->Salesman = "1002";
            FOR Sales->Amount > 5000
ENDDO
```

Assembly

XSharp.RT.DLL

See Also

DbContinue(), DbLocate(), EoF(), Found(), LOCATE, SEEK

1.8.4.5.9 COPY STRUCTURE Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Create an empty database file with field definitions from the current database file.

Syntax

```
COPY STRUCTURE [FIELDS <idFieldList>] TO <xcTargetFile>
```

Arguments

FIELDS <idFieldList>   The set of fields to copy to the new database structure in the order specified. The default is all fields.

TO <xcTargetFile>   The name of the target database file, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension for database files is determined by the RDD .

If <xcTargetFile> does not exist, it is created. If it exists, this command attempts to open the file in exclusive mode and, if successful, the file is overwritten without warning or error. If access is denied because, for example, another process is using the file, NetErr() is set to TRUE.

Examples

In this example, COPY STRUCTURE creates a temporary file. After the user enters data into the temporary file, the master database file is updated with the new information:

```
USE sales NEW
COPY STRUCTURE TO temp
USE temp NEW
lMore := TRUE
DO WHILE lMore
   APPEND BLANK
   @ 10, 10 GET Temp->Salesman
   @ 11, 11 GET Temp->Amount
   READ
   IF Updated()
      SELECT sales
      APPEND BLANK
      REPLACE Sales->Salesman WITH Temp->Salesman
      REPLACE Sales->Amount WITH Temp->Amount
      SELECT Temp
      ZAP
   ELSE
      lMore := FALSE
   ENDIF
ENDDO
CLOSE DATABASES
```

Assembly

XSharp.RT.DLL

See Also

COPY STRUCTURE EXTENDED, CREATE, DbCopyStruct(), SetDefault(), SetPath()

1.8.4.5.10 COPY STRUCTURE EXTENDED Command

| Note | This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available |

Purpose

Copy the field definitions in a database file structure to a structure-extended file as data.

Syntax

```
COPY STRUCTURE EXTENDED TO <xcTargetFile>
```

Arguments

TO <xcTargetFile>   The name of the target structure-extended database file, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension for database files is determined by the RDD .

If <xcTargetFile> does not exist, it is created. If it exists, this command attempts to open the file in exclusive mode and, if successful, the file is overwritten without warning or error. If access is denied because, for example, another process is using the file, NetErr() is set to TRUE.

Description

COPY STRUCTURE EXTENDED creates a database file whose contents is the structure of the current database file, with a record for the definition of each field.
The structure-extended database file has the following structure:

Field Name	Type	Length	Decimals
Field_Name	Character	10	
Field_Type	Character	1	
Field_Len	Numeric	3	0
Field_Dec	Numeric	3	0

Used in application programs, COPY STRUCTURE EXTENDED permits you to create or modify the structure of a database file programmatically. To create a new database file from the structure-extended file, use CREATE FROM. If you need an empty structure-extended file, use CREATE.

Notes

Character field lengths greater than 255: Field lengths greater than 255 are represented as a combination of the Field_Dec and Field_Len fields. After performing COPY STRUCTURE EXTENDED, you can use the following formula to determine the length of any character field:

```
nFieldLen := IIf((Field_Type = "C" .AND.  ;
        Field_Dec != 0), Field_Dec * 256 + ;
        Field_Len, Field_Len)
```

Examples

This example creates STRUC.DBF from SALES.DBF as a structure-extended file, then lists the contents of STRUC.DBF to illustrate the typical layout of field definitions:

```
USE sales NEW
COPY STRUCTURE EXTENDED TO struc
USE struc NEW
LIST Field_Name, Field_Type, Field_Len, Field_Dec


// Result:
1 BRANCH              C        3        0
2 SALESMAN              C        4        0
3 CUSTOMER              C        4        0
4 PRODUCT               C        25       0
5 AMOUNT              N        8        2
6 NOTES               C        0        125
// Field length is 32,000 characters
```

Assembly

XSharp.RT.DLL

See Also

COPY STRUCTURE, CREATE, CREATE FROM, FieldName(), DbCopyXStruct(), SetDefault(), SetPath(), Type()

1.8.4.5.11 COPY TO ARRAY Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Copies data from the currently selected table to an array.

Syntax

```
COPY TO ARRAY <ArrayName>
    [FIELDS FieldList | FIELDS LIKE <Skeleton> | FIELDS EXCEPT
<Skeleton>]
    [<Scope>] [WHILE <lCondition>] [FOR <lCondition>] [NOOPTIMIZE]
```

Arguments

<ArrayName>	Specifies the name of the array that contains the data to be copied to the new records. New records are added to the table until all rows in the array are appended.
FIELDS <idFieldList>	The list of fields to process. The default is all fields with the exception of memo fields, unless the command supports the MEMO clause. Only fields with the same names and types in both files are appended. If fields with the same name do not match in data type, a runtime error is raised.
FIELDS LIKE <Skeleton>	You can specify field names with a wild card, such as FIELDS LIKE *name
FIELDS EXCEPT <Skeleton>	You can exclude fields, such as for example the primary keys: FIELDS EXCEPT Id <Skeleton> supports wildcards (* and ?). For example, to replace all fields that begin with the letters A and P, use: FIELDS LIKE A*,P*

Please note that you can combine FIELDS LIKE and FIELDS EXCEPT but you cannot combine a fields list with the LIKE and EXCEPT clauses.

<Scope>	The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of:
	[NEXT <NEXT>] Optionally specifies the number of records to process starting
	                                            with the first record of the source file.
	[RECORD <rec>] An optional record ID If specified, the processing begins
	                                            with this data record in the source file.
	[<rest:REST>] The option REST specifies whether records are sequentially
	                                            searched only from the current up to the last record.
	                                            If a condition is specified, the option ALL is the default value.
	[ALL] The option ALL specifies that

all records from the source file are imported.

This is the default setting.

WHILE <lCondition>	A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.
FOR <lCondition>	A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.
NOOPTIMIZE	This clause is parsed but not yet supported.

See Also

[APPEND FROM ARRAY](#), [GATHER](#), [SCATTER](#)

1.8.4.5.12 COPY TO Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Export records to a new database or text file.

Syntax

```
COPY TO <xcTargetFile> [FIELDS <idFieldList>] [<Scope>]
    [WHILE <lCondition>] [FOR <lCondition>]
    [SDF] | [DELIMITED [WITH BLANK | <xcDelimiter>]] |
    [VIA <cDriver>] [INHERIT FROM <acRDDs>]
```

Arguments

TO <xcTargetFile>	The name of the target file, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension for database files is determined by the RDD. For text files, it is .TXT.

If <xcTargetFile> does not exist, it is created. If it exists, this command attempts to open the file in exclusive mode and, if successful, the file is overwritten without warning or error. If access is denied because, for example, another process is using the file, NetErr() is set to TRUE.

FIELDS <idFieldList>

The list of fields to append from <xcSourceFile>. The default is all fields.
Only fields with the same names and types in both files are appended. If fields with the same name do not match in data type, a runtime error is raised.

<Scope>

The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of:

[NEXT <NEXT>]   Optionally specifies the number of records to process starting
   with the first record of the source file.
[RECORD <rec>]   An optional record ID If specified, the processing begins
   with this data record in the source file.
[<rest:REST>]   The option REST specifies whether records are sequentially
   searched only from the current up to the last record.
   If a condition is specified, the option ALL is the default value.
[ALL]   The option ALL specifies that all records from the source file are imported.
   This is the default setting.

WHILE <lCondition>

A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.

FOR <lCondition>

A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.

SDF

A System Data Format file with format specifications as shown in the table below. Records and fields are fixed length.

File Element	Format
Character fields	Padded with trailing blanks
Date fields	yyyymmdd

Logical fields	T or F
Memo fields	Ignored
Numeric fields	Padded with leading blanks for zeros
Field separator	None
Record separator	Carriage return/linefeed
End of file marker	1A hex or Chr(26)

DELIMITED [WITH <xcDelimiter>]

A text file in which character fields are enclosed in double quote marks (the default delimiter) or the specified <xcDelimiter>. Fields and records are variable length, and the format specifications are shown in the table below:

File Element	Format
Character fields	Delimited, with trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros truncated
Field separator	Comma
Record separator	Carriage return/linefeed
End of file marker	1A hex or Chr(26)

DELIMITED WITH BLANK

A text file in which fields are separated by one space and character fields are not enclosed in delimiters. The format specifications are shown in the table below:

File Element	Format
Character fields	Not delimited, trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros truncated
Field separator	Single blank space
Record separator	Carriage return/linefeed
End of file marker	1A hex or Chr(26)

Warning! If the DELIMITED WITH clause is specified on a COPY TO command line, it must be the last clause specified.

VIA <cDriver>

The name of the RDD that will service the work area. If not specified, the default RDD as determined by RDDSetDefault() is used.

INHERIT FROM <acRDDs>

A one-dimensional array with the names of RDDs from which the main RDD inherits special functionality. This allows you to use RDDs with special capabilities, like encryption or decryption, in different work areas with different database drivers. These RDDs overlay special functions of the main RDD (specified with the VIA clause). If

multiple RDDs (specified with this INHERIT FROM clause) implement the same function, the function associated with the last RDD in the list takes precedence.

Notes

Deleted records: If SetDeleted() is FALSE, deleted records in the source file are copied to *<xcTargetFile>* where they retain their deleted status.

Visibility: If SetDeleted() is TRUE, however, deleted records are not visible and are, therefore, not processed. Similarly, filtered records (with DbSetFilter() or a conditional controlling order) are not processed.

Examples

This example demonstrates copying to another database file:

```
USE sales NEW
COPY TO temp
```

This example demonstrates the layout of an SDF file with four fields, one for each data type:

```
USE testdata NEW
COPY NEXT 1 TO temp SDF
TYPE temp.txt
// Result:  Character   12.0019890801T
```

The next example demonstrates the layout of a DELIMITED file:

```
COPY NEXT 1 TO temp DELIMITED
TYPE temp.txt
// Result:  "Character",12.00,19890801,T
```

Finally, this example demonstrates the layout of a DELIMITED file WITH a different delimiter:

```
COPY NEXT 1 TO temp DELIMITED WITH '
TYPE temp.txt
// Result:  'Character',12.00,19890801,T
```

Assembly

XSharp.RT.DLL

See Also

APPEND FROM, COPY FILE, COPY STRUCTURE, DbCopy(), DbCopyDelim(), DbCopySDF(), RDDSetDefault(), SetDefault(), SetPath(), SetDeleted()

1.8.4.5.13 COPY TO Command (FoxPro)

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Export records to a new database or text file.

Syntax

```
COPY TO <xcTargetFile> [DATABASE <DbName> [NAME <LongName>]
   [FIELDS FieldList | FIELDS LIKE <Skeleton> | FIELDS EXCEPT
<Skeleton>]
   [<Scope>] [FOR <lCondition>] [WHILE <lCondition> ]
   [ [WITH] CDX ] | [ [WITH] PRODUCTION ] [NOOPTIMIZE]
   [ [TYPE] [ FOXPLUS | FOX2X | DIF | MOD | SDF | SYLK | WK1 | WKS
| WR1
   | WRK | CSV | XLS | XL5 | DELIMITED [ WITH <Delim> | WITH
BLANK
   | WITH TAB | WITH CHARACTER <cDelim> ] ] ] [AS <nCodePage>]
```

Arguments

TO <xcTargetFile>   The name of the target file, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension for database files is determined by the RDD. For text files, it is .TXT.

                             If <xcTargetFile> does not exist, it is created. If it exists, this command attempts to open the file in exclusive mode and, if successful, the file is overwritten without warning or error. If access is denied because, for example, another process is using the file, NetErr() is set to TRUE.

DATABASE <DbName>	Specifies a database to which the new table is added. **NOT SUPPORTED AT THIS MOMENT**
NAME <LongName>	Specifies a long name for the new table. Long names can contain up to 128 characters and can be used instead of short file names in the database **NOT SUPPORTED AT THIS MOMENT**
FIELDS <idFieldList>	The list of fields to process. The default is all fields with the exception of memo fields, unless the command supports the MEMO clause. Only fields with the same names and types in both files are appended. If fields with the same name do not match in data type, a runtime error is raised.
FIELDS LIKE <Skeleton>	You can specify field names with a wild card, such as FIELDS LIKE *name
FIELDS EXCEPT <Skeleton>	You can exclude fields, such as for example the primary keys: FIELDS EXCEPT Id <Skeleton> supports wildcards (* and ?). For example, to replace all fields that begin with the letters A and P, use: `FIELDS LIKE A*,P*`

Please note that you can combine FIELDS LIKE and FIELDS EXCEPT but you cannot combine a fields list with the LIKE and EXCEPT clauses.

WHILE <lCondition>	A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.

<Scope>	The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of:
	[NEXT <NEXT>] Optionally specifies the number of records to process starting
	            with the first record of the source file.
	[RECORD <rec>] An optional record ID If specified, the processing begins
	            with this data record in the source file.
	[<rest:REST>] The option REST specifies whether records are sequentially
	            searched only from the current up to the last record.
	            If a condition is specified, the option ALL is the default value.
	[ALL] The option ALL specifies that all records from the source file are imported.
	            This is the default setting.

FOR <lCondition>	A condition that each visible record within the scope must meet in order to be processed. If a record does not meet

the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.

DELIMITED WITH <Delim>	Indicates that character fields are separated by a character other than the quotation mark.
DELIMITED WITH BLANK	Specifies files that contain fields separated by spaces instead of commas.
DELIMITED WITH TAB	Specifies files that contain fields separated by tabs rather than commas.
WITH CHARACTER <cDelim>	Specifies files that contain fields all enclosed by the character specified with Delimiter. If Delimiter is a semicolon (the character used in Visual FoxPro to indicate command line continuation), enclose the semicolon in quotation marks. You can also specify the BLANK and TAB keywords for Delimiter. The WITH Delimiter clause can be combined with the WITH CHARACTER clause.
TYPE	Specifies the file type if the file you create is not a XBase table. Although you must specify a file type, you do not need to include the TYPE keyword. From the various types that FoxPro allows only the following ones are supported in X# at this moment: SDF An SDF file is an ASCII text file in which records have a fixed length and end with a carriage return and line feed. Fields are not delimited. The file name extension is assumed to be .txt for SDF files.

CSV   A comma separated value file. A CSV file has field names as the first line in the file; the field names are ignored when the file is imported.
        The file name extension is assumed to be .csv for CSV files.
FOXPLUS   Visual FoxPro memo files have a different structure than FoxBASE memo files.
                If your source table contains a memo field, include the FOXPLUS clause to create a table that can be used in FoxBASE+.
                The Visual FoxPro memo field cannot contain binary data because FoxBASE+ does not support binary data in memo fields.
FOX2X   Creates a new table that can be opened in earlier versions of FoxPro (versions 2.0, 2.5, and 2.6).
OTHER   **NOT SUPPORTED AT THIS MOMENT**

AS <nCodePage>	Specifies the codepage to use for the target file. **NOT SUPPORTED AT THIS MOMENT**

Notes

Deleted records: If SetDeleted() is FALSE, deleted records in the source file are copied to *<xcTargetFile>* where they retain their deleted status.

Visibility: If SetDeleted() is TRUE, however, deleted records are not visible and are, therefore, not processed. Similarly, filtered records (with DbSetFilter() or a conditional controlling order) are not processed.

Examples

This example demonstrates copying to another database file:

```
USE sales NEW
COPY TO temp
```

This example demonstrates the layout of an SDF file with four fields, one for each data type:

```
USE testdata NEW
COPY NEXT 1 TO temp SDF
TYPE temp.txt
// Result:  Character   12.0019890801T
```

The next example demonstrates the layout of a DELIMITED file:

```
COPY NEXT 1 TO temp DELIMITED
TYPE temp.txt
// Result:  "Character",12.00,19890801,T
```

Finally, this example demonstrates the layout of a DELIMITED file WITH a different delimiter:

```
COPY NEXT 1 TO temp DELIMITED WITH '
TYPE temp.txt
// Result:  'Character',12.00,19890801,T
```

Assembly

XSharp.RT.DLL

See Also

APPEND FROM, COPY FILE, COPY STRUCTURE, DbCopy(), DbCopyDelim(), DbCopySDF(), RDDSetDefault(), SetDefault(), SetPath(), SetDeleted()

1.8.4.5.14 COUNT Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Tally the number of records from the current work area that match the specified record scope and condition, and place the result in the specified variable.

Syntax

```
COUNT TO <idVar> [<Scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

TO <idVar>   The variable or field that holds the COUNT result. A variable
                              that either does not exist or is invisible is created as a
                              private variable.

<Scope>   The portion of the current database file to process. The
                              default is all visible records. Scope is one or more clauses
                              of:
                              [NEXT <NEXT>]   Optionally specifies the
                              number of records to process starting
                                                          with the first record of the
                              source file.
                              [RECORD <rec>]   An optional record ID If
                              specified, the processing begins
                                                          with this data record in the
                              source file.
                              [<rest:REST>]   The option REST specifies
                              whether records are sequentially
                                                          searched only from the current
                              up to the last record.

                                                          If a condition is specified, the

option ALL is the default value.

[ALL]   The option ALL specifies that all records from the source file are imported.

This is the default setting.

WHILE <lCondition>   A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.

FOR <lCondition>   A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.

Examples

This example demonstrates a COUNT using a particular Branch in SALES.DBF:

```
USE sales NEW
COUNT TO nBranchCnt FOR Branch = 100
? nBranchCnt                 // Result:  4
```

The next example tallies the number of records in SALES.DBF whose Branch has the value of 100 and assigns the result to the Count field in BRANCH.DBF for branch 100:

```
USE branch INDEX branch NEW
SEEK 100
USE sales INDEX salesbranch NEW
SEEK 100
COUNT TO Branch->Count WHILE Sales->Branch = 100
```

Assembly

XSharp.RT.DLL

See Also

AVERAGE, DBEval(), SUM, TOTAL

1.8.4.5.15 CREATE Command

Purpose

Create an empty structure-extended file.

Syntax

```
CREATE <xcTargetFile> [NEW] [ALIAS <xcAlias>] [VIA <cDriver>]
```

Arguments

<xcTargetFile>
The name of the empty structure-extended database file, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension for database files is determined by the RDD. After the file is created, it remains open in the mode specified by the SetExclusive() flag for the work area.

If <xcTargetFile> does not exist, it is created. If it exists, this command attempts to open the file in exclusive mode and, if successful, the file is overwritten without warning or error. If access is denied because, for example, another process is using the file, NetErr() is set to TRUE.

The structure of <xcTargetFile> is the same as the structure generated by COPY STRUCTURE EXTENDED, which you can refer to for more details.

NEW
Selects the next unoccupied work area before opening <xcTargetFile>. If this clause is not specified, the current work area is used.

ALIAS <xcAlias>
An identifier name to associate with the work area when <xcTargetFile> is opened. If this clause is not specified, the alias defaults to the database file name. Duplicate alias names are not allowed within a single application.

VIA <cDriver>
The name of the RDD that will service the work area. If not specified, the default RDD as determined by RDDSetDefault() is used.

Notes

Unlike COPY STRUCTURE EXTENDED, the file created by CREATE is empty. You must add records to the structure-extended file before you can use it to create a new database file with CREATE FROM.

Examples

This example creates a new structure-extended file, places the definition of one field into it, then creates a new database file from the extended structure:

```
CREATE tempstru
APPEND BLANK
REPLACE Field_Name WITH "Name",;
   Field_Type WITH "C",;
   Field_Len WITH 25,;
   Field_Dec WITH 0
CLOSE
CREATE newfile FROM tempstru
```

Assembly

XSharp.RT.DLL

See Also

COPY STRUCTURE EXTENDED, CREATE FROM, DbCopyStruct(), DbCreate(), RDDSetDefault(),SetDefault(), SetPath(), SetExclusive()

1.8.4.5.16 CREATE FROM Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Produce a new database file with the field definitions taken from the contents of a structure-extended file.

Syntax

```
CREATE <xcTargetFile> FROM <xcSourceFile> [NEW] [ALIAS <xcAlias>] [VIA
<cDriver>]
```

Arguments

<xcTargetFile>   The name of the target database file to create, including an optional drive, directory, and extension. After the file is created, it remains open in the mode specified by the SetExclusive() flag for the work area.

If <xcTargetFile> does not exist, it is created. If it exists, this command attempts to open the file in exclusive mode and, if successful, the file is overwritten without warning or error. If access is denied because, for example, another process is using the file, NetErr() is set to TRUE.

<xcSourceFile>   The name of a structure-extended file to use as the structure definition for the new database file, including an optional drive, directory, and extension.

If <xcSourceFile> does not exist, a runtime error is raised. If it exists, this command attempts to open the file in shared mode and, if successful, it proceeds. If access is denied because, for example, another process has exclusive use of the file, NetErr() is set to TRUE.

See SetDefault() and SetPath() for file searching and creation rules. The default extension for database files is determined by the RDD.

To qualify as a structure-extended file, the structure of the database file must be the same as the structure generated by COPY STRUCTURE EXTENDED, which you can refer to for more details.

Note: For data dictionary applications, you can have additional fields within the structure-extended file to describe the extended field attributes. You can, for example, have fields to describe such field attributes as a description, key flag, label, color, picture, and a validation expression for the VALID clause. CREATE FROM creates the new database file from the required fields only, ignoring all other fields in the extended structure. Moreover, it is not sensitive to the order of the required fields.

NEW	Selects the next unoccupied work area before opening <xcTargetFile>. If this clause is not specified, the current work area is used.
ALIAS <xcAlias>	An identifier name to associate with the work area when <xcTargetFile> is opened. If this clause is not specified, the alias defaults to the database file name. Duplicate alias names are not allowed within a single application.
VIA <cDriver>	The name of the RDD that will service the work area. If not specified, the default RDD as determined by RDDSetDefault() is used.

Notes

Character field lengths greater than 255: There are two methods for creating a character field with a length greater than 255 digits:
- Specify the field length using both the Field_Len and Field_Dec fields according to the following formulation:
- `_FIELD->Field_Len := <nLength> % 256`
- `_FIELD->Field_Dec := Integer(<nLength> / 256)`

- Modify the structure of the structure-extended file by changing the length of Field_Len to 5, then specify the actual field length.

Examples

This example is a procedure that simulates an interactive CREATE utility:

```
FUNCTION Start()
    CreateDatabase("newfile")

FUNCTION CreateDatabase(cNewDbf)
    // Create empty structure-extended file
    CREATE tmpext
    USE tmpext
    lMore := TRUE
    DO WHILE lMore
        // Input new field definitions
        APPEND BLANK
        CLEAR
        @ 5, 0 SAY "Name.....:  " GET Field_Name
        @ 6, 0 SAY "Type.....:  " GET Field_Type
        @ 7, 0 SAY "Length...:  " GET Field_Len
        @ 8, 0 SAY "Decimals.:  " GET Field_Dec
        READ
        lMore := (!EMPTY(Field_Name))
    ENDDO
```

```
// Remove all blank records
DELETE ALL FOR EMPTY(Field_Name)
PACK
CLOSE

// Create new database file
CREATE (cNewDbf) FROM tmpext
ERASE tmpext.dbf
```

The next example creates a new definition in a structure-extended file for a character field with a length of 4000 characters:

```
APPEND BLANK
REPLACE Field_Name WITH "Notes",;
    Field_Type WITH "C",;
    Field_Len WITH 4000 % 256,;
    Field_Dec WITH INTEGER(4000 / 256)
```

Assembly

XSharp.RT.DLL

See Also

COPY STRUCTURE, COPY STRUCTURE EXTENDED, CREATE, DbCopyXStruct(), DbCreate(), RDDSetDefault(), SetDefault(), SetExclusive(), SetPath()

1.8.4.5.17 DELETE Command

Note This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Tag records so they can be filtered with SetDeleted(TRUE), queried with Deleted(), or physically removed from the database file with PACK.

Syntax

```
DELETE [<Scope>] [WHILE <lCondition>] [FOR <lCondition>]  [[IN|ALIAS]
<workarea>]
```

Arguments

<Scope>	The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of:

[NEXT <NEXT>]   Optionally specifies the number of records to process starting

with the first record of the source file.

[RECORD <rec>]   An optional record ID If specified, the processing begins

with this data record in the source file.

[<rest:REST>]   The option REST specifies whether records are sequentially

searched only from the current up to the last record.

If a condition is specified, the option ALL is the default value.

[ALL]   The option ALL specifies that all records from the source file are imported.

This is the default setting.

WHILE <lCondition>	A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.
FOR <lCondition>	A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.
IN|ALIAS <workarea>	Specifies the work area for which the operation must be performed

X#

Description

This command merely marks the records for deletion. To permanently remove records that are marked for deletion, PACK the file Before packing, you can reinstate deleted records with RECALL.

Notes

Visibility:	If the current record is deleted and SetDeleted() is TRUE, it will be visible until the record pointer is moved.
Display of delete marker:	Display commands, such as LIST and DISPLAY, identify deleted records with an asterisk character (*).
Deleting all records:	To permanently remove all records from a database file regardless of their delete status, ZAP the file.
Shared mode:	For a shared database, this command requires all records that it operates on to be locked. You can accomplish this using one or more record locks or a file lock, depending on the scope of the command.

Examples

This example demonstrates use of the FOR clause to mark a set of records for deletion:

```
USE sales INDEX salesman NEW
DELETE ALL FOR Inactive
```

Assembly

XSharp.RT.DLL

See Also

DbDelete(), DBRLock(), Deleted(), FLock(), PACK, RECALL, RLock(), SetDeleted(), ZAP

1.8.4.5.18 GATHER Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Replaces the data in the current record of the currently selected table with data from an array, a set of variables, or an object.

Syntax

```
GATHER FROM ArrayName | MEMVAR | NAME ObjectName
    [FIELDS FieldList | FIELDS LIKE Skeleton | FIELDS EXCEPT
Skeleton]
    [MEMO]
```

Arguments

FIELDS <idFieldList>	The list of fields to process. The default is all fields with the exception of memo fields, unless the command supports the MEMO clause. Only fields with the same names and types in both files are appended. If fields with the same name do not match in data type, a runtime error is raised.
FIELDS LIKE <Skeleton>	You can specify field names with a wild card, such as FIELDS LIKE *name
FIELDS EXCEPT <Skeleton>	You can exclude fields, such as for example the primary keys: FIELDS EXCEPT Id <Skeleton> supports wildcards (* and ?). For example, to replace all fields that begin with the letters A and P, use: FIELDS LIKE A*,P*

Please note that you can combine FIELDS LIKE and FIELDS EXCEPT but you cannot combine a fields list with the LIKE and EXCEPT clauses.

FROM <ArrayName>	Specifies the array whose data replaces the data in the current record. The contents of the elements of the array, starting with the first element, replace the contents of the corresponding fields of the record. The contents of the first array element replace the first field of the record; the contents of the second array element replace the second field, and so on. If the array has fewer elements than the table has fields, the additional fields are ignored. If the array has more elements than the table has fields, the additional array elements are ignored.
MEMVAR	Specifies the variables or array from which data is copied to the current record. Data is transferred from the variable to

the field that has the same name as the variable. The contents of a field are not replaced if a variable doesn't exist with the same name as the field.

NAME <ObjectName>	Specifies an object whose properties have the same names as fields in the table. The contents of each field are replaced by the value of the property with the same names as the fields. The contents of a field are not replaced if a property doesn't exist with the same name as the field.
MEMO	Specifies that the contents of memo or Blob fields are replaced with the contents or array elements or variables. If you omit MEMO, memo and Blob fields are skipped when GATHER replaces the contents of fields with the contents of an array or variable. General and picture fields are always ignored in GATHER, even if you include the MEMO keyword.

See Also

APPEND FROM ARRAY, COPY TO ARRAY, SCATTER

1.8.4.5.19 GO Command

Purpose

Move the pointer to the specified record.

Syntax

```
GO[TO] <uRecID> | TOP | BOTTOM  [[IN|ALIAS] <workarea>]
```

Arguments

<uRecID>	The record to go to. The data type and interpretation of <uRecID> is determined by the RDD. For .DBF files, it is the record number. If <uRecID> does not exist, the work area is positioned to LastRec() + 1, and both EoF() and BoF() return TRUE.
TOP	Specifies the first logical record in the current work area.
BOTTOM	Specifies the last logical record in the current work area.
IN|ALIAS <workarea>	Specifies the work area for which the operation must be performed

Notes

Visibility: Even though a particular record may not be visible (because, for example, of DbSetFilter(), SetDeleted(TRUE), or a conditional controlling order), you can still go to that record.

Examples

This example saves the current record number, searches for a key, then restores the record pointer to the saved position:

```
FUNCTION KeyExists(uKeyExpr)
   LOCAL nSavRecord := RECNO()
   // Save the current record pointer position
   LOCAL lFound

   SEEK uKeyExpr
   IF (lFound := Found())
         .
         .  <Statements>
         .
   ENDIF

   GOTO nSavRecord    // Restore the record pointer
                                 // position

   RETURN lFound
```

Assembly

XSharp.RT.DLL

See Also

DbGoTo(), DbSetFilter(), DBSetRelation(), LastRec(), RecNo(), SetDeleted() , SET RELATION, SKIP

1.8.4.5.20 JOIN Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Create a new database file by merging selected records and fields from two work areas based on a specified condition.

Syntax

```
JOIN WITH <xcAlias> TO <xcTargetFile> FOR <lCondition> [FIELDS
<idFieldList>]
```

Arguments

WITH <xcAlias>   The alias identifier for the work area to merge with the current work area. If there is no open database associated with <xcAlias>, a runtime error is raised.

TO <xcTargetFile>   The name of the target database file, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension for database files is determined by the RDD .

                          If <xcTargetFile> does not exist, it is created. If it exists, this command attempts to open the file in exclusive mode and, if successful, the file is overwritten without warning or error. If access is denied because, for example, another process is using the file, NetErr() is set to TRUE.

FOR <lCondition>   A condition that is processed for each visible record in the current work area using every visible record in the WITH work area. If a record meets the condition, a new record is written to <xcTargetFile>. If a record does not meet the specified condition, it is ignored and the next record is processed.

                          Warning! The number of records processed will be the LastRec() of the primary work area multiplied by the LastRec() of the secondary work area. For example, if you have two database files with 100 records each, the number of records JOIN processes is the equivalent of sequentially processing a single database file of 10,000 records. Therefore, use this command carefully.

FIELDS <idFieldList>   The projection of fields from both work areas into the new database file. To specify fields in the secondary work area, reference them with the alias operator (->). If the FIELDS clause is not specified, all fields from the current work area are included in the target database file.

Notes

Deleted records: If SetDeleted() is FALSE, deleted records in both source files are processed, but their deleted status is not retained in *<xcTargetFile>*; thus, no record in the target file is marked for deletion, regardless of its deleted status in the source files.
Visibility: If SetDeleted() is TRUE, deleted records (in both files) are not visible and are, therefore, not processed. Similarly, records that are filtered (with DbSetFilter() or a conditional controlling order) are not processed.

Examples

This example joins the CUSTOMER.DBF with INVOICES.DBF to produce PURCHASES.DBF:

```
USE invoices NEW
USE customer NEW
JOIN WITH Invoices TO purchases;
    FOR Last = Invoices->Last;
    FIELDS First, Last, Invoices->Number, ;
          Invoices->Amount
```

Assembly

XSharp.RT.DLL

See Also

DbJoin(), SET RELATION, SetDefault(), SetPath()

1.8.4.5.21 LOCATE Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Search for the first record in the current work area that matches the specified condition and scope.

Syntax

```
LOCATE [<Scope>] FOR <lCondition> [WHILE <lCondition>]
```

X#

Arguments

<Scope>	The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of:

[NEXT <NEXT>]   Optionally specifies the number of records to process starting

with the first record of the source file.

[RECORD <rec>]   An optional record ID If specified, the processing begins

with this data record in the source file.

[<rest:REST>]   The option REST specifies whether records are sequentially

searched only from the current up to the last record.

If a condition is specified, the option ALL is the default value.

[ALL]   The option ALL specifies that all records from the source file are imported.

This is the default setting.

WHILE <lCondition>	A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.
FOR <lCondition>	A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.

Description

LOCATE evaluates each visible record within the scope using the for condition. As soon as a record meets the condition, the process terminates, leaving the record pointer on the matching record and setting the Found() flag to TRUE. If the for condition is FALSE for all records in the scope, the Found() flag is set to FALSE, and the position of the record pointer depends on the scope.

Each work area has its own locate condition which remains active until you execute another locate operation (for example, with LOCATE or DBLocate()), reset the locate condition (for example, with VODBSetLocate()), or terminate the application.

Notes

CONTINUE: Once you locate a record and have processed it, you can resume the search from the current record pointer position with CONTINUE (or DBContinue()). Both the *<Scope>* and the while condition, however, apply only to the initial locate operation and are not known by subsequent continue operations. To continue a pending locate with a scope or while condition, use SKIP then LOCATE REST WHILE *</Condition>* instead of CONTINUE as shown in the example below.

Examples

These examples show typical LOCATE constructs:

```
USE sales INDEX salesman
LOCATE FOR Branch = "200"
? Found(), EOF(), RECNO()  // Result:  TRUE FALSE 5
LOCATE FOR Branch = "5000"
? Found(), EOF(), RECNO()  // Result:  FALSE TRUE 85
```

The next example shows a LOCATE with a WHILE condition that is continued by using LOCATE REST:

```
SEEK "Bill"
LOCATE FOR Branch = "200" WHILE Salesman = "Bill"
DO WHILE Found()
   ? Branch, Salesman
   SKIP
   LOCATE REST FOR Branch = "200" WHILE ;
                   Salesman = "Bill"
ENDDO
```

Assembly

XSharp.RT.DLL

See Also

CONTINUE, DbContinue(), DbLocate(), EoF(), Found(), RecNo(), SEEK, DbSetFilter()

1.8.4.5.22 PACK Command

Purpose

Remove all records marked for deletion from the current database file, rebuild all active orders, and recover all physical space occupied by the deleted records. Note that this will not affect memo files. Unused space in memo files can only be recovered by using COPY.

Syntax

```
PACK  [[IN|ALIAS] <workarea>]
```

Arguments

IN|ALIAS <workarea>   Specifies the work area for which the operation must be performed

Description

PACK is functionally equivalent to DBPack().

Warning! *PACK does not create backup files. You may want to make a backup of the file (using COPY FILE, for example) before issuing this command; otherwise, you will not be able to recover deleted records.*

Examples

The following shows the result of a simple PACK:

```
USE sales NEW
? LastRec()          // Result:  84

DELETE RECORD 4
PACK
? LastRec()          // Result:  83
```

Assembly

XSharp.RT.DLL

See Also

DbPack(), DELETE, Deleted(), RECALL, REINDEX, SetDeleted() , SetExclusive(), ZAP

1.8.4.5.23 RECALL Command

Purpose

Restore records marked for deletion in the current work area.

Syntax

```
RECALL [<Scope>] [WHILE <lCondition>] [FOR <lCondition>]  [[IN|ALIAS]
<workarea>
```

Arguments

<Scope>	The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of:
	[NEXT <NEXT>] Optionally specifies the number of records to process starting
	with the first record of the source file.
	[RECORD <rec>] An optional record ID If specified, the processing begins
	with this data record in the source file.
	[<rest:REST>] The option REST specifies whether records are sequentially
	searched only from the current up to the last record.
	If a condition is specified, the option ALL is the default value.
	[ALL] The option ALL specifies that all records from the source file are imported.
	This is the default setting.
WHILE <lCondition>	A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default

scope to the rest of the visible records in the file.

FOR \<lCondition>　　　A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no \<Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.

IN|ALIAS \<workarea>　　Specifies the work area for which the operation must be performed

RECALL is the inverse of the DELETE command. If SetDeleted() is TRUE, RECALL can restore the current record or a specific record if you specify a RECORD scope.

Important! *Once you PACK a database file, all marked records are physically removed from the file and cannot be recalled.*

Shared mode: For a shared database, this command requires all records that it operates on to be locked. You can accomplish this using one or more record locks or a file lock, depending on the scope of the command.

Examples

This example shows the results of RECALL:

```
USE sales NEW
DELETE RECORD 4
? Deleted()          // Result:  TRUE
RECALL
? Deleted()          // Result:  FALSE
```

Assembly

XSharp.RT.DLL

See Also

DELETE, DbRecall(), Deleted(),, FLock(), PACK, RLock(), SetDeleted() , ZAP

1.8.4.5.24 REPLACE Command

Note　This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Assign new values to the contents of one or more fields in the current record.

Syntax

```
REPLACE <idField> WITH <uValue> [, <idField> WITH <uValue>...]
[<Scope>] [WHILE <lCondition>]
   [FOR <lCondition>]  [[IN|ALIAS] <workarea>]
```

Arguments

<idField>	The name of the field variable to assign a new value. If <idField> is prefaced with an alias or declared using FIELD <idField> IN <idAlias>, the assignment takes place in the designated work area. Otherwise, the current work area is assumed.
WITH <uValue>	Defines the value to assign to <idField>.
<Scope>	The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of: [NEXT <NEXT>] Optionally specifies the number of records to process starting with the first record of the source file. [RECORD <rec>] An optional record ID If specified, the processing begins with this data record in the source file. [<rest:REST>] The option REST specifies whether records are sequentially searched only from the current up to the last record. If a condition is specified, the option ALL is the default value. [ALL] The option ALL specifies that all records from the source file are imported. This is the default setting.
FOR <lCondition>	A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.

WHILE <lCondition> A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.

IN|ALIAS <workarea> Specifies the work area for which the operation must be performed

Description

REPLACE performs the same function as the assignment operator (:=) except that it assumes field variables.

Warning! *When you REPLACE a key field, the index is updated and the relative position of the record pointer within the index is changed. This means that replacing a key field with a scope or a condition can yield an erroneous result. To update a key field, suppress the controlling order (with, for example, SET ORDER TO 0 or DBSetOrder(0)) before the REPLACE. This insures that the record pointer moves sequentially in natural order and that all orders in the order list are updated properly.*

Shared mode: For a shared database, this command requires all records that it operates on to be locked. You can accomplish this using one or more record locks or a file lock, depending on the scope of the command.

Examples

This example shows a simple use of REPLACE:

```
USE customer NEW
APPEND BLANK
USE invoices NEW
APPEND BLANK

REPLACE Charges WITH Customer->Markup * Cost,;
   CustID WITH Customer->CustID,;
   Customer->TranDate WITH TODAY()
```

Using assignment statements in place of the REPLACE command looks like this:

```
Invoices->Charges := Customer->Markup * ;
   Invoices->Cost
Invoices->CustID := Customer->CustID
Customer->TranDate := TODAY()
```

Assembly

XSharp.RT.DLL

See Also

COMMIT, DbRLock(), DbSetOrder(), FLock(), RLock(), SET ORDER

1.8.4.5.25 SCATTER Command

Purpose

Assign new values to the contents of one or more fields in the current record.

Syntax

```
SCATTER [FIELDS <idFieldList> | FIELDS LIKE <Skeleton>
    | FIELDS EXCEPT <Skeleton>] [MEMO] [BLANK]
    | TO ArrayName | MEMVAR
    | NAME ObjectName [ADDITIVE]
```

Arguments

FIELDS <idFieldList>   The list of fields to process. The default is all fields with the exception of memo fields, unless the command supports the MEMO clause.
Only fields with the same names and types in both files are appended. If fields with the same name do not match in data type, a runtime error is raised.

FIELDS LIKE <Skeleton>   You can specify field names with a wild card, such as FIELDS LIKE *name

FIELDS EXCEPT <Skeleton>   You can exclude fields, such as for example the primary keys: FIELDS EXCEPT Id
<Skeleton> supports wildcards (* and ?). For example, to replace all fields that begin with the letters A and P, use:
FIELDS LIKE A*,P*

Please note that you can combine FIELDS LIKE and FIELDS EXCEPT but you cannot combine a fields list with the LIKE and EXCEPT clauses.

MEMO   Specifies that the field list include one or more memo fields.

BLANK	Include the BLANK keyword to create a set of empty variables or to fill the array or object with empty values. Each variable is assigned the same name, data type, and size as its field. If a field list is included, a variable is created for each field in the field list.
TO <ArrayName>	Specifies an array to which the record contents are copied. Starting with the first field, SCATTER copies the contents of each field into each element of the array in sequential order. SCATTER automatically creates a new array. The array elements have the same size and data types as the corresponding fields.
MEMVAR	Scatters the data to a set of variables instead of an array. SCATTER creates one variable for each field in the table and fills each variable with data from the corresponding field in the current record, assigning to the variable the same name, size, and type as its field.
NAME <ObjectName>	Creates an object whose properties have the same names as fields in the table. To copy the value of each field in the table to each object property, do not include the BLANK keyword. To leave the properties empty, include the BLANK keyword
ADDITIVE	To update the property values of an existing and valid Visual FoxPro object specified by ObjectName. Using BLANK with ADDITIVE omits the values for existing properties that have matching field names.

See Also

[GATHER](#), [COPY TO ARRAY](#), [APPEND FROM ARRAY](#)

1.8.4.5.26 SELECT Command

Purpose

Change the current work area.

Syntax

```
SELECT <xnWorkArea> | <xcAlias>
```

Arguments

<xnWorkArea>   A number from 0 to 250 that specifies the work area to select.

<xcAlias>   The alias identifier for the work area to select. If there is no open database file associated with the specified alias, a runtime error is raised.

Description

SELECT causes the specified work area to become the current work area. All subsequent database operations will apply to this work area unless another work area is explicitly specified for an operation.

SELECT is functionally equivalent to DBSelectArea().

Notes

Selecting 0:   Selecting work area 0 causes the lowest numbered unoccupied work area to become the current work area. Using SELECT 0 before opening a file is equivalent to USE with the NEW option.

Aliased expressions: The alias operator (->) can temporarily select a work area while an expression is evaluated and automatically restore the previously selected work area afterward.

Examples

This example opens a series of database files by selecting each work area by number, then using each database file in that work area:

```
SELECT 1
USE customer
SELECT 2
USE invoices
SELECT 3
USE parts
SELECT customer
```

To make your code independent of the work area number used, a better method is to open each database in the next available work area by specifying the NEW clause on the USE command line. In this example USE...NEW is employed instead of SELECT 0, then USE:

```
USE customer NEW
USE invoices NEW
SELECT customer
```

This code fragment changes work areas while saving the current work area name to a variable by using the Select() function. After executing an operation for the new work area, the original work area is restored:

```
nLastArea := Select()
USE newfile NEW

<Statements>...

SELECT (nLastArea)s
```

Assembly

XSharp.RT.DLL

See Also

Alias(), DbSelectArea(), Select(), USE, Used()

1.8.4.5.27 SET DELETED Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines whether to ignore or include records that are marked for deletion.

Syntax

```
SET DELETED ON | OFF | (<lToggle>)
```

Arguments

ON	Specifies that commands that operate on records, including records in related tables, using a scope ignore records that are marked for deletion
OFF	Specifies that commands that operate on records, including records in related tables, using a scope can access records marked for deletion. (Default)
lToggle	A logical expression which must appear in parentheses. True is equivalent to ON, False to OFF

Description

SET DELETED is functionally equivalent to SetDeleted().

Assembly

XSharp.RT.DLL

See Also

DELETE, DbSetFilter(), Deleted(), RECALL, SET FILTER, SetDeleted()

1.8.4.5.28 SET DRIVER Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the default RDD for the application.

Syntax

```
SET DRIVER TO <idDriverName>
```

Description

SET DRIVER is functionally equivalent to DBSetDriver().

Assembly

XSharp.RT.DLL

See Also

DbSetDriver()

1.8.4.5.29 SET FILTER Command

Purpose

Set or clear a filter condition for the current work area.

Syntax

```
SET FILTER TO [<lCondition>]
```

Arguments

TO <lCondition>   The condition used to filter records.

SET FILTER TO without an argument clears the filter condition.

Description

When a filter is set, records that do not meet the filter condition are not logically visible. The filter condition can be returned as a string using the DBFilter() function.
That is, database operations which act on logical records will not consider these records except to access them specifically by record number. So, for example, you can GOTO a filtered record or process one using the RECORD <*nRecord*> scope.

Note: Once a filter is set, it is not activated until the record pointer is moved from its current position. You can use GO TOP to activate it.

SET FILTER TO when specified with a condition is functionally equivalent to DbSetFilter() with the condition expressed as a code block and a string. SET FILTER TO with no arguments is equivalent to DBClearFilter().

Tip: If the RDD you are using supports optimization, use SET OPTIMIZE to control whether the RDD will optimize the filter search based on the available orders in the work area.

Notes

Visibility: As with SetDeleted() , a filter has no effect on INDEX and REINDEX. To create a conditional index, use the FOR condition of the INDEX command.

Examples

This example filters EMPLOYEE.DBF to only those records where the age is greater than 50:

```
USE employee INDEX name NEW
SET FILTER TO Age > 50
LIST Lastname, Firstname, Age, Phone
SET FILTER TO
```

Assembly

XSharp.RT.DLL

See Also

DbClearFilter(), DbFilter(), DbSetFilter(), SET OPTIMIZE, SetDeleted()

1.8.4.5.30 SET MEMOBLOCK Command

Note   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the block size for memo files.

Syntax

```
SET MEMOBLOCK TO <nSize>
```

Description

The initial memo file block size depends on the RDD. For most drivers that support the .DBT memo file format, it is 512 bytes. However, if you are using BLOB files (.DBV) via inheritance from the DBFBLOB driver, the default is 1.

SET MEMOBLOCK is functionally equivalent to calling
RDDInfo(_SET_MEMOBLOCKSIZE, *<nSize>*).

Assembly

XSharp.RT.DLL

See Also

RDDInfo()

1.8.4.5.31 SET OPTIMIZE Command

Purpose

Change the setting that determines whether to optimize using the open orders when
processing a filtered database file.

Syntax

```
SET OPTIMIZE ON | OFF | (<lToggle>)
```

Arguments

ON   Turns optimization on.

OFF   Turns optimization off.

<lToggle>   Turns optimization on if TRUE or off is FALSE.

                            Note: The initial default of this setting depends on the RDD.
                             Check RDDInfo(_SET_OPTIMIZE) to find out the setting for
                            the RDD in use for the current work area.

Description

For RDDs that support optimization, such as DBFCDX, SET OPTIMIZE determines
whether to optimize filters based on the orders open in the current work area. If this flag is
ON, the RDD will optimize the search for records that meet the filter condition to the fullest
extent possible, minimizing the need to read the actual data from the database file.
If this flag is OFF, the RDD will not optimize.

Assembly

XSharp.RT.DLL

See Also

DbSetFilter(), RDDInfo(), <u>SET FILTER</u>

1.8.4.5.32 SET RELATION Command

Purpose

Link a parent work area to one or more child work areas using a key expression, record number, or numeric expression.

Syntax

```
SET RELATION TO [<uRecID> INTO <xcAlias>] [, [TO] <uRecId> INTO
<xcAlias>...] [ADDITIVE] [SCOPED]
```

Arguments

TO <uRecID>	Performs an <xcAlias>->DBSeek(<uRecID>) operation if the child work area has a controlling order; otherwise, performs an <xcAlias>->DBGoTo(<uRecID>) operation. The operation serves to position the child work area to a matching index key value or record number each time the record pointer moves in the parent work area.
INTO <xcAlias>	The alias identifier for the child work area. If there is no open database file associated with the specified alias, a runtime error is raised.
ADDITIVE	Specifies that relations are to be added to the existing relations in the work area. If not specified, existing relations are cleared before the new ones are set.
SCOPED	Causes the SET RELATION command to map to the OrdSetRelation(). If not specified, SET RELATION maps to DBSetRelation().

SET RELATION TO with no arguments clears all relations defined in the current work area.

Description

Each parent work area can be linked to as many as eight child work areas.

Relating work areas synchronizes the child work area with the parent work area. This is achieved by automatically repositioning the child work area whenever the parent work area moves to a new record.

SET RELATION TO when specified with a list of expressions and alias names is functionally equivalent to using several DBSetRelation() (or OrdSetRelation(), if SCOPED is specified) function calls. If no ADDITIVE clause is specified, the command calls DBClearRelation() first.

Notes

Soft seeking: Seek operations that occur as part of relational positioning are never soft seeks (they do not respect the SetSoftSeek() flag). If a relational movement is unsuccessful, the child work area is positioned to LastRec() + 1, its Found() status returns FALSE, and its EoF() status returns TRUE.

Cyclical relations: Do not relate a parent work area to itself either directly or indirectly.

Record number relations: To relate two work areas based on matching record numbers, use RecNo() for the SET RELATION TO expression and make sure the child work area has no active indexes.

Examples

This example relates three work areas in a multiple parent-child configuration with CUSTOMER related to both INVOICES and ZIP:

```
USE invoices INDEX invoices NEW
USE zip INDEX zipcode NEW
USE customer NEW
SET RELATION TO CustNum INTO Invoices, Zipcode INTO Zip
LIST Customer, Zip->City, Invoices->Number, Invoices->Amount
```

Sometime later, you can add a new child relation using the ADDITIVE clause, like this:

```
USE backorder INDEX backorder NEW
SELECT customer
SET RELATION TO CustNum INTO Backorder ADDITIVE
```

Assembly

XSharp.RT.DLL

See Also

DbGoTo(), DBRelation(), DBRSelect(),, DbSeek(), DbSetIndex(), DbSetOrder() , DBSetRelation(), Found(), OrdSetRelation(), RecNo(), SET INDEX, SET ORDER, SetSoftSeek()

1.8.4.5.33 SKIP Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Move the record pointer relative to the current record.

Syntax

```
SKIP [<nRecords>]  [[IN|ALIAS] <workarea>]
```

Arguments

<nRecords>	The number of logical records to move, relative to the current record. A positive value means to skip forward, and a negative value means to skip backward. If <nRecords> is omitted, a value of 1 is assumed.
IN|ALIAS <workarea>	Specifies the work area for which the operation must be performed

Description

SKIP is functionally equivalent to DBSkip(). Specifying the alias is like using DBSkip() in an aliased expression (such as, *<xcAlias>*->DBSkip(*<nRecords>*).

Examples

This example uses SKIP with various arguments and shows the result:

```
USE customer NEW
SKIP
? RECNO()                    // Result:  2
SKIP 10
? RECNO()                    // Result:  12
SKIP -5
? RECNO()                    // Result:  7
```

This example moves the record pointer in a remote work area:

```
USE customer NEW
USE invoices NEW
SKIP ALIAS customer
```

This example prints a report using SKIP to move the record pointer sequentially through the CUSTOMER database file:

```
LOCAL nLine := 99
USE customer NEW
SET PRINTER ON
DO WHILE !EOF()
   IF nLine > 55
        EJECT
        nLine := 1
   ENDIF
   Customer, Address, City, State, Zip
   ++nLine
   SKIP
ENDDO
SET PRINTER OFF
CLOSE customer
```

Assembly

XSharp.RT.DLL

See Also

BoF(), COMMIT, DbSetFilter(), DbSkip(), EoF(), GO, LOCATE, RecNo(), SEEK, SetDeleted()

1.8.4.5.34 SORT Command

Purpose

Copy records from the current work area to a database file in sorted order.

Syntax

```
SORT TO <xcTargetFile> ON <idFieldList> [<Scope>] [WHILE
<lCondition>] [FOR <lCondition>]
```

Arguments

TO <xcTargetFile>
: The name of the target database file to write the sorted records, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension for database files is determined by the RDD.

  If <xcTargetFile> does not exist, it is created. If it exists, this command attempts to open the file in exclusive mode and, if successful, the file is overwritten without warning or error. If access is denied because, for example, another process is using the file, NetErr() is set to TRUE.

ON <idFieldList>
: The sort keys, specified as a comma-separated list of field names. You may optionally add, after each field name, /A (to sort in dictionary order), /C (to ignore capitalization), or /D (to sort in descending order). The default setting is /A.

<Scope>
: The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of:
  [NEXT <NEXT>]   Optionally specifies the number of records to process starting
                         with the first record of the source file.
  [RECORD <rec>]   An optional record ID If specified, the processing begins
                         with this data record in the source file.
  [<rest:REST>]   The option REST specifies whether records are sequentially
                         searched only from the current up to the last record.

If a condition is specified, the option ALL is the default value.

[ALL] The option ALL specifies that all records from the source file are imported. This is the default setting.

WHILE <lCondition>   A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.

FOR <lCondition>   A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.

Description

SORT is functionally equivalent to DBSort().

Examples

This example copies a mailing list using a descending sort key to a smaller list for printing:

```
USE mailing INDEX zip
SEEK "900"
SORT ON Lastname /D, Firstname /D TO invite ;
            WHILE Zip = "900"
USE invite NEW
REPORT FORM rsvplist TO PRINTER
```

Assembly

XSharp.RT.DLL

See Also

ASort(), DbSort(), FLock(), INDEX,SetDefault(), SetPath(), USE

1.8.4.5.35 SUM Command

Purpose

Total a series of numeric expressions for a range of records in the current work area, and assign the results to a series of variables.

Syntax

```
SUM <nValueList> TO <idVarList> [<Scope>] [WHILE <lCondition>]
[FOR <lCondition>]
```

Arguments

<nValueList>	The list of values to sum for each record processed. Note that the <nValueList> is required and not optional as it is in other dialects.
TO <idVarList>	Defines a list of one or more variables to assign the results of the sum. If any variable reference in the list is ambiguous (that is, not declared at compile time or not explicitly qualified with an alias), it is assumed to be MEMVAR. If any variable in the list is not visible or does not exist, a private variable is created using <uValue>.
<Scope>	The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of:

[NEXT <NEXT>]   Optionally specifies the number of records to process starting

with the first record of the source file.

[RECORD <rec>]   An optional record ID If specified, the processing begins

with this data record in the source file.

[<rest:REST>]   The option REST specifies whether records are sequentially

searched only from the current up to the last record.

If a condition is specified, the option ALL is the default value.

[ALL]   The option ALL specifies that all records from the source file are imported.

This is the default setting.

WHILE <lCondition>	A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.
FOR <lCondition>	A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.

Examples

This example illustrates the use of SUM:

```
LOCAL nTotalPrice, nTotalAmount
USE sales NEW
SUM Price * .10, Amount TO nTotalPrice, nTotalAmount

? nTotalPrice                    // Result:  151515.00
? nTotalAmount                   // Result:  150675.00
```

Assembly

XSharp.RT.DLL

See Also

AVERAGE, COUNT, DBEval(), TOTAL

1.8.4.5.36 TOTAL Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Summarize records in the current work area by key value to a database file.

Syntax

```
TOTAL TO <xcTargetFile> ON <uKeyValue> FIELDS <idFieldList>
[<Scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

TO <xcTargetFile>
The name of the target database file to write the totaled records, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension for database files is determined by the RDD.

If <xcTargetFile> does not exist, it is created. If it exists, this command attempts to open the file in exclusive mode and, if successful, the file is overwritten without warning or error. If access is denied because, for example, another process is using the file, NetErr() is set to TRUE.

ON <uKeyValue>
The key value used to summarize the records. To make the summarizing operation accurate, the current database file should be indexed or sorted on this expression.

FIELDS <idFieldList>
The list of numeric fields to total. If the FIELDS clause is not specified, no numeric fields are totaled. Instead each numeric field in the target file contains the value for the first record in the source file matching the key value.

<Scope>
The portion of the current database file to process. The default is all visible records. Scope is one or more clauses of:
[NEXT <NEXT>]   Optionally specifies the number of records to process starting
                                             with the first record of the source file.
[RECORD <rec>]   An optional record ID If specified, the processing begins
                                             with this data record in the source file.
[<rest:REST>]   The option REST specifies whether records are sequentially
                                             searched only from the current up to the last record.
                                             If a condition is specified, the option ALL is the default value.
[ALL]   The option ALL specifies that all records from the source file are imported.
                                             This is the default setting.

WHILE <lCondition>   A condition that each visible record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the visible records in the file.

FOR <lCondition>   A condition that each visible record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next visible record is processed. If no <Scope> or WHILE clause is specified, having a for condition changes the default scope to all visible records.

Description

TOTAL works by first copying the structure of the current database file to specified target file, except for memo fields. It then sequentially scans the current database file within the specified scope of records.

As each record with a unique key value is encountered, that record is copied to the target database file. The values of numeric fields specified in the FIELDS list from successive records with the same key value are added to fields with the same names in the target file. Summarization proceeds until a record with a new key value is encountered at which point the process is repeated.

Important! *To successfully total numeric fields, the numeric fields in the current database file structure must be large enough to hold the largest total possible for that numeric field. A runtime error will be raised if there is a numeric field overflow.*

TOTAL is functionally equivalent to DBTotal().

Notes

Deleted records: If SetDeleted() is FALSE, deleted records in the source file are processed. Records in the target file inherit the deleted status of the first matching record in the source file.

Visibility: If SetDeleted() is TRUE, however, deleted records are not visible and are, therefore, not processed. Similarly, filtered records (with DbSetFilter() or a conditional controlling order) are not processed.

Examples

In this example, a database file is totaled on the key expression of the controlling order using a macro expression. When the macro expression is encountered, the expression is evaluated and the resulting string is substituted for the TOTAL *<uKeyValue>* argument:

```
USE sales INDEX branch NEW
TOTAL ON &(IndexKey(0)) FIELDS Amount TO Summary
```

Assembly

XSharp.RT.DLL

See Also

AVERAGE, DBTotal(), INDEX, SetDefault(), SetPath(), SORT, SUM, UPDATE

1.8.4.5.37   UPDATE Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Replace fields in the current work area with values from another work area, based on the specified key value.

Syntax

```
UPDATE FROM <xcAlias> ON <uKeyValue> [RANDOM]REPLACE <idField>
WITH <uValue>   [, <idField> WITH <uValue>...]
```

Arguments

FROM <xcAlias>   The alias identifier for the work area used to update records in the current work area. If there is no open database file associated with the specified alias, a runtime error is raised.

ON <uKeyValue>   The expression that defines matching records in the FROM work area.

REPLACE <idField>   A field in the current work area to update with a new value.

WITH <uValue>   The value used to update the current field. You must reference any field contained in the FROM work area with the correct alias.

RANDOM   If specified, the current work area must be ordered (using an index order) by <uKeyValue> but the FROM work area

records can be in any order. If not specified, both the current work area and the FROM work area must be ordered (logically or physically) by <uKeyValue>.

Description

UPDATE can only update records in the current work area with unique key values. When there is more than one instance of a key value, only the first record with the key value is updated. The FROM work area, however, can have duplicate key values.

UPDATE is functionally equivalent to DBUpdate().

Notes

Deleted records: If SetDeleted() is FALSE, deleted records in both files are processed. Records in the file being updated retain their deleted status and are not affected by the deleted status of records in the FROM file.

Visibility: If SetDeleted() is TRUE, however, deleted records are not visible and are, therefore, not processed. Similarly, filtered records (with DbSetFilter() or a conditional controlling order) are not processed.

Shared mode: For a shared database, UPDATE requires a file lock on the current database file. The FROM database file can be open in any mode.

Examples

This example updates the CUSTOMER database file with outstanding invoice amounts:

```
USE invoices NEW
USE customer INDEX customer NEW
UPDATE FROM Invoices ON Last;
    REPLACE Owed WITH Owed + Invoices->Amount RANDOM
```

Assembly

XSharp.RT.DLL

See Also

DBCreateIndex(), DBUpdate(), FLock(), INDEX, JOIN, REPLACE, SetUnique(), SORT, TOTAL

1.8.4.5.38 USE Command

| Note | This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available |

Purpose

Open an existing database file, its associated memo file, and optionally associated index files in the current or the next available work area.

Syntax

```
USE [<xcDataFile> [INDEX <xcIndexList>] [ALIAS <xcAlias>] [FIELDS
<aFields>] [NEW] [READONLY]
    [EXCLUSIVE | SHARED] [VIA <cDriver>] [INHERIT <acRDDs>]]
```

Arguments

<xcDataFile>   The name of the database file to open, including an optional drive, directory, and extension. If the database file has a corresponding memo file, it is also opened. The default extension for database and memo files is determined by the RDD.

INDEX <xcIndexFileList>

The names of the index files to open, including an optional drive, directory, and extension for each. The default extension is determined by the RDD and can be obtained using DBOrderInfo(DBOI_INDEXEXT).

If you specify <xcIndexList> as an expression and the value returned is spaces or NIL, it is ignored.

It is recommended that you open index files with SET INDEX or DBSetIndex() for proper resolution in case of a concurrency conflict.

If the database file, its corresponding memo file, or any of the index files does not exist, a runtime error is raised. See SetDefault() and SetPath() for file searching and creation rules.

ALIAS <xcAlias>   An identifier name to associate with the work area when <xcDataFile> is opened. If not specified, the alias defaults to the database file name. Duplicate alias names are not allowed within a single application.

FIELDS <aFields>	An array containing field descriptions in the format returned by DBStruct().
	This argument does not apply to DBF files. It is intended for use with file formats that do not store field descriptions. For example, if you use an RDD that supports SDF or delimited files, you can use this argument to define the file structure, which can then be used with other commands or functions to access the field descriptions. Here is an example of this argument:

```
aFields := {;
      {"First", "C", 35, 0};
            {"Last", "C", 35, 0};
      {"Birthday", "D", 8, 0}}
      USE Names FIELDS aFields VIA "DELIM"
? First              // Return:  Josie
```

NEW	Selects the next available work area before opening <xcDataFile>. If not specified, <xcDataFile> is opened in the current work area.
READONLY	Attempts to open <xcDataFile> with a read-only attribute, prohibiting updates to the work area. If not specified, <xcDataFile> is opened read-write, allowing updates. If <xcDataFile> cannot be accessed using the indicated attribute, a runtime error is raised.
EXCLUSIVE	Attempts to open <xcDataFile> for exclusive (non-shared) use. All other processes are denied access until the database file is closed.
SHARED	Attempts to open <xcDataFile> for shared use.
	If neither SHARED nor EXCLUSIVE is specified, the USE command attempts to open <xcDataFile> in the mode indicated by the SetExclusive() flag. However, it is highly recommended that you specify the open mode as part of the USE command rather than relying on SetExclusive() to determine it for you.
VIA <cDriver>	The name of the RDD that will service the work area. If not specified, the default RDD as determined by RDDSetDefault() is used.
INHERIT <acRDDs>	A one-dimensional array with the names of RDDs from which the main RDD inherits special functionality. This allows you to use RDDs with special capabilities, like encryption or decryption, in different work areas with different database drivers. These RDDs overlay special

functions of the main RDD (specified with the VIA clause). If multiple RDDs (specified with this INHERIT clause) implement the same function, the function associated with the last RDD in the list takes precedence.

USE specified with no arguments closes the database file open in the current work area.

Description

The USE command attempts to open *<xcDataFile>* (and its associated .DBF file, if any) in the indicated mode. If the file is successfully opened, the operation proceeds to open any indicated index files in the same mode — any files that were already open in the work area are closed. The first order in the first index file in the list becomes the controlling order.

If access is denied because, for example, another process has exclusive use of the database file, NetErr() is set to TRUE but no runtime error is raised. For this reason, it is recommended that you open index files as a separate operation (with SET INDEX or DBSetIndex()). Otherwise, a runtime error will result when the USE command tries to open the first index file in the list because the database file will not be open.
When a database file is first opened, the record pointer is positioned at the first logical record in the file (record one if there is no controlling order).

If the database file is opened in shared mode, other processes can have concurrent access to the file and responsibility for data integrity falls on the application program. File and record locking (using, FLock(), RLock(), or DBRLock()) are the basic means of denying other processes access to a particular file or record.

Refer to the CLOSE command for information on how to close files of all types.

USE is functionally equivalent to DBUseArea().

Examples

This example opens a shared database file with associated index files. If NetErr() returns FALSE, indicating the USE was successful, the indexes are opened:

```
USE accounts SHARED NEW
IF !NetErr()
    SET INDEX TO acctnames, acctzip
ELSE
    ? "File open failed"
    BREAK
ENDIF
```

This example opens a database file with several index files specified as extended expressions:

```
cDataFile = "MyDbf"
acIndex = {"MyIndex1", "MyIndex2", "MyIndex3"}
USE (cDataFile) INDEX (acIndex[1]), ;
       (acIndex[2]), (acIndex[3])
```

Assembly

XSharp.RT.DLL

See Also

CLOSE, DbSelect(), DbSetIndex(), DbSetOrder(), DbUseArea(), NetErr(), RddSetDefault() SELECT, SET INDEX, Used()

1.8.4.5.39 ZAP Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Permanently remove all records from database file in the current work area.

Syntax

```
ZAP  [[IN|ALIAS] <workarea>]
```

Arguments

IN|ALIAS <workarea>   Specifies the work area for which the operation must be performed

Description

ZAP is functionally equivalent to DBZap().

Assembly

XSharp.RT.DLL

See Also

DBZap(), <u>DELETE</u>, <u>PACK</u>, <u>USE</u>

1.8.4.6   **Date and Time**

<u>SET CENTURY</u>
<u>SET DATE</u>
<u>SET DATE FORMAT</u>
<u>SET EPOCH</u>

1.8.4.6.1   SET CENTURY Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (<u>-nostddefs</u>) files then this command will not be available. If you tell the compiler to use a different standard header file (<u>-stddef</u>) then this command may also be not available

Purpose

Change the setting that determines whether to include or omit century digits in the date format.

Syntax

```
SET CENTURY ON | OFF | (<lToggle>)
```

Arguments

ON, OFF, lToggle   A logical expression which must appear in parentheses. True is equivalent to ON, False to OFF

Description

SET CENTURY is functionally equivalent to SetCentury().

Assembly

XSharp.RT.DLL

See Also

CToD(), DToC(), DToS(), SetCentury(), SetDateCountry(), SetDateFormat(), SetEpoch(), Today()

1.8.4.6.2 SET DATE Command

Purpose

Change the setting that determines the XSharp date format by selecting from a list of constants with corresponding date formats.

Syntax

```
SET DATE [TO] <kCountrySetting>
```

Description

SET DATE is functionally equivalent to SetDateCountry().

Examples

This example illustrates various system-defined country settings:

```
SET DATE German
? Today()                       // Result:  15.10.19
SET DATE Ansi
? Today()                       // Result:  19.10.15
```

Assembly

XSharp.RT.DLL

See Also

CToD(), DToC(), DToS(), SetCentury(), SetDateCountry(), SetDateFormat(), SetEpoch(), Today()

1.8.4.6.3 SET DATE FORMAT Command

Purpose

Change the setting that determines the XSharp date format.

Syntax

```
SET DATE FORMAT [TO] <cDateFormat>
```

Description

SET DATE FORMAT is functionally equivalent to SetDateFormat().

Examples

In this example the FORMAT clause directly specifies the date format:

```
SET DATE FORMAT "yyyy:mm:dd"
SetCentury(TRUE)
? Today()                        // Result:  2019:10:15
```

Assembly

XSharp.RT.DLL

See Also

CToD(), DToC(), DToS(), SetCentury(), SetDateCountry(), SetDateFormat(), SetEpoch(), Today(),

1.8.4.6.4 SET EPOCH Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines how dates without century digits are interpreted.

Syntax

```
SET EPOCH TO <nYear>
```

Description

SET EPOCH is functionally equivalent to SetEpoch().

Assembly

XSharp.RT.DLL

See Also

CToD(), DToC(), DToS(), SetCentury(), SetDateCountry(), SetDateFormat(), SetEpoch(), Today()

1.8.4.7   Entity Declaration

_DLL
ACCESS
ASSIGN
CLASS (Generic)
CLASS (FoxPro)
CLASS (Xbase++)
CONSTRUCTOR
DECLARE METHOD
DEFINE
DELEGATE
DESTRUCTOR
ENUM
EVENT
FUNCTION
GLOBAL
INTERFACE
METHOD
OPERATOR
PROCEDURE
PROPERTY
STRUCTURE
UNION
VOSTRUCT

1.8.4.7.1 _DLL Statement

Purpose

Declare an entity defined in a DLL to the compiler.

Syntax

```
[Attributes] [Modifiers] _DLL FUNCTION | PROCEDURE
[([<idParam> [AS | REF <idType>] [, ...])]
[AS <idType>]
[<idConvention>]
:<idDLL>.<idEntity>
[CharSet= AUTO|ANSI|UNICODE]
```

Arguments

<idEntity>	The name or number of the entity as defined in the DLL. This is normally, but not necessarily, the same as the entity name defined in <EntityDeclaration>, which may define an alias by which the entity is called in your application. <idEntity> must be part of the public protocol of the DLL identified by <idDLL>. IdEntity may also be specified as a literal string. This should be done to specify names of exported functions in DLLs that contain characters that are not allowed as part of an identifier in X#.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]

<idConvention>	Specifies the calling convention for this entity.
	<idConvention> must be one of the following:

- o   CLIPPER
- o   STRICT
- o   PASCAL
- o   CALLBACK
- o   THISCALL

Most calling conventions are for backward compatibility only.

There are 2 exceptions:

CLIPPER declares that a method has untyped parameters. This is usually only needed for methods without any declared parameters. Otherwise the compiler will assume CLIPPER calling convention when it detects untyped parameters.

Methods and Functions in external DLL may have STRICT, PASCAL, CALLBACK

<idDLL>	The name of the DLL file that contains the entity definition, specified without an extension or path name (that is, its base name). A .DLL extension is assumed (with some exceptions determined by Windows that may have an .EXE extension), and the rules used to search for the file at runtime are explained in the Description section below.
Charset	Optionally specifies which character set string parameters have

Description

_DLL declares an entity that is used by your application but defined in a DLL. This statement tells the compiler not only the location and name (or number) of the DLL entity, but also its calling convention (that is, what parameters it expects and the type of value that it returns). When you declare an entity with the _DLL statement, it also indicates to the compiler that the entity has no additional source code following its declaration. Once declared, the entity may be called in your application in the standard way.

Warning: Entity names contained within the _DLL statement are case sensitive. This is in direct contradiction to the X# compiler which is not case sensitive but is consistent with Windows calling protocols. For this reason you must take extra care if Case Sensitization is turned on or you have the Caps Lock on.

Examples

The following examples illustrate _DLL declarations for two Windows API functions:

```
_DLL FUNCTION MessageBeep(siLevel AS SHORTINT) AS VOID
PASCAL:User.MessageBeep

_DLL FUNCTION MessageBox(hwndParent AS PTR, pszText AS PSZ,
```

```
pszCapt AS PSZ, dwFlags AS DWORD) ;
    AS LONG PASCAL:User.MessageBox
// You can also declare a function with "normal" string parameters
if you want. Add the ANSI or UNICODE clause to indicate which
version you want to call.
_DLL FUNCTION MessageBox(hwndParent AS PTR, pszText AS STRING,
pszCapt AS STRING, dwFlags AS DWORD) ;
    AS LONG PASCAL:User.MessageBox ANSI
```

See Also

[ACCESS](#), [ASSIGN](#), [FUNCTION](#), [METHOD](#), [PROCEDURE](#)

1.8.4.7.2 CLASS Members

Enter topic text here.

1.8.4.7.2.1 ACCESS Statement

Purpose

Declare a method to access a non-exported or virtual instance variable.

Syntax

[Attributes] [Modifiers] **ACCESS** *<idName>*
[([*<idParam>* [AS | REF *<idType>*] [, ...])]
[AS *<idType>*] [*<idConvention>*]
[CLASS *<idClass>*]
[=> <expression>]
CRLF
[*<Body>*]
[END ACCESS]

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, PROTECTED, HIDDEN, INTERNAL, SEALED, ABSTRACT or STATIC.

\<idName>	A valid identifier name for the instance variable whose access method you are defining. Like other methods, access methods are entities; however, the system uses a unique naming scheme for them to prevent collisions with other entity names. Access method names must be unique within a class, but can share the same name as other entities in your application.
TypeParameters	This is supported for methods with generic type arguments. This something like \<T> for a method with one type parameter named T. Usually one of the parameters in the parameter list is then also of type T.
\<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN \<idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS \<idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
AS \<idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
TypeParameterConstraints	Here you can specify constraints for the Type parameters, such as WHERE T IS SomeName or WHERE T IS New
\<idConvention>	Specifies the calling convention for this entity. \<idConvention> must be one of the following: o CLIPPER o STRICT o PASCAL o CALLBACK o THISCALL Most calling conventions are for backward compatibility only. There are 2 exceptions: CLIPPER declares that a method has untyped parameters. This is usually only needed for methods without any declared parameters. Otherwise the compiler will assume

	CLIPPER calling convention when it detects untyped parameters.
	Methods and Functions in external DLL may have STRICT, PASCAL, CALLBACK
CLASS <idClass>	The class to which this method belongs. This clause is mandatory for entities declared outside of a CLASS .. END CLASS construct
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body
END ACCESS	Optional end clause to indicate the end of the ACCESS entity

Description

ACCESS declares a special method, called an access method, that is automatically executed each time you access the named instance variable.

You can define four types of instance variables in a CLASS declaration. All of these, except EXPORT, are called non-exported instance variables because they are not directly accessible externally (i.e., outside of the class).

For example, if you want to access a non-exported instance variable of an object from a function, you must use a method. Indeed, this is the purpose of not exporting the variable: encapsulation by being able to control all references to it through a method. However, the syntax for referencing a method is obviously different from that of referencing a variable. This violates encapsulation and is just plain cumbersome, since users of the class must be aware of how a property of the class is implemented in order to know whether to use a functional style or a variable style of reference.

For example, note the difference in accessing the instance variables *x* and *y* in the function UseClass() when the class uses a regular method for exporting the variable:

```
CLASS Test
    EXPORT x := 100
    INSTANCE y := 10000

METHOD GetValueY() CLASS Test
    RETURN y
```

```
FUNCTION UseClass()
   LOCAL oTest AS Test
   oTest := Test{}
   ? oTest:x
   ? oTest:GetValueY()          // Access y using method
```

If you replace the regular method with an access method, the syntax for accessing both variables is the same even though one of them is insulated by a method:

```
ACCESS y CLASS Test
   RETURN y

FUNCTION UseClass()
   LOCAL oTest AS Test
   oTest := Test{}
   ? oTest:x
   ? oTest:y                    // Using ACCESS method
```

Non-exported variables come in three categories, each with its own properties (see the CLASS statement entry in this guide for details):

- INSTANCE
- PROTECT
- HIDDEN

INSTANCE variables are specifically designed to work with access and assign methods which is the main reason for their late binding. By defining an access method with the same name as an INSTANCE variable, you effectively override the variable by causing all non-assignment references, both external and internal, to invoke the access method.

The exception is that within an access (or assign) method, instance variables of the same name refer to the variable — otherwise, you would never get anywhere.
For example:

```
CLASS Person
   INSTANCE Name, SSN

ACCESS Name CLASS Person
   RETURN Name          // Refers to variable Name

METHOD ShowName() CLASS Person
   ? Name               // Refers to ACCESS method
```

You can also use PROTECT and HIDDEN variables in conjunction with access methods. By defining an access method with the same name as a PROTECT or HIDDEN variable,

you can access the variable externally using the same syntax as you would inside the class. Internal references, however, always refer directly to the variable because of early binding.

Of course, you do not have to give the access method and the instance variable the same name. This is only for your convenience. It is the return value of the method that is used when you access *<idVar>*. Thus, for PROTECT/HIDDEN variables, you can provide an access method with a different name. For example:

```
CLASS Person
    PROTECT Name_Protected

ACCESS Name CLASS Person
    RETURN Name_Protected
```

A virtual variable is one that is not defined as part of the class but composed from other instance variables. In other words, it is a variable that is calculated based on the values of other instance variables. As with non-exported instance variables, you could use a regular method to compute virtual variables, but this means using a different syntax for accessing them. Access methods extend the syntax used for accessing instance variables to virtual variables.
For example:

```
CLASS Person
    INSTANCE Name, SSN

ACCESS Name CLASS Person
    RETURN Name

METHOD Init(cName, cSSN) CLASS Person
    Name := cName
    SSN := cSSN

ACCESS FullID CLASS Person
    RETURN Name + SSN

FUNCTION UseClass()
    LOCAL oFriend AS Person
    oFriend := Person{"Bill Brown", "213-88-9546"}
    ? oFriend:Name        // Bill Brown
    ? oFriend:FullID      // Bill Brown213-88-9546
```

EXPORT variables are a lot faster and easier to use than non-exported variables and access methods, but using them defies the encapsulation that you should strive for to further the integrity of your application. Using access and assign methods, you can use exported variables early in the prototyping stage of an application, and later protect the variables with methods without changing the class interface.

ACCESS is a special case of METHOD and, except for the way you invoke it (i.e., without arguments, like an instance variable), its behavior is the same as any other method. See the METHOD statement in this guide for more details.

Note: Internal references to access methods that do not have a corresponding regular INSTANCE variable (e.g., virtual variables or public access to HIDDEN or PROTECT variables with different names) must use the SELF: prefix. Internal references means references from inside methods of the class or one of its subclasses. If the system does not find an instance variable, it assumes a memory variable (which can produce a compiler error depending on whether Allow Undeclared Variables has been chosen in the compiler settings), and it does not attempt to identify the reference as an access method, unless SELF: is used.

Strongly typed Methods

In addition to XSharp untyped method implementation, **strong typing** of method parameters and return values is now supported, providing you with a mechanism through which highly stable code can be obtained. The type information supplied enables the compiler to perform the necessary type checking and, thus, guarantee a much higher stable code quality.

A further benefit obtained by utilizing strongly typed methods is that of performance. The implementation of typed methods presumes that when the programmer employs strongly typed messages, the compiler can effectively perform an **early binding** for the respective methods invocation. As a result of this implementation, typed methods invocations are somewhat faster than the respective untyped counterparts. These advantages are, however, attained at the price of losing the flexibility which untyped methods offer.

It is, therefore, important to remember that interchangeably using both the typed and the untyped versions of a particular methods in an inheritance chain is neither permissible nor possible.

XSharp allows strong typing of METHODs, ACCESSes and ASSIGNs. The programmer accomplishes the specification of the strongly typed methods with XSharp in two steps:

1.   A mandatory declaration of the typed method is given in its respective class.
This declaration is reponsible for declaring the order of the methods in the so-called virtual table which XSharp employs for the invocation of typed methods. A re-declaration of a method in a subclass is NOT permissible, since it would cause abiguity for the compiler.
2.   Define the strongly typed method.
Unlike strongly typed functions, method typing requires strongly typing of the method arguments, the method return value AND speficying a valid calling convention.
The following calling conventions are valid for typed methods: STRICT, PASCAL or CALLBACK.

Examples

The following example uses ACCESS to perform a calculation based on the value of other instance variables:

```
CLASS Rectangle
    INSTANCE Length, Height AS INT

METHOD Init(nX, nY) CLASS Rectangle
    Length := nX
    Height := nY
    RETURN SELF

ACCESS Area CLASS Rectangle
    RETURN Length * Height

FUNCTION FindArea()
    LOCAL oShape AS Rectangle
    oShape := Rectangle{3, 4}
    ? oShape:Area        // Displays:  12
```

See Also

ASSIGN, CLASS, METHOD, PROPERTY

1.8.4.7.2.2 ASSIGN Statement

Purpose

Declare a method to assign a value to a particular instance variable.

Syntax

[Attributes] [Modifiers] **ASSIGN** *<idVar>*
[([*<idParam>* [AS | REF *<idType>*] [, ...])]
[AS *<idType>*] [*<idConvention>*]
[CLASS *<idClass>*]
[=> <expression>]
CRLF
[*<Body>*]
[END ACCESS]

Arguments

Attributes   An optional list of one or more attributes that describe meta

	information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, PROTECTED, HIDDEN, INTERNAL, SEALED, ABSTRACT or STATIC.
<idVar>	A valid identifier name for the instance variable whose assign method you are defining. Like other methods, assign methods are entities; however, the system uses a unique naming scheme for them to prevent collisions with other entity names. Assign method names must be unique within a class, but can share the same name as other entities in your application.
TypeParameters	This is supported for methods with generic type arguments. This something like <T> for a method with one type parameter named T. Usually one of the parameters in the parameter list is then also of type T.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
TypeParameterConstraints	Here you can specify constraints for the Type parameters, such as WHERE T IS SomeName or WHERE T IS New
<idConvention>	Specifies the calling convention for this entity. <idConvention> must be one of the following:

o CLIPPER
o STRICT
o PASCAL
o CALLBACK
o THISCALL

Most calling conventions are for backward compatibility only.
There are 2 exceptions:
CLIPPER declares that a method has untyped parameters. This is usually only needed for methods without any declared parameters. Otherwise the compiler will assume CLIPPER calling convention when it detects untyped parameters.
Methods and Functions in external DLL may have STRICT, PASCAL, CALLBACK

CLASS <idClass>	The class to which this method belongs. This clause is mandatory for entities declared outside of a CLASS .. END CLASS construct
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body
END ASSIGN	Optional end clause to indicate the end of the ASSIGN entity

Description

ASSIGN declares a special method, called an assign method, that is automatically executed each time you assign a value to the named instance variable (e.g., *<idVar>* := *<uValue>*). The value on the right-hand side of the assignment operator is passed to the ASSIGN method as an argument.

You can define four types of instance variables in a CLASS declaration. All of these, except EXPORT, are called non-exported instance variables because they are not directly accessible externally (i.e., outside of the class).
For example, if you want to assign a value to a non-exported variable in a function, you must use a method. This means that the interface for assigning to instance variables is dependent on the implementation of the class. Assign methods give you a common syntax for assigning values to all instance variables. Used in conjunction with access methods (see the ACCESS statement in this guide), assign methods let you enforce encapsulation principles in your application while maintaining a stable class interface.

X#

The following example illustrates using a traditional method for assigning a value to a non-exported instance variable. Note the difference in assigning values to the instance variables *x* and *y* in UseClass():

```
CLASS Test
    EXPORT x
    INSTANCE y

METHOD PutValueY(nPut) CLASS Test
    y := nPut
    RETURN y

FUNCTION UseClass()
    LOCAL oTest AS Test
    oTest := Test{}
    ? oTest:x := 100
    ? oTest:PutValueY(100)  // Using regular method
```

If you replace the method with an assign method, as follows, the syntax for assignment to both variables is the same:

```
ASSIGN y(nPut) CLASS Test
    y := nPut
    RETURN y



FUNCTION UseClass()
    LOCAL oTest AS Test
    oTest := Test{}
    ? oTest:x := 100
    ? oTest:y := 100        // Using ASSIGN method
```

Note: ASSIGN methods should generally return the actual assigned value to ensure that the hypothetical variable it represents actually behaves like a variable — technically, that assignment is associative. This will allow chained assignments the same way as for regular variables (e.g., *Var2 := oTest1:y := oTest1:x := Var1 := 10000*).

Non-exported variables come in three categories, each with its own properties (see the CLASS statement entry in this guide for details):
- INSTANCE
- PROTECT
- HIDDEN

INSTANCE variables are specifically designed to work with assign and access methods which is the main reason for their late binding. By defining an assign method with the same name as an INSTANCE variable, you effectively override the variable by causing all assignment references, both external and internal, to invoke the assign method. The exception is that within an assign (or access) method, instance variables of the same name refer to the variable — otherwise, you would never get anywhere.

You can also use PROTECT and HIDDEN variables in conjunction with assign methods. By defining an assign method with the same name as a PROTECT or HIDDEN variable, you can make assignments to the variable externally using the same syntax as you would inside the class. Internal references, however, always refer directly to the variable because of early binding.

Of course, you do not have to give the assign method and the instance variable the same name. This is only for your convenience. It is the method itself that determines which instance variable name to use when you access *<idVar>* via an assignment. Thus, for PROTECT/HIDDEN variables, you can provide an assign method with a different name. For example:

```
CLASS Person
    PROTECT Name_Protected

ASSIGN Name(cPutName) CLASS Person
    Name_Protected := cPutName
    RETURN Name_Protected
```

EXPORT variables are a lot faster and easier to use than non-exported variables and assign methods, but using them defies the encapsulation that you should strive for to further the integrity of your application. Using assign and access methods, you can use exported variables early in the prototyping stage of an application, and later protect the variables with methods without changing the class interface.

ASSIGN is a special case of METHOD and, except for the way you invoke it (i.e., without arguments, like an instance variable), its behavior is the same as any other method. See the METHOD statement in this guide for more details.

Note: Internal references to access methods that do not have a corresponding regular INSTANCE variable (e.g., virtual variables or public access to HIDDEN or PROTECT variables with different names) must use the SELF: prefix. Internal references means references from inside methods of the class or one of its subclasses. If the system does not find an instance variable, it assumes a memory variable (which can produce a compiler error depending on whether Allow Undeclared Variables has been chosen in the compiler settings), and it does not attempt to identify the reference as an access method, unless SELF: is used.

Strongly typed Methods

In addition to XSharp untyped method implementation, **strong typing** of method parameters and return values is now supported, providing you with a mechanism through

which highly stable code can be obtained. The type information supplied enables the compiler to perform the necessary type checking and, thus, guarantee a much higher stable code quality.

A further benefit obtained by utilizing strongly typed methods is that of performance. The implementation of typed methods presumes that when the programmer employs strongly typed messages, the compiler can effectively perform an **early binding** for the respective methods invocation. As a result of this implementation, typed methods invocations are somewhat faster than the respective untyped counterparts. These advantages are, however, attained at the price of losing the flexibility which untyped methods offer.

It is, therefore, important to remember that interchangeably using both the typed and the untyped versions of a particular methods in an inheritance chain is neither permissible nor possible.

XSharp allows strong typing of METHODs, ACCESSes and ASSIGNs. The programmer accomplishes the specification of the strongly typed methods with XSharp in two steps:

1. A mandatory declaration of the typed method is given in its respective class.
    This declaration is reponsible for declaring the order of the methods in the so-called virtual table which XSharp employs for the invocation of typed methods. A re-declaration of a method in a subclass is NOT permissible, since it would cause abiguity for the compiler.
2. Define the strongly typed method.
    Unlike strongly typed functions, method typing requires strongly typing of the method arguments, the method return value AND speficying a valid calling convention.
    The following calling conventions are valid for typed methods: STRICT, PASCAL or CALLBACK.

Examples

The following example uses ASSIGN to establish a public protocol for making assignments to INSTANCE variables:

```
CLASS Rectangle
    INSTANCE Length, Height AS INT

ASSIGN Length(nX) CLASS Rectangle
    Length := nX
    RETURN Length

ASSIGN Height(nY) CLASS Rectangle
    Height := nY
    RETURN Height

FUNCTION UseClass()
```

```
LOCAL oShape AS Rectangle
oShape := Rectangle{}
oShape:Length := 3
oShape:Height := 4a
```

See Also

ACCESS, CLASS, METHOD, PROPERTY

1.8.4.7.2.3 CONSTRUCTOR Statement

Purpose

Declare a method to access a non-exported or virtual instance variable.

Syntax

[Attributes] [Modifiers] **CONSTRUCTOR**[([<*idParam*> [AS | REF|OUT|IN <*idType*>] [, ...])]
[=> <expression>]
CRLF
[<*Body*>]
[END CONSTRUCTOR]

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional

X#

	parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body
END CONSTRUCTOR	Optional end clause to indicate the end of the CONSTRUCTOR entity

See Also

ASSIGN, CLASS, METHOD

1.8.4.7.2.4 DECLARE METHOD Statement

Purpose

Forward declare a method, access or Assign

Description

DECLARE METHOD, DECLARE ASSIGN and DECLARE ACCESS are recognized by the X# compiler but no longer needed, so they are ignored.

1.8.4.7.2.5 DESTRUCTOR Statement

Purpose

Declare a method to access a non-exported or virtual instance variable.

Syntax

[Attributes] [Modifiers] **DESTRUCTOR [()]**
[=> <expression>]
CRLF
[<*Body*>]
[END DESTRUCTOR]

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body
END DESTRUCTOR	Optional end clause to indicate the end of the DESTRUCTOR entity

See Also

ASSIGN, CLASS, METHOD

1.8.4.7.2.6 EVENT Statement

Purpose

Declare a method to access a non-exported or virtual instance variable.

Syntax

[Attributes] [Modifiers] **EVENT**<*idName*>
[AS <*idType*>] [<*idConvention*>]**CRLF**
[
  [REMOVE <Expression>] [ADD <Expression>]
 | [ADD <Body > END ADD]
 | [ADD => <expression>]
 | [REMOVE <Body > END REMOVE]
 | [REMOVE => <expression>]
  END EVENT
 ]

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
<idName>	A valid identifier name for the event that you are defining. Like other methods, events are entities. Event names must be unique within a class, but can share the same name as other entities in your application.
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
<Expression>	Expression that implements the accessor
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body

Description

There are 3 types of event declarations:
- Single line declaration without ADD / REMOVE accessors
- Single line declaration with ADD / REMOVE accessors
- Multi line declaration with ADD accessor block and/or REMOVE accessor bock

See Also

ASSIGN, CLASS, METHOD

1.8.4.7.2.7 METHOD Statement

Purpose

Declare a method name and an optional list of local variable names.

Syntax

[Attributes] [Modifiers] **METHOD** *<idMethod>*
[Typeparameters]
[([*<idParam>* [AS | REF|OUT|IN *<idType>*] [, ...])]
[AS *<idType>*]
[TypeparameterConstraints]
[*<idConvention>*]
[CLASS *<idClass>*]
[=> <expression>]
CRLF
[*<Body>*]
[END METHOD]

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, PROTECTED, HIDDEN, INTERNAL, SEALED, ABSTRACT or STATIC.

<idMethod>	A valid identifier name for the method. Method names must be unique within a class, but can share the same name as other entities (including access and assign methods) in your application.
TypeParameters	This is supported for methods with generic type arguments. This something like <T> for a method with one type parameter named T. Usually one of the parameters in the parameter list is then also of type T.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
TypeParameterConstraints	Here you can specify constraints for the Type parameters, such as WHERE T IS SomeName or WHERE T IS New
<idConvention>	Specifies the calling convention for this entity. <idConvention> must be one of the following: - CLIPPER - STRICT - PASCAL - CALLBACK - THISCALL Most calling conventions are for backward compatibility only. There are 2 exceptions: CLIPPER declares that a method has untyped parameters. This is usually only needed for methods without any declared parameters. Otherwise the compiler will assume CLIPPER calling convention when it detects untyped parameters.

	Methods and Functions in external DLL may have STRICT, PASCAL, CALLBACK
CLASS <idClass>	The class to which this method belongs. This clause is mandatory for entities declared outside of a CLASS .. END CLASS construct
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body
END METHOD	Optional end clause to indicate the end of the inline METHOD entity

Description

A method is a subprogram comprised of a set of declarations and statements to be executed whenever you refer to the method using the message send operator, as in:

```
<idObject>:<idMethod>([<uArgList>])
```

Or:

```
SELF:<idMethod>([<uArgList>])
```

Classes, instance variables (see the CLASS statement in this guide), and methods are the basic object-oriented programming units. You will use methods in your applications to organize computational blocks of code for a specific class of objects.

Notes

The Start() method: All applications must either have one function or procedure named Start().
Start() serves as the startup routine when the application is executed. Start() must be declared without parameters or with an string array parameter and must either have an Int return value or of type Void

VO Compatibility:

VO has 2 special method names for constructing and destructing class objects: Init() and Axit().

In X# These methods should be named CONSTRUCTOR and DESTRUCTOR.

If you compile with compiler option /vo1 then you can still use the 'old' names. The compiler will then automatically map the Init() method to constructor and the Axit() method to destructor. This is NOT recommended.

The Init() method: If you define a method named Init(), it is called automatically when you create an instance of the class to which the method belongs. Arguments listed within the instantiation operators ({}) are passed as parameters to the Init() method. Common uses for the Init() method are to initialize instance variables, allocate memory needed by the object, register the object, create subsidiary objects, and set up relationships between objects.

The Axit() method: You do not need to deallocate memory used by objects because the garbage collector takes care of this for you automatically. However, in some cases an object can manage other resources that do need proper disposition. For example, if an object opens a database only for its own use, it should close it when it is finished and make the work area available for other uses.

If you register an object with the RegisterAxit() function, for example in the Init() method, and provide a method named Axit(), this Axit() method will automatically be called by the garbage collector just before the object is destroyed. Thus, in the Axit() method you can close databases, deallocate memory, or close communications links.

NoIVarGet()/NoIVarPut() methods: If you define methods named NoIVarGet() and NoIVarPut(), they will automatically be invoked if an instance variable that does not exist is referenced. They are called with the instance variable name as a parameter, in the form of a symbol and, in the case of NoIVarPut(), with the assigned value. This feature is useful in detecting and preventing a runtime error and for creating virtual variables dynamically at runtime. For example, the DBServer class uses this technique to make the database fields appear to be exported instance variables of a database object:

```
METHOD NoIVarGet(symFieldName) CLASS DBServer
   BEGIN SEQUENCE
      RETURN FieldGetAlias(symAlias, symFieldName)
   RECOVER
      // Pass it up if it was not a field name
      SUPER:NoIVarGet(symFieldName)
   END SEQUENCE

METHOD NoIVarPut(symFieldName, uValue) ;
   CLASS DBServer
   BEGIN SEQUENCE
      RETURN FieldGetAlias(symAlias, ;
         symFieldName, uValue)
   RECOVER
      // Pass it up if it was not a field name
      SUPER:NoIVarPut(symFieldName, uValue)
```

```
        END SEQUENCE

    FUNCTION DatabaseTest()
        LOCAL oDBServer AS DBServer
        oDBServer := DBServer{"customer"}
        ? oDBCustomer:CustName
        oDBCustomer:ZipCode := "12345"
```

The NoMethod() method: If you define a method named NoMethod(), it will automatically be invoked if a method name that you invoke within the same class cannot be found. The arguments passed will be the same as the original method. This feature is useful in detecting and preventing a runtime error when a method cannot be found. Use the NoMethod() function to find out the name of the method that could not be found.

Exporting locals through code blocks: When you create a code block, you can access local variables defined in the creating entity within the code block definition without having to pass them as parameters (i.e., local variables are visible to the code block). Using this fact along with the fact that you can pass a code block as a parameter, you can export local variables. For example:

```
    METHOD One() CLASS MyClass EXPORT LOCAL
        LOCAL nVar := 10 AS INT, cbAdd AS CODEBLOCK
        cbAdd := {|nValue| nValue + nVar}

        ? SELF:Two(cbAdd)        // Result:  210

    METHOD Two(cbAddEmUp) CLASS MyClass
        RETURN EVAL(cbAddEmUp,200)
```

When the code block is evaluated in Two(), *nVar*, which is local to method One(), becomes visible even though it is not passed directly as a parameter.

Invoking methods: The syntax to invoke a method of an object is as follows:

```
    <idObject>:<idMethod>([<uArgList>])
```

where *<uArgList>* is an optional comma-separated list of arguments to pass to the named method. *<idObject>* identifies the object to whom the method invocation is to be sent; to refer to the same object within a method you use SELF: or SUPER: (see note below) instead of *<idObject>*:.

You can invoke a method within an expression or as a program statement. If called as a program statement, the return value is ignored.

You can also use a method invocation as an aliased expression by prefacing it with an alias and enclosing it in parentheses:

```
<idAlias>->(<idObject>:<idMethod>([<uArgList>]))
```

When you do this, the work area associated with *<idAlias>* is selected, the method is executed, and the original work area is reselected. You can specify an aliased expression as a program statement, as you would any other expression.
A method can call itself recursively. This means you can refer to a method in its own *<MethodBody>*.

The visibility of typed methods can also be influenced by using the HIDDEN and PROTECT modifiers as in their use with instance variables. SUPER calls of HIDDEN methods in parent classes cannot be done by methods of the sub-classes.

Calling convention: Methods use the CLIPPER calling convention, unless they are strongly typed. See the FUNCTION statement in this guide for more information.

SELF and SUPER: SELF is a special variable that contains a reference to the object that is the receiver of a message (a message is sent to an object each time you use the send operator to invoke a method or access an instance variable). Whenever a message is sent to an object, a reference to the object is placed in SELF before the corresponding method is invoked. Within methods of the class, you must use SELF with the message send operator (:) to send messages to the current object. Using SELF: to access instance variables is optional; see the ACCESS and ASSIGN entries in this guide for details on when it is required (it is always allowed).

SUPER is another special variable that contains a reference to the class that is the nearest ancestor of the method lookup. It passes a message up the inheritance tree to the appropriate superclass and is meaningful only if the current object's class inherits from another class. You can use SUPER with the message send operator (:) to refer directly to a method defined in a superclass. If you redefine a method in a subclass (by creating a method with the same name as one in a superclass), SUPER is the only way you can override the redefined method with the superclass version.
SUPER: is useful when defining a subclass which adds some unique behavior, but nonetheless inherits standard behavior from its superclass. For example:

```
CLASS Person
   PROTECT cName AS STRING, symName AS SYMBOL

METHOD Init(cFirstName, cLastName) CLASS Person
   cName := cFirstName + " " + cLastName
   symName := String2Symbol(cName)

CLASS Customer INHERIT Person
   PROTECT wCustNo AS DWORD

METHOD Init(cFirstName, cLastName, nCustNo) ;
         CLASS Customer
```

```
SUPER:Init(cFirstName, cLastName)
wCustNo := nCustNo
```

The SELF and SUPER variables are allowed only in method definitions. SELF is the default return value for all methods.

Parameters: As an alternative to specifying method parameters in the METHOD declaration statement, you can use a PARAMETERS statement to specify them. This practice, however, is not recommended because it is less efficient and provides no compile-time integrity validation. See the PARAMETERS statement in this guide for more information.

Strongly typed Methods

In addition to XSharp untyped method implementation, **strong typing** of method parameters and return values is now supported, providing you with a mechanism through which highly stable code can be obtained. The type information supplied enables the compiler to perform the necessary type checking and, thus, guarantee a much higher stable code quality.

A further benefit obtained by utilizing strongly typed methods is that of performance. The implementation of typed methods presumes that when the programmer employs strongly typed messages, the compiler can effectively perform an **early binding** for the respective methods invocation. As a result of this implementation, typed methods invocations are somewhat faster than the respective untyped counterparts. These advantages are, however, attained at the price of losing the flexibility which untyped methods offer.

It is, therefore, important to remember that interchangeably using both the typed and the untyped versions of a particular methods in an inheritance chain is neither permissible nor possible.

XSharp allows strong typing of METHODs, ACCESSes and ASSIGNs. The programmer accomplishes the specification of the strongly typed methods with XSharp in two steps:

1.   A mandatory declaration of the typed method is given in its respective class.
This declaration is reponsible for declaring the order of the methods in the so-called virtual table which XSharp employs for the invocation of typed methods. A re-declaration of a method in a subclass is NOT permissible, since it would cause abiguity for the compiler.
2.   Define the strongly typed method.
Unlike strongly typed functions, method typing requires strongly typing of the method arguments, the method return value AND speficying a valid calling convention.
The following calling conventions are valid for typed methods: STRICT, PASCAL or CALLBACK.

Examples

This example creates a class of two-dimensional coordinates with methods to initialize the coordinates, draw a grid, and plot the point:

```
FUNCTION Start()
    LOCAL oPointSet AS Point2D
    oPointSet := Point2D{2, 3}
    oPointSet:ShowGrid()
    oPointSet:Plot()

CLASS Point2D                   // Define Point2D class
    INSTANCE x, y AS INT

METHOD Init(iRow, iCol) CLASS Point2D
    x := iRow
    y := iCol
    RETURN SELF

METHOD Plot() CLASS Point2D
    @ x + 11, y + 36 SAY CHR(249)

METHOD ShowGrid() CLASS Point2D
    LOCAL iCounter AS INT
    CLS
    FOR iCounter := 1 TO 21
        IF iCounter = 11
            @ iCounter, 1 SAY REPLICATE(CHR(196), 71)
            @ iCounter, 36 SAY CHR(197)
        ELSE
            @ iCounter, 36 SAY CHR(179)
        ENDIF
    NEXT
```

See Also

ACCESS, ASSIGN, CLASS, FUNCTION, PROPERTY, OPERATOR, CONSTRUCTOR, DESTRUCTOR, EVENT

1.8.4.7.2.8 OPERATOR Statement

Purpose

Declare a method to access a non-exported or virtual instance variable.

Syntax

[Attributes] [Modifiers] **OPERATOR** *<operatortype>*
[([*<idParam>* [AS | REF *<idType>*] [, ...])]
[AS *<idType>*]
[=> <expression>]
CRLF
[*<Body>*]
[END OPERATOR]

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed).

CANNOT be combined with an Expression Body

END OPERATOR   Optional end clause to indicate the end of the OPERATOR
                     entity

See Also

[ASSIGN](#), [CLASS](#), [METHOD](#)

1.8.4.7.2.9 PROPERTY Statement

Purpose

Declare a method to access a non-exported or virtual instance variable.

Syntax

```
[Attributes] [Modifiers] PROPERTY<idName>
[([<idParam> [AS | REF <idType>] [, ...])]
[AS <idType>] [<idConvention>]
[
 AUTO [Attributes] [Modifiers] GET | SET | INIT
  |[ [Attributes] [Modifiers] GET <Expression> ]  [ [Attributes] [Modifiers] SET
<Expression>]   [ [Attributes] [Modifiers] INIT <Expression>]
    CRLF
  |[ [Attributes] [Modifiers] GET <Body > END GET]
  |[ [Attributes] [Modifiers] GET => <Expression>]
  |[ [Attributes] [Modifiers]  SET <Body > END SET]
  |[ [Attributes] [Modifiers] SET => <Expression>]
  |[ [Attributes] [Modifiers]  INIT <Body > END INIT]
  |[ [Attributes] [Modifiers] INIT => <Expression>]
  END PROPERTY
  ]
```

Arguments

Attributes   An optional list of one or more attributes that describe meta
                     information for am entity, such as for example the
                     [TestMethod] attribute on a method/function containing tests
                     in a MsTest class library. Please note that Attributes must
                     be on the same line or suffixed with a semi colon when they
                     are written on the line above that keyword.

<idName>   A valid identifier name for the property that you are defining.
                     Like other methods, properties are entities. Property names
                     must be unique within a class, but can share the same
                     name as other entities in your application.

<idParam>   A parameter variable. A variable specified in this manner is

X#

automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.

AS | REF|OUT|IN <idType>　　Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references.

The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters.

Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]

AS <idType>　　Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.

<Expression>　　Expression that implements the accessor

=> <Expression>　　Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body

<Body>　　Program statements that form the code of this entity.
The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed).
CANNOT be combined with an Expression Body

Description

There are 3 types of property declarations:
- Single line declaration without AUTO clause
- Single line declaration with GET / SET / INIT accessors
- Multi line declaration with GET accessor block and/or SET/INIT accessor bock. For multi line declarations the END PROPERTY is mandatory

INIT accessor declare that a property can only be changed in the constructor of a class. A property cannot have both a SET and an INIT accessor.

See Also

[ASSIGN](#), [CLASS](#), [METHOD](#)

1.8.4.7.3 CLASS Statement (All dialects)

Purpose

Declare a class name to the compiler.

Syntax

[Attributes] [Modifiers] **CLASS** *<idClass>* [**INHERIT** *<idClass>*]
[**IMPLEMENTS** <idInterface>[, <IdInterface2>,..]
[ClassMembers]
END CLASS

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, PROTECTED, HIDDEN, INTERNAL, SEALED, ABSTRACT or STATIC.
<idClass>	A valid identifier name for the class. A class is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a class and a global variable, for example, with the same name.
INHERIT <idClass>	The name of an existing class (called a superclass) from which the new class inherits methods and instance variables (with the exception of HIDDEN).
IMPLEMENTS <idInterface>	The name(s) of the interface(s) that this class implements
ClassMembers	This can be any of Instance Variables, [ACCESS](#), [ASSIGN](#), [CONSTRUCTOR](#), [DESTRUCTOR](#), [EVENT](#), [METHOD](#), [OPERATOR](#), [PROPERTY](#)

Description

After the class name is declared to the compiler, it is followed by 0 or more instance variable declaration statements. You use a class name to declare variables (see GLOBAL and LOCAL statements in this guide) designed to hold instances of a specific class, to instantiate instances of the class, and to define methods (see the METHOD statement in this guide) and subclasses for the class.

Notes

Binding of instance variables: Instance variables can be either early or late bound, depending on how you declare them and how you use them.

Early binding happens if the memory location of a variable is known at compile time. The compiler knows exactly how to reference the variable and can, therefore, generate code to do so.

Late binding is necessary if the memory location of a variable is unknown at compile time. The compiler cannot determine from the program source code exactly where the variable is or how to go about referencing it, so it generates code to look the symbol up in a table. The lookup is performed at runtime.

Since there is no need for a runtime lookup with early bound instance variables, using them instead of late bound variables will significantly improve the performance of your application. The following table summarizes the binding and visibility issues for the four types of instance variables:

Variable Type	Binding	Visibility
EXPORT	Early, if possible	Application-wide for CLASS and module-wide for STATIC CLASS
INSTANCE	Always late	In class and subclasses
HIDDEN	Always early	In class only
PROTECT	Always early	In class and subclasses

Object instantiation: Once you declare a class, you create instances of the class using the class name followed by the instantiation operators, {}. The syntax is as follows:

```
<idClass>{[<uArgList>]}
```

where *<uArgList>* is an optional comma-separated list of values passed as arguments to a special method called Init() (see the METHOD statement in this guide for more information on the Init() method).

Accessing instance variables: The syntax to access an exported instance variable externally (i.e., from any entity that is not a method of its class) is as follows:

```
<idObject>:<idVar>
```

You can access non-exported instance variables only from methods in which they are visible. Within a method, you use the following syntax for accessing all instance variables:

```
[SELF:]<idVar>
```

The SELF: prefix is optional except in the case of an access/assign method (see the ACCESS and ASSIGN statement entries in this guide for more information and the METHOD statement for more information on SELF).
Instance variables are just like other program variables. You can access them anywhere in the language where an expression is allowed.

The prefix [STATIC] is no longer supported by XSharp

Examples

The following example defines two classes, one of which inherits values from the other, and demonstrates how to create a class instance with initial values for the instance variables:

```
FUNCTION Start()
   LOCAL oCust AS Customer
   oCust := Customer{"Louis", 92.07.22, "GA", 987}
   oCust:DisplayAll()
   ? oCust:Name
   ...

// Declare Person class

CLASS Person
   EXPORT Name AS STRING
   INSTANCE Birth AS DATE

// Declare Customer class to inherit from Person

CLASS Customer INHERIT Person
   PROTECT CustNum AS SHORTINT
   INSTANCE Address AS STRING

// Declare method to initialize instance variables
// Note that cName and dBirth are available to the
// Customer class, even though they are not declared
```

```
    // as part of the class – they are inherited from Person

    METHOD Init(cOne, dTwo, cThree, nFour) ;
             CLASS Customer
       Name := cOne
       Birth := dTwo
       Address := cThree
       CustNum := nFour
       RETURN SELF

    // Declare method to display all instance variables

    METHOD DisplayAll() CLASS Customer
       ? "Name:        ", Name
       ? "Birth Date:  ", Birth
       ? "Address:     ", Address
       ? "Number:      ", CustNum
```

See Also

ACCESS, ASSIGN, CONSTRUCTOR, DESTRUCTOR, EVENT, METHOD, OPERATOR, PROPERTY

1.8.4.7.3.1 Instance Variables

Purpose

Declare fields/ instance variables with optional initial values

Syntax

[Attributes] [Modifiers] [INSTANCE] [DIM] <idVar>[[<dimensions>]]   [:= <uValue>]   [, ...] [AS <idType>] [, ...]]

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, PROTECTED, HIDDEN, INTERNAL, SEALED, ABSTRACT or STATIC.

DIM	An optional keyword that specifies that you want to create a variable of a (.Net) array type
INSTANCE	An optional keyword that you have to use when no modifiers are used. Without modifier the fields will become PUBLIC
<idVar>	A valid identifier name for the field to declare.
[<dimensions>]	The initial dimensions for a variable of type array. This may be used with the DIM keyword, in which case it is a .Net array, or without the DIM keyword in which case it is a VO compatible dynamic array.
<uValue>	The initial value to assign to the variable.
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.

1.8.4.7.3.2 Other Classmembers

Normal class members (apart from Instance Variables)

ACCESS
ASSIGN
CONSTRUCTOR
DESTRUCTOR
EVENT
METHOD
OPERATOR
PROPERTY

The DECLARE (ACCESS | ASSIGN | METHOD) <idName> [, <idName>] clause is supported by the compiler but ignored

Nested types

'Regular' classes can also have embedded types:

CLASS
DELEGATE
STRUCTURE
INTERFACE
ENUM

1.8.4.7.4 CLASS Statement (FoxPro dialect)

Purpose

Declare a class name to the compiler.

Syntax

[Attributes] **DEFINE** [Modifiers] **CLASS** *<idClass>* [**AS** *<idParentClass>*] [**OF** <classLib>] [**OLEPUBLIC**]
   [ClassMembers]
   (**ENDDEFINE** | END DEFINE)

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, PROTECTED, HIDDEN, INTERNAL, SEALED, ABSTRACT or STATIC.
<idClass>	A valid identifier name for the class. A class is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a class and a global variable, for example, with the same name.
AS *<idParentClass>*	The name of an existing class (called a superclass) from which the new class inherits methods and instance variables. The AS <idParent> clause is mandatory when compiled with /fox1+, and option when compiled with /fox1-. When compiled with /fox1+ the compiler assumes that the parent class is either the Custom class or a class derived from the Custom class.
OF *<classLib>*	This clause is parsed but ignored in X#.
OLEPUBLIC	This clause is parsed but ignored in X#.

ClassMembers

Examples

See Also

1.8.4.7.4.1 Properties and Fields

Note   This command is only available in the FOXPRO dialect

Purpose

Declare fields and or properties with optional initial values

Syntax

        [FIELD] [modifiers] <IdName>, <IdName> ...] [AS <idType>]
        [[.]Object.] <IdName> = <Expression> ...]

Both syntaxes to declare and/or initialize properties are supported.
We have added an optional AS DataType clause.
We have also added an optional FIELD clause that allows you to declare fields (opposed
to properties)

Arguments

FIELD	When you include the FIELD keyword then the names will be the names of fields in the class
Modifiers	An optional list of modifiers that specify the visibility or scope of the property, such as PUBLIC,. PROTECTED, HIDDEN.
<IdName>	A valid identifier name for the fields or properties to declare.

AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
Expression	The initial value to assign to the field/property

Notes

The way in which properties are implemented depends on the value of the /fox1 compiler switch. When this switch is enabled, then all properties will read/write from a property collection that is declared in the Custom Object. When this switch is NOT enabled then 'normal' auto properties will be declared with a backing field in this class.

1.8.4.7.4.2 IMPLEMENTS clause

Note   This command is only available in the FOXPRO dialect

Purpose

Declare an interface that the clause implements

Syntax

IMPLEMENTS <idInterface> [EXCLUDE] IN TypeLib | TypeLibGUID | ProgID]

Arguments

IMPLEMENTS <idInterface>	The name(s) of the interface(s) that this class implements
EXCLUDE	This clause is not supported
IN [Typelib etc]	This clause is not supported.

1.8.4.7.4.3 ADD OBJECT Clause

Note   This command is only available in the FOXPRO dialect

Purpose

Adds objects from other classes to the class definition.

Syntax

ADD OBJECT [Modifiers] <ObjectName> AS <idType> [NOINIT] [WITH <Propertylist>]]

Arguments

Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, PROTECTED, HIDDEN, INTERNAL
<ObjectName>	A valid identifier name for the fields or properties to declare.
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
NOINIT	This clause is not supported
WITH <PropertyList>	Specifies a list of properties and their values for the object that you add to the class definition.

The object will be instantiated and added to the list of "child objects" in the class. Also a property will be created with the name and type specified in the ADD OBJECT clause.

Notes

The way in which properties are implemented depends on the value of the /fox1 compiler switch. When this switch is enabled, then all properties will read/write from a property collection that is declared in the Custom Object. When this switch is NOT enabled then 'normal' auto properties will be declared with a backing field in this class.

1.8.4.7.4.4 COMMAttrib Clause

Note This command is only available in the FOXPRO dialect

Purpose

Declare COM attributes (not supported)

Syntax

```
        PEMName_COMATTRIB = nFlags | DIMENSION
PEMName_COMATTRIB[numElements]
        [PEMName_COMATTRIB[1] = nFlags
            PEMName_COMATTRIB[2] = cHelpString
            PEMName_COMATTRIB[3] = cPropertyCapitalization
            PEMName_COMATTRIB[4] = cPropertyType
            PEMName_COMATTRIB[5] = nOptionalParams]]
```

This clause is not supported in X#.

1.8.4.7.4.5 FUNCTION and PROCEDURE

Note   This command is only available in the FOXPRO dialect

Purpose

Defines method and event functions and procedures for the class definition.

Syntax

[Modifiers] FUNCTION | PROCEDURE Name[_ACCESS |_ASSIGN]
  ([cParamName | cArrayName[] [AS Type][@]]) [AS Type]
   [HELPSTRING cHelpString] | THIS_ACCESS(cMemberName) [NODEFAULT]
    cStatements
[ENDFUNC | ENDPROC]]

See the topics for FUNCTION and PROCEDURE for more details.

Arguments

_ACCESS	The _ACCESS or _ASSIGN suffixes specify to create an Access or Assign method for a property with the same name.
_ASSIGN	
HELPSTRING	The HELPSTRING clause is not supported in X#
THIS_ACCESS	The THIS_ACCESS clause is not supported in X#
NODEFAULT	The NODEFAULT clause is not supported in X#
ENDFUNC	This may also be written as END FUNCTION
ENDPROC	This may also be written as END PROCEDURE

1.8.4.7.5 CLASS Statement (Xbase++ dialect)

Purpose

Declare a class name to the compiler.

Syntax

[Attributes] [Modifiers] **CLASS** *<idClass>* [**FROM** *<idParentClass>*] [**SHARING**
*<idParentClass>,...>]
   [**IMPLEMENTS** <idInterface>[, <IdInterface2>,..]
   [ClassMembers]
   ENDCLASS

[CLASS] METHOD [<ClassName>:] <MethodName> [([<Parameters,...>])]
[<Body>]

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, allowed values are STATIC, FREEZE and FINAL
<idClass>	A valid identifier name for the class. A class is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a class and a global variable, for example, with the same name.
FROM <idParentClass>	The name of an existing class (called a superclass) from which the new class inherits methods and instance variables (with the exception of HIDDEN). X# does NOT allow multiple inheritance.
SHARING <idParentClass>	This clause is not supported by X#.
IMPLEMENTS <idInterface>	The name(s) of the interface(s) that this class implements
ClassMembers	This may be a list of variable declarations, method declarations and inline method implementations
[CLASS] METHOD	This implements one or more methods outside of the class declaration. The CLASS keyword indicates that it is a STATIC method, as opposed to an INSTANCE method.

ClassMembers

Fields
Method Declarations
Method Implementation

See Also

1.8.4.7.5.1 Fields

Purpose

Declare fields/ instance variables with optional initial values

Syntax

[Visibility :]
[CLASS|STATIC] VAR <idVar,...> [IS <Name>] [IN <SuperClass>] [AS <idType>]
[SHARED]
[READONLY]
[ASSIGNMENT HIDDEN | PROTECTED | EXPORTED]
[NOSAVE]

Arguments

Visibility	This sets the visibility for the variables on the line following the statement. This can be HIDDEN ,PROTECTED, EXPORTED or INTERNAL. The default visibility is HIDDEN.
CLASS | STATIC	Declares that this is a class level field.
<idVar>	A valid identifier name for the field to declare.
IS .. IN ..	This clause is not supported in X#
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
SHARED	This clause is not supported in X#
READONLY	Declares this field as Readonly. It must be initialized in the (class) constructor
ASSIGNMENT ..	This clause is not supported in X#
NOSAVE	This sets the [NonSerializable] attribute on the field.

Note

Xbase++ has a complicated set of rules on how to control read/write access to fields. In .Net we simply use the Visibility .

1.8.4.7.5.2 METHOD Declarations

Purpose

Declare and implement methods for a class, both for instances and the class itself

Syntax

Method forward declaration
[Modifiers] METHOD <MethodName,...> [IS <MethodName>] [IN <SuperClass>]

Access/Assign method forward declaration
[Attributes] ACCESS ASSIGN [CLASS] METHOD <MethodName> [VAR <VarName>] [AS *idType*]
[Attributes] ACCESS | ASSIGN [CLASS] METHOD <MethodName> [VAR <VarName>] [AS *idType*]

Method inline declaration
[Attributes] INLINE [CLASS] METHOD <MethodName>[[([<*idParam*> [AS|REF| OUT|IN *idType*] [, ...])]) [AS *idType*]
[=> <expression>]
CRLF
[<*Body*>]
[END METHOD]

Arguments

Modifiers	An optional list of modifiers that specify info about the method. (DEFERRED , FINAL , INTRODUCE , OVERRIDE , CLASS , SYNC, NEW, STATIC, ASYNC, UNSAFE, EXTERN).
<MethodName,...>	is a comma separated list with the names of the instance methods being declared. The name for a method follows the same convention as function and variable names. It must begin with a underscore or a letter and must contain alpha numeric characters.
IS <MethodName>	The IS methodname clause is not supported by *X#*
IN <SuperClass>	The IN Superclass clause is not supported (and not needed) by *X#*
Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.

CLASS	Optional modifier that specify that the declaration is for a class level method or class level property
ACCESS ASSIGN	Declares a Get/Set method for a property. You can to use one or both of these keywords.
<VarName>	The Get/Set method may have a different name than the property that they implement.
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
INLINE	Specifies that the whole method is included between the CLASS .. ENDCLASS keywords (other methods are so called forward declarations)
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body
END METHOD	Optional end clause to indicate the end of the inline METHOD entity

Note

The visibility of a method is determined by the visibility attribute set with one of the statements EXPORTED:, PROTECTED:, HIDDEN: or INTERNAL:

Special method Names

In Xbase++ there are some reserved method names:

Init   This is the name of the constructor
InitClass   This is the name of the class constructor.

The implementation of constructors in .Net is somewhat different from Xbase++.
- Therefore the class constructor cannot have any parameters.
- The parameters of the Init() method become the constructor parameters.

1.8.4.7.5.3 METHOD Implementation

Note   This command is only available in the Xbase++ dialect

Purpose

Provide the implementation for methods that are forward defined between CLASS .. ENDCLASS

Syntax

    [Attributes] [ACCESS | ASSIGN] [Modifiers] METHOD [<ClassName>:]
<MethodName>[[([<idParam> [AS|REF|OUT|IN <idType>] [, ...])]) [AS <idType>]
    [=> <expression>]
    CRLF
    [<Body>]
    [END METHOD]?

Arguments

Modifiers   An optional list of one or more modifiers. CLASS, STATIC, ABSTRACT, UNSAFE, ASYNC, EXTERN
STATIC is a synonym for CLASS.

Attributes   An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.

ACCESS | ASSIGN	Declares that this method implements a Getter or Setter for a property. This must also be defined as ACCESS or ASSIGN in the class declaration.
ClassName	The name of the class in which the class method is declared. When only one class is declared in the PRG file, the class name is optional. Otherwise <ClassName>: is required.
<idMethod>	A valid identifier name for the method. Method names must be unique within a class, but can share the same name as other entities (including access and assign methods) in your application.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body
END METHOD	Optional end clause to indicate the end of the inline

METHOD entity

Special method Names

In Xbase++ there are some reserved method names:

Init	This is the name of the constructor
InitClass	This is the name of the class constructor.

The implementation of constructors in .Net is somewhat different from Xbase++.
- Therefore the class constructor cannot have any parameters.
- The parameters of the Init() method become the constructor parameters.

1.8.4.7.6 DEFINE Statement

Purpose

Declare a constant name and its value to the compiler.

Syntax

[Modifiers] DEFINE *<idConstant>* := *<uValue>* [AS <idType>]

Arguments

Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, STATIC, INTERNAL, EXPORT and UNSAFE.
<idConstant>	A valid identifier name for the constant. A constant is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a constant and a global variable, for example, with the same name.
<uValue>	A constant value that is assigned to <idConstant>. This value can be a literal representation of one of the data types listed below or a simple expression involving only operators, literals, and other DEFINE constants; however, more complicated expressions (including class instantiation) are not allowed.
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.

Description

Once the constant name and value is declared and initialized with the DEFINE statement, you can not change the value of *<idConstant>* without provoking a compiler error. The constant value *<uValue>* will be used whenever the *<idConstant>* identifier name is encountered in your application.

You can hide a constant name from a routine by declaring a variable with the same name (with LOCAL, MEMVAR, or FIELD). The search order for a variable name is as follows:
1. LOCALs, local parameters, MEMVARs, and FIELDs
2. SELF instance variables (i.e., without *<idObject>*: prefix in class methods)
3. GLOBALs and DEFINEs

Tip: You can perform a conditional build based on the value of a DEFINE constant. See the #ifdef and #ifndef statements in this chapter for more information and examples.

Examples

The following example assigns an application name to the constant *cAppName*. This value is then displayed at the beginning and end of the application run:

```
DEFINE cAppName := "Accounts Payable"
...
FUNCTION Start()
   ? "Start of ", cAppName, " application."
   ...
   ? "End of ", cAppName, " application."
```

See Also

#ifdef, #ifndef, GLOBAL

1.8.4.7.7 DELEGATE Statement

Purpose

Declare a delegate to the compiler.

Syntax

[Attributes] [Modifiers] **DELEGATE** *<idDelegate>*
[Typeparameters]
[([*<idParam>* [AS | REF|OUT|IN *<idType>*] [, ...])]
[AS *<idType>*]
[TypeparameterConstraints]

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, PROTECTED, HIDDEN, INTERNAL, SEALED, ABSTRACT or STATIC.
<idDelegate>	A valid identifier name for the delegate. Delegate names must be unique within a namespace.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
TypeParameterConstraints	Here you can specify constraints for the Type parameters, such as WHERE T IS SomeName or WHERE T IS New

Description

A delegate is a reference type that encapsulates a function or method. Delegates are similar to function pointers in native code languages such as Visual Objects, C and C++, but unlike function pointers, delegates are object-oriented, secure and type-safe.

The **DELEGATE** statement declares a special type of class which is partially implemented by the compiler, and partially implemented by the CLR. All delegates inherit from System.MulticastDelegate.

Every delegate has a signature, which is a combination of its parameter and return value types.

Instantiation

You can explicitly call the delegate constructor like:

```
f := MyDelegate{ NULL, @SomeClass.Test() }
```

for static methods :

```
f := MyDelegate{ SELF, @SomeClass.Test() }
```

for instance methods, it is also possible to write:

```
f := SomeClass.Test
```

for static methods:

```
f := SELF:Test
```

Example

```
DELEGATE MyDelegate( x AS STRING ) AS STRING
```

1.8.4.7.8 ENUM Statement

Purpose

Declare an enum to the compiler.

Syntax

```
[attributes] [Modifiers]
ENUM <idEnumName> [AS type]
      memberName [:= value]
         [...]
END [ENUM]
```

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, PROTECTED, HIDDEN, INTERNAL, SEALED, ABSTRACT or STATIC.
idEnum	A valid identifier name for the enum. Enum names must be unique within a namespace.
AS type	The data type of the enumeration members (optional).
memberName	The name(s) of the enumeration members.

Description

A type declared with the **ENUM** keyword is an "enumeration" type, a type that consists of a set of named constants which are called the enumerator list. Enumeration types implicitly inherit from System.Enum.

An enumeration type has an underlying type, which is the type of the items in the enumerator list. The default underlying type is INT; if the AS clause is specified then the type can be any signed or unsigned integral type except System.Char.

By default, the first member in the enumerator list has a value of 0. The value of every other member is the value of the previous member plus 1. An element's value can be explicitly set to any value in the range of the underlying type by using the assignment operator (:=) followed by either a literal integer value, or an expression that resolves to an integer value at compile time. Two or more members can explicitly set to the same value.

The default value for an **ENUM** E is the expression (E)0., which may or may not represent a member of the enum type, depending on whether or not a member having the value of 0 exists.

Enumeration members are referenced by specifying the name of the enumeration type and the name of the member, separated by a period. If the enumeration type was declared in the namespace currently being compiled, the name of the enumeration type is sufficient. Otherwise, either the fully-qualified name of the enumeration must be used (e.g. System.Windows.Forms.MessageBoxButtons) or the namespace must be imported using the USING directive.

The System.FlagsAttribute attribute may be placed on an **ENUM** to indicate that the elements of the enum may be combined using a bitwise OR operation.

The enum name may include one or more namespace names, separated by periods. If no period is found in the enum name, then the default namespace is assumed. The default namespace is the base name of the output assembly name, unless explicitly overridden by the /ns compiler option.

Example

```
ENUM days
   Sunday      // 0
   Monday      // 1
   Tuesday     // 2
   Wednesday   // 3
   Thursday    // 4
   Friday      // 5
   Saturday    // 6
END CLASS

FUNCTION example1() AS VOID
   LOCAL i AS INT
   i := (INT) days.Friday
   ? days.Friday, i      // prints: Friday 5

ENUM direction
   north         // has default value of 0
   east  := 90
   south := 180
   west  := 270
END ENUM

FUNCTION example2( x AS direction ) AS VOID
   IF x == direction.north
      goNorth()
   ELSEIF x == direction.east
      goEast()
   ELSEIF x == direction.south
      goSouth()
   ELSEIF x == direction.west
      goWest()
   ELSE
      Debug.Assert( "Unknown value for direction" )
   ENDIF

[System.FlagsAttribute];
ENUM CarOptions
   SunRoof        := 0x1
   Spoiler        := 0x2
```

```
        FogLights       := 0x4
        TintedWindows := 0x8
    END ENUM

    FUNCTION example3() AS VOID
        LOCAL options AS CarOptions
        options := CarOptions.SunRoof | CarOptions.FogLights
        ? options        // prints: SunRoof, FogLights
        ? (INT) options  // prints: 5
```

In the first and third examples, note that using the ? statement on an enum type prints out the textual value of the enum. This is because the names of the enum members are stored in the assembly's metadata, and the ToString() method in System.Enum (which every enum type inherits from) uses the metadata to obtain and return the name of the enum member, rather than its underlying numerical value.

Also note that even though the textual names of the enum members are returned from Enum.ToString(), the compiler uses the literal numeric values in the compiled code. So the expression IF x == direction.north for example actually compiles as IF x == 0, because the mapping between enumeration members and their underlying values occurs at compile time, not at runtime. This makes using enumeration types as efficient as #define, while providing much higher level of type safety and compile time error checking. However, this also means that changing the values of enum members can cause existing code that uses the enum to break unless it is recompiled.

Notes

1.8.4.7.9 FUNCTION Statement

Purpose

Declare a function name and an optional list of local variable names to the compiler. When used inside a FoxPro DEFINE CLASS .. ENDDEFINE this declares a method.

Syntax

[Attributes] [Modifiers] **FUNCTION** *<idFunction>*
[Typeparameters]
[([*<idParam>* [AS | REF|OUT|IN *<idType>*] [, ...])]
[AS *<idType>*]
[TypeparameterConstraints]
[*<idConvention>*]
[EXPORT LOCAL]
[DLLEXPORT STRING_CONST]
[=> <expression>]
CRLF

[*<Body>*]
[ENDFUNC | END FUNCTION]

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, STATIC, INTERNAL, EXPORT and UNSAFE. Please note that functions and procedures used as class members in FoxPro compatible classes can have more modifiers.
<idFunction>	A valid identifier name for the function. A function is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a function and a class, for example, with the same name.
TypeParameters	This is supported for methods with generic type arguments. This something like <T> for a method with one type parameter named T. Usually one of the parameters in the parameter list is then also of type T.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.

TypeParameterConstraints	Here you can specify constraints for the Type parameters, such as WHERE T IS SomeName or WHERE T IS New
<idConvention>	Specifies the calling convention for this entity. <idConvention> must be one of the following:
	o CLIPPER
	o STRICT
	o PASCAL
	o CALLBACK
	o THISCALL
	Most calling conventions are for backward compatibility only.
	There are 2 exceptions:
	CLIPPER declares that a method has untyped parameters. This is usually only needed for methods without any declared parameters. Otherwise the compiler will assume CLIPPER calling convention when it detects untyped parameters.
	Methods and Functions in external DLL may have STRICT, PASCAL, CALLBACK
EXPORT LOCAL	This clause is allowed by X# but ignored.
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body
ENDFUNC | END FUNCTION	These (optional)keywords indicate the logical end of the function.

Description

A function is a subprogram comprised of a set of declarations and statements to be executed whenever you refer to <*idFunction*> followed by a pair of parentheses (see Notes section, below).

Functions and procedures (see the PROCEDURE statement in this guide) are the basic procedural programming units. You will use them in your applications to organize computational blocks of code.

STATIC FUNCTION allows you to limit the visibility of a function name to the current module, thereby restricting access to the function. This feature is useful when designing a module that will contain some public routines (i.e., with application-wide visibility) and others that are strictly support routines (i.e., only needed by other routines in the same module).

Simply declare all support functions using STATIC FUNCTION. Doing this gives you two immediate advantages. First, no other module in the application will inadvertently call one of your support routines. Second, since static references are resolved at compile time and public references are resolved at link time, there is no possibility of a name conflict. For example, if you have a static Service() function declared in module X and a public Service() function declared in module Y, all references to Service() in X execute the static version and all other references to Service() in the application execute the public version.

Notes

The Start() function: All applications must either have one function or procedure named Start() or be linked with the GUI Classes library and have a method Start() of CLASS App. Start() serves as the startup routine when the application is executed. Start() cannot declare any parameters and, under normal circumstances, should not return a value. If you want to use strong typing in the declaration statement, you must specify AS USUAL PASCAL.

Exporting locals through code blocks: When you create a code block, you can access local variables defined in the creating entity within the code block definition without having to pass them as parameters (i.e., local variables are visible to the code block). Using this fact along with the fact that you can pass a code block as a parameter, you can export local variables. For example:

```
FUNCTION One() EXPORT LOCAL
    LOCAL nVar := 10 AS INT, cbAdd AS CODEBLOCK
    cbAdd := {|nValue| nValue + nVar}
    ? NextFunc(cbAdd)              // Result:  210

FUNCTION NextFunc(cbAddEmUp)
    RETURN (EVAL(cbAddEmUp, 200))
```

When the code block is evaluated in NextFunc(), *nVar*, which is local to function One(), becomes visible even though it is not passed directly as a parameter.

Calling a function: The syntax to call a function is as follows:

```
<idFunction>([<uArgList>])
```

where *<uArgList>* is an optional comma-separated list of arguments to pass to the named function. The function receives the arguments in the order passed using the parameter variables specified as part of the function declaration.

Note that although the parentheses are not required in the FUNCTION statement if the function has no parameters, they are always required in the invocation.

You can call a function within an expression or as a program statement. If called as a program statement, the return value is ignored.

You can also call a function as an aliased expression, as in:

```
<idAlias>-><idFunction>([<uArgList>])
```

When you do this, the work area associated with *<idAlias>* is selected, the function is executed, and the original work area is reselected. You can specify an aliased expression as a program statement, as you would any other expression.

A function can call itself recursively. This means you can refer to a function in its own *<FunctionBody>*.
The specific manner in which you call a function depends on the calling convention (*<idConvention>*) that you specify (either explicitly or implicitly) when you declare the function.

CLIPPER calling convention: If you declare the function without any data types in the parameter list, the function uses the CLIPPER calling convention by default. You can also specify the CLIPPER calling convention in the FUNCTION declaration statement, providing that you do not use strong typing in the parameter list.
Although it does not allow strongly typed parameters, the CLIPPER calling convention supports strong typing of the function return value.

With the CLIPPER calling convention, the number of parameters declared for the function does not have to match the number of arguments passed when you call the function. You can skip any argument by leaving it out of the list (specifying two consecutive commas) or by omitting it from the end of the list. For example:

```
FUNCTION Start()
   MyFunc(1,, 3)      // Skip second argument
   MyFunc(1, 2)           // Skip final argument

FUNCTION MyFunc(x, y, z)


   ...
```

A parameter not receiving a value is automatically initialized to NIL by the function so that you can check for skipped arguments. You can use PCount() to help determine the number of arguments passed — this function returns the position of the last argument passed.

Any parameter specified in a CLIPPER function can receive arguments passed by value or reference — the semantics are determined when the function is called rather than when it is declared. The default method for expressions and variables is by value. All variables except field variables, when prefaced with the reference operator (@), are

passed by reference. Field variables cannot be passed by reference and are always passed by value.

STRICT calling convention: If you declare the function with any data types in the parameter list, the function uses the STRICT calling convention by default. You can also specify the STRICT calling convention in the FUNCTION declaration statement.

Using the STRICT calling convention, you give up many of the features allowed with the CLIPPER calling convention, but you gain in compilation speed, application integrity, and execution speed by strongly typing the parameters and return value and declaring the passing semantics of the function.

STRICT functions do not support a variable number of arguments, PCount(), or the ability to be used in macro expressions.

Like CLIPPER functions, STRICT functions allow the calling semantics to be determined when the function is called, but only for polymorphic parameters (i.e., those not strongly typed). When a parameter is typed, the calling semantics are also declared depending on whether you use the AS or the REF keyword. AS means that the parameter must be passed by value and REF means that it must be passed by reference (with the reference operator (@)).

PASCAL calling convention: To specify this calling convention, use PASCAL as the last keyword in the FUNCTION declaration statement. Syntactically, the PASCAL calling convention is identical to STRICT and the usage restrictions are the same, but internally it is handled differently. It is identical to the Microsoft Pascal calling convention, and its primary use is for low-level interfacing with Windows.

CALLBACK calling convention: To specify this calling convention, use CALLBACK as the last keyword in the FUNCTION declaration statement. This is a special PASCAL calling convention with Windows prologue and epilogue. It is used for low-level interfacing with Windows.

Parameters: As an alternative to specifying parameters in the FUNCTION declaration statement, you can use a PARAMETERS statement to specify them. This practice, however, is not recommended because it is less efficient and provides no compile-time integrity validation. See the PARAMETERS statement in this guide for more information.

Examples

This example demonstrates a function that formats numeric values as currency:

```
FUNCTION Start()
   ? Currency(1000)          // Result:  $1,000.00

FUNCTION Currency(nNum)
   LOCAL cNum
   IF nNum < 0
         cNum := Transform(-1 * nNum, ;
```

```
                          "999,999,999,999.99")
              cNum := PadL("($" + LTRIM(cNum)+ ")", ;
                      LEN(cNum))
        ELSE
              cNum := Transform(nNum, ;
                      "999,999,999,999.99")
              cNum := PadL("$" + LTRIM(cNum), ;
                      LEN(cNum))
        ENDIF
        RETURN cNum
```

The next example demonstrates a function that takes a string formatted as a comma-separated list and returns an array with one element per item:

```
    aList := ListAsArray("One, Two")
    // Result:  {"One", "Two"}

    FUNCTION ListAsArray(cList)
        LOCAL nPos
        LOCAL aList := {}                // Define an empty array

        DO WHILE (nPos := AT(",", cList)) != 0
                                          // Add a new element
              AADD(aList, SUBSTR(cList, 1, nPos - 1))
              cList := SUBSTR(cList, nPos + 1)
        ENDDO
        AADD(aList, cList)

        RETURN aList                     // Return the array
```

This example checks for a skipped argument by comparing the parameter to NIL:

```
    FUNCTION MyFunc(param1,param2,param3)
        IF param2 = NIL
              param2 := "default value"
        ENDIF ...
```

Here the Currency() function (defined above) is used as an aliased expression:

```
USE invoices NEW
USE customer NEW
? Invoices->Currency(Amount)
```

See Also

1.8.4.7.10 GLOBAL Statement

Purpose

Declare a variable or array that is available to the entire application or module.

Syntax

[Modifiers] GLOBAL *<idVar>* [:= *<uValue>*] [AS | IS *<idType>*]
[Modifiers] GLOBAL DIM *<ArraySpec>* AS | IS *<idType>*

Arguments

Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, STATIC, INTERNAL, EXPORT and UNSAFE.
<idVar>	A valid identifier name for the variable. A global variable is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a global variable and a function, for example, with the same name.
<uValue>	A constant value that is assigned to <idVar>. This value can be a literal representation of one of the data types listed below or a simple expression involving only operators, literals, and DEFINE constants; however, more complicated expressions (including class instantiation) are not allowed.
	Note: Although <uValue> can be a literal array, it must be one-dimensional. Multi-dimensional literal arrays are not allowed. For example, {1, 2, 3} is allowed, but { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} } is not.
	Note: Although the Chr() function cannot be used in <uValue>, the _Chr() operator can. _Chr() is otherwise identical in functionality to Chr().

If <uValue> is not specified, the initial value of the variable depends on the data type you declare (e.g., NIL if you do not use strong typing, 0 for AS INT, etc.)

DIM <ArraySpec>   The specification for a dimensioned array to declare.

<ArraySpec>   The specification for a dynamic array to declare.

In both cases, <ArraySpec> is one of the following:

<idArray>[<nElements>, <nElements>, <nElements>]
<idArray>[<nElements>][<nElements>][<nElements>]
All dimensions except the first are optional.

<idArray> is a valid identifier name for the array to declare. For dynamic arrays, array elements are initialized to NIL. For dimensioned arrays, the initial value of the elements depends on the data type as explained above for <uValue>.

<nElements> defines the number of elements in a particular dimension of an array. The number of dimensions is determined by how many <nElements> arguments you specify.

<nElements> can be a literal numeric representation or a simple numeric expression involving only operators, literals, and DEFINE constants; however, more complicated expressions (such as function calls) are not allowed.

AS <idType>   Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.

IS <idType>   Specifies a VOSTRUCT or UNION data type in which the memory needed to hold the structure is allocated on the stack (<idStructure> is the only <idType> allowed with the IS keyword.) See the VOSTRUCT entry in this guide for more information on data structure memory allocation.

Notes

Search order for variables: You can hide a global variable name from a routine by declaring another variable with the same name (with LOCAL, MEMVAR, or FIELD). The search order for a variable name is as follows:
1.   LOCALs, local parameters, MEMVARs, and FIELDs
2.   SELF instance variables (i.e., without *<idObject>*: prefix in class methods)
3.   GLOBALs and DEFINEs

Examples

The following example illustrates using the GLOBAL statement to create a global variable, a global dimensioned array, and a global dynamic array. The dynamic array, since it is declared with STATIC GLOBAL, is visible only in the current module:

```
GLOBAL cAppName := "Accounts Payable" AS STRING
GLOBAL DIM aiValues[2][10] AS INT
STATIC GLOBAL aPoly[100]
...
FUNCTION Start()
    ? "Start of ", cAppName, " application."
    AFill(aPoly, 0)
    ...
    ? "End of ", cAppName, " application."
```

See Also

DEFINE, LOCAL

1.8.4.7.11 INTERFACE Statement

Purpose

Declare a interface name to the compiler.

Syntax

[Attributes] [Modifiers] **INTERFACE** *<idInterface>* [INHERIT *<idInterface>*]
[InterfaceMembers]
END INTERFACE

Arguments

Attributes   An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.

Modifiers   An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, PROTECTED, HIDDEN, INTERNAL, SEALED, ABSTRACT or STATIC.

<idInterface>	A valid identifier name for the class. A class is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a class and a global variable, for example, with the same name.
INHERIT *<idInterface>*	The name of an existing class (called a superclass) from which the new class inherits methods and instance variables (with the exception of HIDDEN).
InterfaceMembers	This can be any of ACCESS, ASSIGN, METHOD, OPERATOR, PROPERTY, EVENT

Description

Notess

See Also

ACCESS, ASSIGN, CLASS, EVENT, METHOD, OPERATOR, PROPERTY, STRUCTURE

1.8.4.7.12 LOCAL FUNCTION Statement

Purpose

Declare a local function

Syntax

[Modifiers] **LOCAL FUNCTION** *<idFunction>*
[Typeparameters]
[([*<idParam>* [AS | REF|OUT|IN *<idType>*] [, ...])]
[AS *<idType>*]
[TypeparameterConstraints]
[=> <expression>]
CRLF
[*<Body>*]
END FUNCTION

Arguments

[Modifiers]	The only valid modifiers for a local function are UNSAFE and/or ASYNC

<idFunction>	A valid identifier name for the function. A function is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a function and a class, for example, with the same name.
TypeParameters	This is supported for methods with generic type arguments. This something like <T> for a method with one type parameter named T. Usually one of the parameters in the parameter list is then also of type T.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS	REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
TypeParameterConstraints	Here you can specify constraints for the Type parameters, such as WHERE T IS SomeName or WHERE T IS New
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body

END FUNCTION   These **mandatory** keywords indicate the logical end of the function.

Description

A local function is defined as a nested function inside a containing member. The END FUNCTION is mandatory so the compiler knows where the function ends and its surrounding container continues.
In the example below the WAIT command is part of the Start() function and will be executed after result of the call to Fact() is shown.

Example

```
FUNCTION Start AS VOID
    ? Fact(10)
    LOCAL FUNCTION Fact(nNum AS LONG) AS LONG
        IF nNum == 1
            RETURN 1
        ENDIF
        RETURN nNum * Fact(nNum-1)
    END FUNCTION
    WAIT
    RETURN
```

See Also

FIELD, LOCAL, MEMVAR, METHOD, PROCEDURE, RETURN, FUNCTION

1.8.4.7.13 LOCAL PROCEDURE Statement

Purpose

Declare a local procedure

Syntax

[Modifiers] **LOCAL PROCEDURE** *<idFunction>*
[Typeparameters]
[([*<idParam>* [AS | REF|OUT|IN *<idType>*] [, ...])]
[TypeparameterConstraints]
[=> <expression>]
CRLF
[*<Body>*]
END PROCEDURE

Arguments

[Modifiers]	The only valid modifiers for a local function are UNSAFE and/or ASYNC
<idFunction>	A valid identifier name for the function. A function is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a function and a class, for example, with the same name.
TypeParameters	This is supported for methods with generic type arguments. This something like <T> for a method with one type parameter named T. Usually one of the parameters in the parameter list is then also of type T.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
TypeParameterConstraints	Here you can specify constraints for the Type parameters, such as WHERE T IS SomeName or WHERE T IS New
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function

definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed).
CANNOT be combined with an Expression Body

END PROCEDURE   These **mandatory** keywords indicate the logical end of the function.

Description

A local function is defined as a nested function inside a containing member. The END PROCEDURE is mandatory so the compiler knows where the function ends and its surrounding container continues.
In the example below the WAIT command is part of the Start() function and will be executed after the call to Log(3)

Example

```
FUNCTION Start AS VOID
    Log(1)
    Log(2)
    Log(3)

    LOCAL PROCEDURE Log(nNum AS LONG)
        ? nNum
        RETURN
    END PROCEDURE
    WAIT

    RETURN
```

See Also

FIELD, LOCAL, MEMVAR, METHOD, PROCEDURE, RETURN, PROCEDURE

1.8.4.7.14 PROCEDURE Statement

Purpose

Declare a procedure name and formal parameters. When used inside a FoxPro DEFINE CLASS .. ENDDEFINE this declares a method.

Syntax

[Attributes] [Modifiers] **PROCEDURE** *<idProcedure>*
[Typeparameters]
[([*<idParam>* [AS | REF|OUT|IN *<idType>*] [, ...])]
[AS *<idType>*]
[TypeparameterConstraints]
[*<idConvention>*]
[_INIT1 | _INIT2 | _INIT3 | EXIT]
[EXPORT LOCAL]
[DLLEXPORT STRING_CONST]
[=> <expression>]
CRLF
[*<Body>*]
[ENDPROC | END PROCEDURE]

Alternative INIT / EXIT Procedures

[Attributes] [INIT | EXIT] **PROCEDURE** *<idProcedure>*
[Typeparameters]
[([*<idParam>* [AS | REF|OUT|IN *<idType>*] [, ...])]
[AS *<idType>*]
[TypeparameterConstraints]
[*<idConvention>*]
[=> <expression>]
CRLF
[*<Body>*]
[ENDPROC | END PROCEDURE]

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, STATIC, INTERNAL, EXPORT and UNSAFE. Please note that functions and procedures used as class members in FoxPro compatible classes can have more modifiers.
<idProcedure>	A valid identifier name for the function. A function is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a function and a class, for example, with the same name.

TypeParameters	This is supported for methods with generic type arguments. This something like <T> for a method with one type parameter named T. Usually one of the parameters in the parameter list is then also of type T.
<idParam>	A parameter variable. A variable specified in this manner is automatically declared local. These variables, also called formal parameters, are used to receive arguments that you pass when you call the entity.
AS | REF|OUT|IN <idType>	Specifies the data type of the parameter variable (called strong typing). AS indicates that the parameter must be passed by value, and REF indicates that it must be passed by reference with the @ operator. OUT is a special kind of REF parameter that does not have to be assigned before the call and must be assigned inside the body of the entity. IN parameters are passed as READONLY references. The last parameter in the list can also be declared as PARAMS <idType>[] which will tell the compiler that the function/method may receive zero or more optional parameters. Functions or Methods of the CLIPPER calling convention are compiled to a function with a single parameter that this declared as Args PARAMS USUAL[]
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
TypeParameterConstraints	Here you can specify constraints for the Type parameters, such as WHERE T IS SomeName or WHERE T IS New

<idConvention>	Specifies the calling convention for this entity. <idConvention> must be one of the following:
	    o CLIPPER
	    o STRICT
	    o PASCAL
	    o CALLBACK
	    o THISCALL
	Most calling conventions are for backward compatibility only.
	There are 2 exceptions:
	CLIPPER declares that a method has untyped parameters. This is usually only needed for methods without any declared parameters. Otherwise the compiler will assume CLIPPER calling convention when it detects untyped parameters.
	Methods and Functions in external DLL may have STRICT, PASCAL, CALLBACK

EXPORT LOCAL	This clause is allowed by X# but ignored.

_INITn, EXIT	When an application is executed, all INIT procedures in all modules associated with the application (including libraries) are called automatically. There are three priority levels for INIT procedures indicated by the _INIT1, _INIT2, and _INIT3 keywords. _INIT1 procedures are called first, _INIT2 second, and _INIT3 third. All INIT procedures are called before the application Start() routine. EXIT procedures are called at application shutdown
=> <Expression>	Single expression that replaces the multiline body for the entity. CANNOT be compiled with a body
<Body>	Program statements that form the code of this entity. The <Body> can contain one or more RETURN statements to return control to the calling routine and to serve as the function return value. If no return statement is specified, control passes back to the calling routine when the function definition ends, and the function will return a default value depending on the return value data type specified (NIL if the return value is not strongly typed). CANNOT be combined with an Expression Body
ENDPROC | END PROCEDURE	These (optional)keywords indicate the logical end of the function.

You must follow the guidelines below when specifying an INIT procedure:
- No arguments are allowed

Notes

Please not that INIT and EXIT procedures have an INTERNAL scope always. You cannot access these from outside of the assembly where they are defined, so prevent you from accidentally calling them. If you need to call them, then you may consider to store the actual code in a normal function or procedure and call that code from the INIT or EXIT procedure.

Examples

```
PROCEDURE First AS VOID PASCAL _INIT1
```

INIT procedures are necessary for having automatic initialization routines for libraries and other modules in an application besides the main startup module (i.e., the one containing the Start() routine). Although Start() routines are limited to one per application, there are no inherent limits for the total number of INIT procedures within an application.

The following example shows a skeleton of a typical procedure that uses declared variables:

```
PROCEDURE Skeleton(cName, cClassRoom, Bones, nJoints)
    LOCAL nCrossBones, aOnHand := {"skull", "metacarpals"}
    STATIC nCounter := 0

    <Executable Statements>...
```

The next example determines whether an argument was skipped by comparing the parameter to NIL:

```
PROCEDURE MyProc(param1, param2, param3)
    IF param2 != NIL
        param2 := "default value"
    ENDIF
    <Statements>...
```

This example invokes the procedure, UpdateAmount(), as an aliased expression:

```
USE invoices NEW
USE customer NEW
Invoices->UpdateAmount(Amount + Amount * nInterest)
```

See Also

FIELD, FUNCTION, LOCAL, MEMVAR, METHOD,RETURN

1.8.4.7.15 STRUCTURE Statement

Purpose

Declare a class name to the compiler.

Syntax

[Attributes] [Modifiers] **STRUCTURE** *<idStructure>*
[IMPLEMENTS <idInterface>[, <IdInterface2>,..]
[StructureMembers]
END STRUCTURE

Arguments

Attributes	An optional list of one or more attributes that describe meta information for am entity, such as for example the [TestMethod] attribute on a method/function containing tests in a MsTest class library. Please note that Attributes must be on the same line or suffixed with a semi colon when they are written on the line above that keyword.
Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, PROTECTED, HIDDEN, INTERNAL, SEALED, ABSTRACT or STATIC.
<idStructure>	A valid identifier name for the class. A class is an entity and, as such, shares the same name space as other entities. This means that it is not possible to have a class and a global variable, for example, with the same name.
IMPLEMENTS <idInterface>	The name(s) of the interface(s) that this class implements
StructrureMembers	This can be any of ACCESS, ASSIGN, CONSTRUCTOR, DESTRUCTOR, EVENT, METHOD, OPERATOR, PROPERTY, just like in the CLASS declaration

In this case, the variables x and z are typed as INT, while the variables cName and cAddr are typed as STRING.

Description

After the structure name is declared to the compiler, it is followed by 0 or more instance variable declaration statements. You use a structure name to declare variables (see GLOBAL and LOCAL statements in this guide) designed to hold instances of a specific class, to instantiate instances of the class, and to define methods (see the METHOD statement in this guide) and subclasses for the class.

Notes

Binding of instance variables: Instance variables can be either early or late bound, depending on how you declare them and how you use them.

Early binding happens if the memory location of a variable is known at compile time. The compiler knows exactly how to reference the variable and can, therefore, generate code to do so.

Late binding is necessary if the memory location of a variable is unknown at compile time. The compiler cannot determine from the program source code exactly where the variable is or how to go about referencing it, so it generates code to look the symbol up in a table. The lookup is performed at runtime.

Since there is no need for a runtime lookup with early bound instance variables, using them instead of late bound variables will significantly improve the performance of your

application. The following table summarizes the binding and visibility issues for the four types of instance variables:

Variable Type	Binding	Visibility
EXPORT	Early, if possible	Application-wide for CLASS and module-wide for STATIC CLASS
INSTANCE	Always late	In class and subclasses
HIDDEN	Always early	In class only
PROTECT	Always early	In class and subclasses

Object instantiation: Once you declare a class, you create instances of the class using the class name followed by the instantiation operators, {}. The syntax is as follows:

```
<idClass>{[<uArgList>]}
```

where *<uArgList>* is an optional comma-separated list of values passed as arguments to a special method called Init() (see the METHOD statement in this guide for more information on the Init() method).

Accessing instance variables: The syntax to access an exported instance variable externally (i.e., from any entity that is not a method of its class) is as follows:

```
<idObject>:<idVar>
```

You can access non-exported instance variables only from methods in which they are visible. Within a method, you use the following syntax for accessing all instance variables:

```
[SELF:]<idVar>
```

The SELF: prefix is optional except in the case of an access/assign method (see the ACCESS and ASSIGN statement entries in this guide for more information and the METHOD statement for more information on SELF).
Instance variables are just like other program variables. You can access them anywhere in the language where an expression is allowed.

The prefix [STATIC] is no longer supported by XSharp

Examples

The following example defines two classes, one of which inherits values from the other, and demonstrates how to create a class instance with initial values for the instance variables:

See Also

[ACCESS](#), [ASSIGN](#), [CONSTRUCTOR](#), [DESTRUCTOR](#), [EVENT](#), [METHOD](#), [OPERATOR](#), [PROPERTY](#)

1.8.4.7.16 UNION Statement

Purpose

Declare a union entity and its member names.

Syntax

> [Modifiers] UNION *<idUnion>* [ALIGN 1|2|4|8]
> MEMBER *<idVarList>* AS | IS *<idType>* [,…]
> MEMBER DIM *<ArraySpec>* [,…] AS | IS *<idType>* [,…]
> [END UNION]

Note: The MEMBER statement is shown using two syntax diagrams for convenience. You can declare variables and dimensioned arrays using a single MEMBER statement if each definition is separated by a comma.

Arguments

Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, STATIC, INTERNAL, EXPORT and UNSAFE.
<idUnion>	A valid identifier name for the union. A union is an entity and shares the same name space as other entities. This means that it is not possible to have a union and a constant, for example, with the same name.
MEMBER	Declares one or more union member variables or dimensioned arrays. You can specify multiple MEMBER declarations on separate lines.
<idVarList>	A comma-separated list of identifier names for the union member variables.
DIM <ArraySpec>	The specification for a dimensioned array to use as a union member. <ArraySpec> is one of the following: <idArray>[<nElements>, <nElements>, <nElements>] <idArray>[<nElements>],[<nElements>], [<nElements>]

All dimensions except the first are optional.

<nElements> defines the number of elements in a particular dimension of an array. The number of dimensions is determined by how many <nElements> arguments you specify.

<nElements> can be a literal numeric representation or a simple numeric expression involving only operators, literals, and DEFINE constants; however, more complicated expressions (such as function calls) are not allowed.

AS <idType>

Specifies the data type of the variable you are declaring (called strong typing). For DIM arrays, declares the data type for all array elements. The AS <idType> is required for all union members.

Refer to the CLASS entry for a list of valid values for <idType>. Note that the following data types are not supported in unions because they are dynamic types that require garbage collection:
- ARRAY
- FLOAT
- OBJECT
- <idClass>
- STRING
- USUAL
-

IS <idType>

Specifies a union data type in which the memory needed to hold the union is allocated on the stack (i.e., <idUnion> is the only <idType> allowed with the IS keyword).

ALIGN 1|2|4|8

Specifies the memory alignment of the structure. The default is 4, which means that all members are aligned at DWORD boundaries, since that gives the best performance on a 32 bits platform and is also the default alignment for most C/C++ compilers. You may want to change this when you need to match a C/C++ structure that has been defined with a different alignment (the #pragma pack in a C/C++ header file).

Note: The default alignment for C/C++ compilers is 4 as well, unless the structure contains doubles (REAL8 in XSharp). In that case the C/C++ compiler uses an alignment of 8. XSharp does NOT automatically choose an alignment of 8, so you must add the ALIGN 8 to your structure in these circumstances.

Description

UNIONs are like STRUCTUREs, but all members start at offset zero (0). In other words, assigning a value to a union member affects all other union members. As the size of the union is equal to the size of the biggest member, changing one member will change all of the others because they occupy the same memory.

You use the UNION statement to mark the beginning of the definition of a union entity, followed by one or more MEMBER statements that define what the union looks like.

Examples

The following is a conversion example:

```
UNION wb ALIGN 1
   MEMBER w AS WORD
   BYTE bLo AS BYTE
   BYTE bHi AS BYTE

FUNCTION x
 LOCAL u IS wb

 u.w := 0x1234

 ? u.bLo  // 52 (=0x34)
 ? u.bHi  // 18 (=0x12)
```

See Also

STRUCTURE

1.8.4.7.17 VOSTRUCT Statement

Note   This command is only available in the VO and Vulcan dialects

Purpose

Declare a data structure and its member names.

Syntax

[Modifiers] VOSTRUCT *<idStructure>* [ALIGN 1|2|4|8]
       MEMBER *<idVarList>* AS | IS *<idType>* [, ...]
       MEMBER DIM *<ArraySpec>* [, ...] AS | IS *<idType>* [, ...]
[END VOSTRUCT]

Note: The MEMBER statement is shown using two syntax diagrams for convenience. You can declare variables and dimensioned arrays using a single MEMBER statement if each definition is separated by a comma.

Arguments

Modifiers	An optional list of modifiers that specify the visibility or scope of the entity, such as PUBLIC, STATIC, INTERNAL, EXPORT and UNSAFE.
<idStructure>	A valid identifier name for the structure. A structure is an entity and shares the same name space as other entities. This means that it is not possible to have a structure and a constant, for example, with the same name.
MEMBER	Declares one or more structure member variables or dimensioned arrays. You can specify multiple MEMBER declarations on separate lines.
<idVarList>	A comma-separated list of identifier names for the structure member variables.
DIM <ArraySpec>	The specification for a dimensioned array to use as a structure member. <ArraySpec> is one of the following: <idArray>[<nElements>, <nElements>, <nElements>] <idArray>[<nElements>][<nElements>][<nElements>] All dimensions except the first are optional.
	<idArray> is a valid identifier name for the array to declare.
	<nElements> defines the number of elements in a particular dimension of an array. The number of dimensions is determined by how many <nElements> arguments you specify.
	<nElements> can be a literal numeric representation or a simple numeric expression involving only operators, literals, and DEFINE constants; however, more complicated expressions (such as function calls) are not allowed.
AS <idType>	Specifies the data type of the variable you are declaring (called strong typing). For DIM arrays, declares the data type for all array elements. The AS <idType> is required for all structure members.
	Refer to the CLASS entry for a list of valid values for <idType>. Note that the following data types are not supported in structures because they are dynamic types that require garbage collection: ARRAY FLOAT OBJECT <idClass>

STRING
USUAL

IS <idType>   Specifies a structure data type in which the memory needed
                            to hold the structure is allocated on the stack (i.e.,
                            <idStructure> is the only <idType> allowed with the IS
                            keyword).

ALIGN 1|2|4|8   Specifies the memory alignment of the structure. The
                            default alignment is based on the size of the structure
                            members. See the paragraph about alignment below.
                            You may want to change this when you need to match a
                            C/C++ structure that has been defined with a different
                            alignment (the #pragma pack in a C/C++ header file).
                            .

Notes

AS vs. IS: Once you have defined a structure, you can use its name to declare variables
(see GLOBAL and LOCAL statements in this guide) designed to hold instances of a
specific structure. When you declare a structure variable, you have the choice of using AS
or IS typing. The difference between these two declaration methods is as follows:

- IS automatically allocates the memory needed to hold the structure on the stack
  and deallocates the memory when the declaring entity returns.
- AS requires that you allocate memory using MemAlloc() when you initialize
  structure variables. You must also deallocate the memory used by the structure
  variable using MemFree() before the declaring entity returns.

Important! *IS typing is much simpler than AS typing, and in most cases should satisfy
your requirements for using structures. AS typing is recommended for experienced
systems programmers who can, for various reasons, object to using the stack in this
manner.*

Allocating substructures: An interesting property of a structure is that it can contain other
structures as members but, if you type these substructures using AS, you must allocate
and deallocate memory for them. This is true regardless of whether the containing
structure is typed with AS or IS:

```
VOSTRUCT SysOne
    MEMBER iAlpha AS INT
    MEMBER pszName AS PSZ

VOSTRUCT SysTwo
    MEMBER iBeta AS INT
    MEMBER strucOne AS SysOne

FUNCTION UseStruct()
    LOCAL strucVar AS SysTwo
    strucVar := MemAlloc(_SizeOf(SysTwo))
```

X#

```
strucVar.strucOne := MemAlloc(_SizeOf(SysOne))
...
MemFree(strucVar.strucOne)
MemFree(strucVar)
```

To simplify your programming, it makes sense to use IS for declaring substructures. Then, the memory for the substructure will be allocated and deallocated with the memory for its containing structure:

```
VOSTRUCT SysTwo
    MEMBER iBeta    AS INT
    MEMBER strucOne   IS SysOne

FUNCTION UseStruct()
    LOCAL strucVar AS SysTwo
    strucVar := MemAlloc(_SizeOf(SysTwo))
    ...
    MemFree(strucVar)
```

Accessing structure members: Structure variables are complex, the components being members that you declare within the structure. To access a structure member, use the dot operator (.) as follows:

```
<idStructVar>.<idMember>
```

Where *<idStructVar>* is a variable name or dimensioned array element that you have previously declared using a structure name, and *<idMember>* is a variable name or dimensioned array element declared within the VOSTRUCT definition as a MEMBER.

Examples

This example illustrates IS structure typing. No allocation is necessary but you must pass the structure by reference to calls.

```
VOSTRUCT SysOne        // Define SysOne data structure
    MEMBER iAlpha AS INT
    MEMBER pszName AS PSZ

FUNCTION Tester(strucSysOne AS SysOne) AS INT
```

```
    RETURN strucSysOne.iAlpha

    FUNCTION UseStruct()
        LOCAL strucVar IS SysOne
        strucVar.iAlpha := 100
        ? Tester(@strucVar)
        ...
```

This example illustrates AS structure typing. This requires memory allocation and deallocation:

```
    VOSTRUCT SysOne        // Define SysOne data structure
        MEMBER iAlpha AS INT
        MEMBER pszName AS PSZ

    FUNCTION Tester(strucSysOne AS SysOne) AS INT
    RETURN strucSysOne.iAlpha

    FUNCTION UseStruct()
        LOCAL strucVar AS SysOne
        strucVar := MemAlloc(_SizeOf(SysOne))
        strucVar.iAlpha := 100
        ? Tester(strucVar)
        ...
        MemFree(strucVar)
```

With MEMBER, you can list several groups of variable and array names separated by commas and followed by an AS | IS *<idType>* clause to indicate that all names listed are to be typed as indicated. In this example, the variable *x* and the dimensioned array *z* are typed as INT, while the variables *ptrX* and *ptrY* are typed as PTR.

```
    VOSTRUCT SysOne        // Define SysOne data structure
        MEMBER x, DIM z[100] AS INT, ptrX, ptrY AS PTR
```

Default VoStruct Alignment

You can choose to specify an alignment clause in the structure definition or let XSharp determine the best alignment for you.
The default alignment uses the following mechanism:
- Each member of a size <= 8 gets a memory address inside the structure that is a multiple of its size. So WORD and SHORT members get aligned on EVEN boundaries, DWORD, LONG, PTR, PSZ members get aligned to 4-byte boundaries and REAL8

members get aligned to 8- byte boundaries. Byte members are not aligned, they can appear everywhere in the structure.

- The total size of the structure is aligned to the size of the largest member. This is done to make sure that a dim array of structures (multiple structures adjacent in memory) also align properly
- When a structure contains an sub-structure (an IS declaration) the alignment of the outer structure uses the information from the inner structure.

With manual (explicit) alignment each element of the structure is aligned to a memory address that is a multiple of the alignment specified.

Some examples of automatic alignment

```
VOSTRUCT test1              // Offset
   MEMBER W  AS WORD        // 0
   MEMBER dw AS DWORD       // 4
   MEMBER b  AS BYTE        // 8
   // Total size of structure = 12 bytes (largest element = 4, so
padded to 12)
   // Memory layout of structure
   // 0123|4567|8901
   // WW..|DWDW|B...
   //
   // WW   = Word
   // DWDW = Dword
   // B    = Byte
   // .    = Padding

VOSTRUCT test1              // Offset
   MEMBER W  AS WORD        // 0
   MEMBER r8 AS REAL8       // 8
   MEMBER b  AS BYTE        // 16
   // Total size of structure = 24 bytes (largest element = 8, so
padded to 24)
   // Memory layout of structure
   // 01234567|89012345|67890123
   // WW......|R8R8R8R8|B.......
```

Explicit (manual) Structure Alignment

In some situations you need to match a structure declaration from a C/C++ header file that has explicit alignment. Then you need to add the ALIGN clause to your structure declaration.
This forces the compiler to align the structure elements to a multiple of the specified size.
An alignment of 1 tells the compiler NOT to use padding but to align all elements of a structure next to eachother. This is the most compact, but may be slower.

Some examples of explicit alignment

```
VOSTRUCT test1 ALIGN 1        // Offset
   MEMBER W  AS WORD          // 0
   MEMBER r8 AS DWORD         // 2
   MEMBER b  AS BYTE          // 6
   // Total size of structure = 7 bytes (multiple of 1)
   // Memory layout of structure
   // 01|2345|6
   // WW|DWDW|B
   //
   // WW   = Word
   // DWDW = Dword
   // B    = Byte
   // .    = Padding

VOSTRUCT test1 ALIGN 2        // Offset
   MEMBER W  AS WORD          // 0
   MEMBER dw AS DWORD         // 2
   MEMBER b  AS BYTE          // 6
   // Total size of structure = 8 bytes ( multiple of 2)
   // Memory layout of structure
   // 01|23|45|67
   // WW|DW|DW|B.
VOSTRUCT test1 ALIGN 4        // Offset
   MEMBER W  AS WORD          // 0
   MEMBER r8 AS DWORD         // 4
   MEMBER b  AS BYTE          // 8
   // Total size of structure = 12 bytes (multiple of 4)
   // Memory layout of structure
   // 0123|4567|8901
   // WW..|DWDW|B...

VOSTRUCT test1 ALIGN 8        // Offset
   MEMBER W  AS WORD          // 0
   MEMBER r8 AS DWORD         // 8
   MEMBER b  AS BYTE          // 16
   // Total size of structure = 24 bytes (multiple of 8)
   // Memory layout of structure
   // 01234567|89012345|67890123
   // WW.......|DWDW....|B.......
```

See Also

GLOBAL, LOCAL, MemAlloc(), MemFree(),, UNION

1.8.4.8   Environment

1.8.4.8.1 SET ANSI Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines whether database files are created using ANSI or OEM format and whether certain text file operations convert between the two character sets.

Syntax

SET ANSI ON | OFF | (<*IToggle*>)

Arguments

ON, OFF, IToggle   A logical expression which must appear in parentheses. True is equivalent to ON, False to OFF

Description

SET ANSI is functionally equivalent to SetAnsi().

Assembly

XSharp.RT.DLL

See Also

SetAnsi()

1.8.4.8.2 SET CENTURY Command

Purpose

Change the setting that determines whether to include or omit century digits in the date format.

Syntax

```
SET CENTURY ON | OFF | (<lToggle>)
```

Arguments

ON, OFF, lToggle   A logical expression which must appear in parentheses. True is equivalent to ON, False to OFF

Description

SET CENTURY is functionally equivalent to SetCentury().

Assembly

XSharp.RT.DLL

See Also

CToD(), DToC(), DToS(), SetCentury(), SetDateCountry(), SetDateFormat(), SetEpoch(), Today()

1.8.4.8.3 SET DATE Command

Purpose

Change the setting that determines the XSharp date format by selecting from a list of constants with corresponding date formats.

Syntax

```
SET DATE [TO] <kCountrySetting>
```

Description

SET DATE is functionally equivalent to SetDateCountry().

Examples

This example illustrates various system-defined country settings:

```
SET DATE German
? Today()                       // Result:  15.10.19
SET DATE Ansi
? Today()                       // Result:  19.10.15
```

Assembly

XSharp.RT.DLL

See Also

CToD(), DToC(), DToS(), SetCentury(), SetDateCountry(), SetDateFormat(), SetEpoch(), Today()

1.8.4.8.4 SET DATE FORMAT Command

Note   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines the XSharp date format.

Syntax

```
SET DATE FORMAT [TO] <cDateFormat>
```

Description

SET DATE FORMAT is functionally equivalent to SetDateFormat().

Examples

In this example the FORMAT clause directly specifies the date format:

```
SET DATE FORMAT "yyyy:mm:dd"
SetCentury(TRUE)
? Today()                      // Result:  2019:10:15
```

Assembly

XSharp.RT.DLL

See Also

CToD(), DToC(), DToS(), SetCentury(), SetDateCountry(), SetDateFormat(), SetEpoch(), Today(),

1.8.4.8.5 SET DECIMALS Command

Note   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines the number of decimal places used to display numbers.

Syntax

SET DECIMALS TO [<*nDecimals*>]

Arguments

nDecimals   Specifies the minimum number of decimal places to display. The default is two decimal places. The maximum number of decimal places is 18; the minimum is zero

Description

SET DECIMALS TO with no argument is equivalent to SET DECIMALS TO 0. SET DECIMALS is functionally equivalent to SetDecimal().

Assembly

XSharp.RT.DLL

See Also

SetDecimal(), SetDecimalSep(), SetFixed(),

1.8.4.8.6 SET DEFAULT Command

Note This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines the XSharp default drive and directory.

Syntax

SET DEFAULT TO [<*xcPathspec*>]

Description

SET DEFAULT is functionally equivalent to SetDefault().

Assembly

XSharp.Core.DLL

See Also

CurDir(), GetDefault(), GetCurPath, SetDefault(), SetPath()

1.8.4.8.7 SET DIGITFIXED Command

Note This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that fixes the number of digits used to display numeric output.

Syntax

SET DIGITFIXED ON | OFF | (</Toggle>)

Arguments

ON
OFF
lToggle   A logical expression which must appear in parentheses.
                                 True is equivalent to ON, False to OFF

Description

SET DIGITFIXED is functionally equivalent to SetDigitFixed().

Assembly

XSharp.RT.DLL

See Also

SetDigitFixed()

1.8.4.8.8 SET DIGITS Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines the number of digits that will be shown to the left of the decimal point when a number is displayed.

Syntax

SET DIGITS TO [<nDigits>]

Description

SET DIGITS is functionally equivalent to SetDigit().

Assembly

XSharp.RT.DLL

See Also

SetDigit()

1.8.4.8.9 SET DRIVER Command

Purpose

Change the default RDD for the application.

Syntax

```
SET DRIVER TO <idDriverName>
```

Description

SET DRIVER is functionally equivalent to DBSetDriver().

Assembly

XSharp.RT.DLL

See Also

DbSetDriver()

1.8.4.8.10 SET EPOCH Command

Purpose

Change the setting that determines how dates without century digits are interpreted.

Syntax

```
SET EPOCH TO <nYear>
```

Description

SET EPOCH is functionally equivalent to SetEpoch().

Assembly

XSharp.RT.DLL

See Also

CToD(), DToC(), DToS(), SetCentury(), SetDateCountry(), SetDateFormat(), SetEpoch(), Today()

1.8.4.8.11 SET EXACT Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Toggles and exact match for character string comparisons.

Syntax

SET EXACT ON | OFF | (<*lToggle*>)

Arguments

ON, OFF, lToggle   A logical expression which must appear in parentheses. True is equivalent to ON, False to OFF

Description

SET EXACT is functionally equivalent to SetExact().

Assembly

XSharp.RT.DLL

See Also

SetExact()

1.8.4.8.12 SET FIXED Command

Purpose

Change the setting that fixes the number of decimal digits used to display numbers.

Syntax

SET FIXED ON | OFF | (<*lToggle*>)

Arguments

ON, OFF, lToggle   A logical expression which must appear in parentheses. True is equivalent to ON, False to OFF

Description

SET FIXED is functionally equivalent to SetFixed().

Assembly

XSharp.RT.DLL

See Also

Exp(), Log(), SetDecimal), SetFixed(), SQrt(), Val()

1.8.4.8.13 SET PATH Command

Purpose

Change the setting that determines the XSharp search path for opening files.

Syntax

SET PATH TO [*<xcPathSpecList>*]

Description

SET PATH is functionally equivalent to SetPath() except that semicolons are not allowed as separators in SET PATH's *<xcPathSpecList>*.

Assembly

XSharp.Core.DLL

See Also

CurDir(), GetCurPath, SetDefault(), SetPath()

1.8.4.9 File

COPY FILE
DELETE FILE
DIR
ERASE
RENAME
SET DEFAULT
SET PATH

1.8.4.9.1 COPY FILE Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Copy a file to a new file or to a device.

Syntax

```
COPY FILE <xcSourceFile> TO <xcTargetFile> | <xcDevice>
```

Arguments

`<xcSourceFile>`	The name of the source file to copy, including an optional drive, directory, and extension.
	If `<xcSourceFile>` does not exist, a runtime error is raised. If it exists, this command attempts to open the file in shared mode and, if successful, it proceeds. If access is denied because, for example, another process has exclusive use of the file, NetErr() is set to TRUE.
`TO <xcTargetFile>`	The name of the target file, including an optional drive, directory, and extension.
	If `<xcTargetFile>` does not exist, it is created. If it exists, this command attempts to open the file in exclusive mode and, if successful, the file is overwritten without warning or error. If access is denied because, for example, another process is using the file, NetErr() is set to TRUE.
	See SetDefault() and SetPath() for file searching and creation rules. This command does not supply a default extension for either file name.
`TO <xcDevice>`	The name of the target device specified without a trailing colon. When you specify one of the following device names: PRN, LPT1, LPT2, LPT3, COM1, or COM2, COPY FILE searches the Windows section of WIN.INI for a Device entry to use for the printing device. If there is no Device entry, it looks for a Devices entry and presents the user with a list box of devices from which to choose.

Examples

This example copies a file to a new file, then tests for the existence of the new file:

```
COPY FILE test.prg TO real.prg
? File("real.prg")          // Result:  TRUE
```

The next example prints the contents of a file by copying it to the default device:

```
COPY FILE real.prg TO PRN
```

Assembly

XSharp.Core.DLL

See Also

COPY TO, FCopy(),DELETE FILE, RENAME, SetDefault(), SetPath()

1.8.4.9.2 DELETE FILE Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Remove a file from disk.

Syntax

```
DELETE FILE | ERASE <xcSourceFile>
```

Arguments

<xcSourceFile>   The name of the file to delete, including an optional drive, directory, and extension. SetDefault() and SetPath() do not affect this command. It assumes the current Windows drive and directory if none is specified. No default extension is supplied.

Warning! Files must be closed before deleting them.

Examples

This example removes a specified file from disk then tests to see if the file was removed:

```
? File("temp.dbf")          // Result:  TRUE
DELETE FILE temp.dbf
? File("temp.dbf")          // Result:  FALSE
```

Assembly

XSharp.Core.DLL

See Also

CurDir(), FErase(), File(), USE, SetDefault(), SetPath()

1.8.4.9.3 DIR Command

Purpose

Display a listing of files.

Syntax

```
DIR [<xcFileSpec>]
```

Arguments

<xcFileSpec>   The file specification for the directory search. Besides a file name, this specification may also include an optional drive, directory, and extension. The file name and extension may include the standard wildcard characters (* and ?). If you do not specify a drive and directory, this function respects SetDefault().

If the argument is omitted, the default file specification is *.DBF.

If <xcFileSpec> is not specified, the listing includes the file name, date of last update, and number of records. Otherwise, it includes the file name, extension, number of bytes, and date of last update.

Notes

To save the directory listing in an array, use Directory() instead of Dir().

Examples

This example displays all files, all database files, and all text files in the current directory:

```
cFilespec := "*.*"
DIR (cFilespec)              // Display all files
DIR                                 // Display all .dbf files
DIR *.txt                           // Display all text files
```

Assembly

XSharp.RT.DLL

See Also

ADir(), Directory(), SetDefault()

1.8.4.9.4   ERASE Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Remove a file from disk.

Syntax

```
ERASE | DELETE FILE <xcSourceFile>
```

Description

ERASE is the same as DELETE FILE. See DELETE FILE for a complete explanation of this command.

Assembly

XSharp.Core.DLL

See Also

DELETE FILE, FErase()

1.8.4.9.5 RENAME Command

Purpose

Change the name of a file to a new name.

Syntax

```
RENAME <xcSourceFile> TO <xcTargetFile>
```

Arguments

<xcSourceFile>   The name of the source file to rename, including an optional drive, directory, and extension. If <xcSourceFile> does not exist, a runtime error is raised.

TO <xcTargetFile>   The name of the new file, including an optional drive, directory, and extension. If <xcTargetFile> exists or is open, RENAME does nothing.

SetDefault() and SetPath() do not affect this command. It assumes the current Windows drive and directory if none is specified. No default extensions are supplied.

Description

If the target directory is different from the source directory, the file moves to the new directory.

Warning! *Files must be closed before renaming. Attempting to rename an open file will produce unpredictable results. When a database file is renamed, remember that any associated memo file must also be renamed. Failure to do so can compromise the integrity of your application.*

Examples

This example renames a file, checking for the existence of the target file before beginning the RENAME operation:

```
xcOldFile := "oldfile.txt"
xcNewFile := "newfile.txt"
```

```
IF !File(xcNewFile)
   RENAME (xcOldFile) TO (xcNewFile)
ELSE
   ? "File already exists"
ENDIF
```

Assembly

XSharp.RT.DLL

See Also

COPY FILE, CurDir(), DELETE FILE, File(), FErase(), FRename(), RUN, SetDefault(), SetPath()

1.8.4.9.6 SET DEFAULT Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines the XSharp default drive and directory.

Syntax

SET DEFAULT TO [<*xcPathspec*>]

Description

SET DEFAULT is functionally equivalent to SetDefault().

Assembly

XSharp.Core.DLL

See Also

CurDir(), GetDefault(), GetCurPath, SetDefault(), SetPath()

1.8.4.9.7 SET PATH Command

Purpose

Change the setting that determines the XSharp search path for opening files.

Syntax

SET PATH TO [*<xcPathSpecList>*]

Description

SET PATH is functionally equivalent to SetPath() except that semicolons are not allowed as separators in SET PATH's *<xcPathSpecList>*.

Assembly

XSharp.Core.DLL

See Also

CurDir(), GetCurPath, SetDefault(), SetPath()

1.8.4.10 Index/Order

CLOSE
DELETE TAG
FIND
INDEX
REINDEX
SEEK
SET DESCENDING
SET INDEX
SET OPTIMIZE
SET ORDER
SET SCOPE
SET SCOPEBOTTOM
SET SCOPETOP
SET SOFTSEEK
SET UNIQUE

1.8.4.10.1 DELETE TAG Command

Purpose

Delete one or more orders from open index files.

Syntax

```
DELETE TAG <xcOrder> [IN <xcIndexFile>] [, <xcOrder> [IN
<xcIndexFile>]...]
```

Arguments

<xcOrder>
The name of the order to be removed. If <xcOrder> is a NULL_STRING or spaces, it is ignored. If <xcOrder> cannot be found, a runtime error is raised.

IN <xcIndexFile>
The name of an open index file, including an optional drive and directory (no extension should be specified). Use this argument to remove ambiguity when there are two or more orders with the same name in different index files.
If <xcIndexFile> is not open by the current process in the current work area, a runtime error is raised.

Description

DELETE TAG requires that the current database be open in exclusive mode. If this condition is not met when DELETE TAG is invoked, a runtime error is raised.

If you specify an index file name, DELETE TAG deletes the indicated order from that file. Otherwise, the command searches all index files open in the current work area and deletes the first occurrence of *<xcOrder>* that it finds.
If the controlling order is deleted, the database reverts to its natural order and DbSetFilter() scoping.

Note: The RDD determines the order capacity of an index file, and DELETE TAG is supported only by RDDs with multiple-order capabilities. For single-order index files, you must delete the entire file.

Examples

This example illustrates how to delete selected orders from an index file:

```
USE customer VIA "DBFMDX" NEW
Customer->DBSetIndex("customer")

// Delete the Cust01 and Cust02 orders from the
// Customer index file
DELETE TAG Cust01 IN customer, Cust02 IN customer
// or
// Customer->DBDeleteOrder("Cust01", "customer")
// Customer->DBDeleteOrder("Cust02", "customer")
```

Assembly

XSharp.RT.DLL

See Also

DBCreateIndex(), DbCreateOrder(), DbDeleteOrder(), INDEX, SetDefault(), SetPath()

1.8.4.10.2 FIND Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Search an order for the first key matching the specified string, position the record pointer to the corresponding record, and set the Found() flag.

Note: FIND is a compatibility command and is no longer recommended. It is superseded by SEEK. See SEEK for more information.

Syntax

```
FIND <xcKeyValue>
```

Arguments

<xcKeyValue>   Part or all of the order key of a record to search for. When
                                an expression is specified instead of a literal string, FIND is
                                equivalent to SEEK.

Description

If SetSoftSeek() is FALSE and FIND does not find a record, the record pointer is positioned to LastRec() + 1, EOF() returns TRUE, and Found() returns FALSE.

If SetSoftSeek() is TRUE and FIND does not find a record, the record pointer is positioned to the record with the next greater key value, and Found() returns FALSE. In this case, EOF() returns TRUE only if there are no keys in the index greater than the search argument.

If the record is found, Found() is set to TRUE and the record pointer is positioned on the found record.

Examples

These examples show simple FIND results:

```
USE sales INDEX branch NEW
FIND ("500")
? Found(), EOF(), RECNO()        // Result:  FALSE TRUE 85
FIND 200
? Found(), EOF(), RECNO()        // Result:  TRUE FALSE 5
FIND "100"
? Found(), EOF(), RECNO()        // Result:  TRUE FALSE 1
```

Assembly

XSharp.RT.DLL

See Also

EoF(), Found(), RecNo(), SEEK, SET INDEX, SET ORDER, SetSoftSeek()

1.8.4.10.3   INDEX Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Create an index file and add an order to it.

Syntax

```
INDEX ON <uKeyValue> [TAG <xcOrder>] [TO <xcIndexFile>]
[<Scope>] [WHILE <lCondition>] [FOR <lCondition>]
[EVAL <cbEval> [EVERY <nInterval>]
[UNIQUE] [ASCENDING | DESCENDING]
[USECURRENT] [ADDITIVE] [CUSTOM] [NOOPTIMIZE]
```

Note: Although both the TAG and the TO clauses are optional, you must specify at least one of them.

Arguments

<uKeyValue>	The order key expression. The data type of the key expression and all other limitations, including the length of the key and the key expression, are determined by the RDD.
TAG <xcOrder>	The name of the order to be created. For single-order index files, the file name without an extension or path, is the default order name. For multiple-order index files, the order name is required.
TO <xcIndexFile>	The name of the target index file, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension is determined by the RDD and can be obtained using DBOrderInfo(DBOI_INDEXEXT).

In RDDs that support production indexes , the production index file is assumed if <xcIndexFile> is not specified.

If <xcIndexFile> does not exist, it is created.

If <xcIndexFile> exists, the INDEX command must first obtain exclusive use of the file. If the attempt is unsuccessful because, for example, the file is open by another process, NetErr() is set to TRUE.

If the attempt is successful and the RDD specifies that index files can only contain a single order, the current contents of the file is erased before the new order is added to it. If the RDD specifies that index files can contain multiple orders, <xcOrder> is added to <xcIndexFile> if it does not already exist; otherwise it is replaced. |
| <Scope> | The portion of the current database file to process. The default scope is all records. INDEX ignores the |

DbSetFilter() and SetDeleted() settings, as well as any filter imposed by the current controlling order.

The scope is not stored in the index file and is not used for reindexing or update purposes.

WHILE <lCondition>   A condition that each record within the scope must meet, starting with the current record. As soon as the while condition fails, the process terminates. If no <Scope> is specified, having a while condition changes the default scope to the rest of the records in the file.

The while condition is not stored in the index file and is not used for reindexing or update purposes.

FOR <lCondition>   A condition that each record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next record is processed. Duplicate key values are not added to the index file when a for condition is used.

Unlike the while condition and the scope, the for condition is stored as part of the index file and is used when updating or rebuilding the order with DbReindex() or REINDEX. Any limitations on the for condition are determined by the RDD.

Note: If no <Scope>, while condition, or for condition is specified, the index uses the condition specified by DBSetOrderCondition(), if any.

EVAL <cbEval>   A code block that is evaluated at intervals specified by EVERY <nInterval>. The default interval is 1. This is useful in producing a status bar or odometer that monitors the ordering progress. The return value of <cbEval> must be a logical value. If <cbEval> returns FALSE, indexing halts.

EVERY <nInterval>   A numeric expression that determines the number of times <cbEval> is evaluated. This option offers a performance enhancement by evaluating the condition at intervals instead of for every record processed. The EVERY keyword is ignored if you specify no EVAL clause.

UNIQUE   Creates the order with uniqueness as an attribute, which means that only those records with unique key values will be included in the order. If UNIQUE is not specified, SetUnique() is used to determine the order's uniqueness attribute (refer to SetUnique() for more information on how unique orders are maintained).
Note that keys from deleted records are also included in the index, and may hide keys from non-deleted records.

ASCENDING

Specifies that the keys be sorted in increasing order. If neither ASCENDING nor DESCENDING is specified, ASCENDING is assumed.

DESCENDING

Specifies that the keys be sorted in decreasing order. Using this keyword is the same as specifying the Descend() function within <uKeyValue>, but without the performance penalty during order updates. If you create a DESCENDING index, you will not need to use the Descend() function during a SEEK.

Whether an order is ascending or descending is an attribute that is stored in the index file and used for reindexing and update purposes.

USECURRENT

Specifies that only records in the controlling order — and within the current range as specified by OrdSetScope() — will be included in this order. This is useful when you have already created a conditional order and want to reorder the records which meet that condition, and/or to further restrict the records meeting a condition. If not specified, all records in the database file are included in the order.

ADDITIVE

Specifies that any open orders should remain open. If not specified, all open orders are closed before creating the new one. Note, however, that the production index file is never closed.

CUSTOM

For RDDs that support them, CUSTOM specifies that a custom built order will be created. A custom built order is initially empty, giving you complete control over order maintenance. The system does not automatically add and delete keys from a custom built order. Instead, you explicitly add and delete keys using OrdKeyAdd() and OrdKeyDel(). This capability is excellent for generating pick lists of specific records and other custom applications.

NOOPTIMIZE

Specifies that the FOR condition will be optimized. If NOOPTIMIZE is not specified, the FOR condition will be optimized if the RDD supports optimization.

Description

After it is created (or replaced), the new order is added to the order list for the work area. Other orders already associated with the work area, including the controlling order, are unaffected.
If no order list exists for the work area, the type of RDD determines how the controlling order is set. For RDDs that support only single-order index files (such as DBFNTX), the controlling order is set to the order in the specified index file. For RDDs that support multi-order index files (such as DBFMDX), the controlling order is normally set to the first order in the index file.

Notes

RDD support: Not all RDDs support all aspects of the INDEX command.

Examples

The following example creates a single-order index based on the Acct field:

```
USE customer NEW
INDEX ON Customer->Acct TO CuAcct
```

This example creates a conditional order based on a for condition. This index will contain only records whose TransDate is greater than or equal to January 1, 1999:

```
USE invoice NEW
INDEX ON Invoice->TransDate TO InDate ;
   FOR Invoice->TransDate >= CTOD("01/01/2020")
```

The next example creates an order in a multiple-order index file:

```
USE customer NEW VIA "DBFMDX"
INDEX ON Customer->Acct TAG CuAcct TO customer
```

This example creates an order that calls a routine MyMeter() during its creation:

```
DEFINE Mtr_Increment := 10

FUNCTION Start()
   USE customer NEW
   INDEX ON Customer->Acct TO CuAcct EVAL ;
      {|| MyMeter() } EVERY Mtr_Increment

FUNCTION MyMeter()
   STATIC nRecsDone := 0

   nRecsDone += Mtr_Increment
   ? (nRecsDone/LastRec()) * 100

   RETURN TRUE
```

Assembly

XSharp.RT.DLL

See Also

1.8.4.10.4   REINDEX Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Rebuild all orders in the order list of the current work area.

Syntax

```
REINDEX [EVAL <lCondition> [EVERY <nRecords>]
```

Arguments

EVAL <cbEval>   A code block that is evaluated at intervals specified by EVERY <nInterval>. The default interval is 1. This is useful in producing a status bar or odometer that monitors the ordering progress. The return value of <cbEval> must be a logical value. If <cbEval> returns FALSE, indexing halts.

EVERY <nInterval>   A numeric expression that determines the number of times <cbEval> is evaluated. This option offers a performance enhancement by evaluating the condition at intervals instead of for every record processed. The EVERY keyword is ignored if you specify no EVAL clause.

Description

REINDEX is functionally equivalent to DbReindex().

Caution! *REINDEX does not recreate the header of the index file when it recreates the index. Because of this, REINDEX does not help if there is corruption of the file header.*

To guarantee a valid index, always use INDEX ON in place of REINDEX to rebuild damaged index files

Examples

The following example reindexes the orders in the current work area:

```
USE sales INDEX salesman, territory NEW
REINDEX
```

This example reindexes using a progress indicator:

```
USE sales INDEX salesman, territory NEW
REINDEX EVAL NtxProgress() EVERY 10

FUNCTION NtxProgress()
   LOCAL cComplete := LTRIM(STR((RECNO() / ;
            LastRec()) * 100))
   @ 23, 00 SAY "Indexing..." + cComplete + "%"
RETURN TRUE
```

Assembly

XSharp.RT.DLL
a

See Also

DBCreateIndex(), DbCreateOrder(), DbReindex(), INDEX, PACK, SET INDEX, USE

1.8.4.10.5 SEEK Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Search an order for the first key matching the specified expression, position the record pointer to the corresponding record, and set the Found() flag.

Syntax

```
SEEK <uKeyValue> [SOFTSEEK] [LAST]  [[IN|ALIAS] <workarea>]
```

Arguments

<uKeyValue>	An expression to match with an order key value.
SOFTSEEK	If SOFTSEEK is specified (or if SetSoftSeek() is TRUE), the record pointer is positioned to the record with the next higher key value, and Found() returns FALSE after an unsuccessful SEEK. EoF() returns TRUE only if there are no keys in the order greater than <uKeyValue>.
	If SOFTSEEK is not specified and SetSoftSeek() is FALSE, the record pointer is positioned to LastRec() + 1, EOF() returns TRUE, and Found() returns FALSE after an unsuccessful SEEK.
LAST	If LAST is specified, SEEK finds the last occurrence of the specified key value. If LAST is not specified, SEEK finds the first occurrence.
IN|ALIAS <workarea>	Specifies the work area for which the operation must be performed

Description

If the SEEK is successful, Found() is set to TRUE and the record pointer is positioned to the matching record.

Examples

The following example searches for "Doe" using the SEEK command:

```
USE customer NEW
SET INDEX TO customer
SEEK "Doe"

IF Found()
   .
   .  <Statements>
   .
ENDIF
```

The following example performs a soft seek for "Doe" using SOFTSEEK clause of the SEEK command. Note that the SOFTSEEK clause does not have any effect of the SetSoftSeek() flag:

```
USE customer NEW
SET INDEX TO customer

? SetSoftSeek()                    // FALSE
SEEK "Doe" SOFTSEEK
? SetSoftSeek()                    // Still FALSE

IF !Found()
   ? Customer->Name        // Returns next logical
                                   // name after "Doe"

ENDIF
```

Assembly

XSharp.RT.DLL

See Also

DbSeek(), DbSetIndex(), DbSetOrder(), SET INDEX, SET ORDER

1.8.4.10.6 SET DESCENDING Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the descending flag of the controlling order.

Syntax

```
SET DESCENDING ON | OFF | (<lToggle>)
```

Note: The initial default of this setting depends on whether the controlling order was created with descending as an attribute.

Arguments

ON
OFF
lToggle   A logical expression which must appear in parentheses.
                             True is equivalent to ON, False to OFF

Description

SET DESCENDING is functionally equivalent to OrdDescend().

Assembly

XSharp.RT.DLL

See Also

OrdDescend()

1.8.4.10.7   SET INDEX Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a
> function call. If you disable the standard header (-nostddefs) files then this command will not be
> available. If you tell the compiler to use a different standard header file (-stddef) then this
> command may also be not available

Purpose

Open one or more index files and add their orders to the order list in the current work area.

Syntax

```
SET INDEX TO [<xcIndexFileList> [ADDITIVE]]
```

Arguments

TO <xcIndexFileList>   The names of the index files to open, including an optional
                             drive, directory, and extension for each. See SetDefault()
                             and SetPath() for file searching and creation rules. The
                             default extension is determined by the RDD and can be
                             obtained using DBOrderInfo(DBOI_INDEXEXT).

                             If a file in the list does not exist, a runtime error is raised. If it
                             exists, this command attempts to open the file in the same
                             mode as the corresponding database file. If access is
                             denied because, for example, another process is using the

file and this one is asking for exclusive use, NetErr() is set to TRUE. Otherwise, the file open is successful, the command proceeds to the next file in the list.

Concurrency conflicts with index files are rare since they should be used with only one database file. If a concurrency problem arises, it will normally be when you attempt to open the database file.

SET INDEX TO with no file name clears the current order list.

ADDITIVE   Adds the orders from the indicated index files to the current order list, leaving the controlling order intact. If not specified, a new order list is constructed from the indicated index files, replacing the current order list.

Description

If no order list exists for the work area or if SET INDEX is replacing the current order list, the type of RDD determines whether or not the controlling order is set. For RDDs that support only single-order index files (such as DBFNTX), the controlling order is set to the order in the specified index file. For RDDs that support multi-order index files (such as DBFMDX), you usually need to set the controlling order explicitly (using, for example, SET ORDER or DbSetOrder() otherwise, the data file may be processed in natural order even though there is an order list in effect.

After the new index files are opened, the work area is positioned to the first logical record in the controlling order and all subsequent database operations process the records using the controlling order.

During database processing, all orders in the order list are updated whenever a key value is added or changed, respecting any for condition or unique flag in the order. To change the controlling order without affecting the current order list, use SET ORDER or DBSetOrder(). To find out information about a particular order or index file, use DBOrderInfo().

SET INDEX TO when specified with an index file list is functionally equivalent to using several DBSetIndex() function calls. If no ADDITIVE clause is specified, the command calls DBClearFilter() first.

Examples

This example opens a database and several associated index files:

```
USE sales NEW
SET INDEX TO sales, sales1, sales2
```

The next example opens an index file without closing any that are already open:

```
SET INDEX TO sales3 ADDITIVE
```

Assembly

XSharp.RT.DLL

See Also

CLOSE, DBClearIndex(),DbOrderInfo(), DBSetIndex(), DBSetOrder(),INDEX, REINDEX, SET ORDER, SetDefault(), SetPath() USE

1.8.4.10.8 SET ORDER Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Set the controlling order for the current work area.

Syntax

```
SET ORDER TO [<nPosition> | TAG <xcOrder> [IN <xcIndexFile>]]
```

Arguments

<nPosition> | TAG <xcOrder>

The name of the new controlling order or a number representing its position in the order list. Using the order name is the preferred method since the position may be difficult to determine using multiple-order indexes. Specifying a value of 0 has the special effect of returning the database file to its natural order. Specifying an invalid value will raise a runtime error.

Note: The syntax of this command differs from other Xbase dialects where the TAG keyword is optional.

IN <xcIndexFile>

The name of an index file, including an optional drive and directory (no extension should be specified). Use this argument to remove ambiguity when there are two or more orders with the same name in different index files.

If <xcIndexFile> is not open by the current process in the current work area, a runtime error is raised.

SET ORDER TO with no arguments is the same as SET ORDER TO 0.

Description

The controlling order determines the order in which the database file is processed. No matter which order is currently controlling the logical order of the database file, all orders in the order list are properly updated when records are added or updated. This is true even if you SET ORDER TO 0. Changing the controlling order does not move the record pointer.

Before using this command, use SET INDEX or DBSetIndex() to add orders from an index file to the order list.
SET ORDER is functionally equivalent to DBSetOrder().

Examples

The following example illustrates a typical use of SET ORDER to select between several orders:

```
USE customer NEW
INDEX ON Lastname TO names
INDEX ON City + State TO region
SET INDEX TO names, region
//
SET ORDER TO TAG "Region"
? DBOrderInfo(DBOI_EXPRESSION)
// Result:  City + State

SET ORDER TO 0
? DBOrderInfo(DBOI_EXPRESSION)
// Result:  NULL_STRING

SET ORDER TO TAG "Names"
? DBOrderInfo(DBOI_EXPRESSION)
// Result:  Lastname
```

Assembly

XSharp.RT.DLL

See Also

DbOrderInfo(), DbSeek(),DbSetIndex(), DbSetOrder(), INDEX, SEEK, SET INDEX, USE

1.8.4.10.9 SET SCOPE Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the top boundary for scoping key values in the controlling order.

Syntax

```
SET SCOPE TO [<uNewTop> [, <uNewBottom>]]
```

Arguments

<uNewTop>	The top range of key values that will be included in the controlling order's current scope. <uNewTop> can be an expression that matches the data type of the key expression in the controlling order or a code block that returns the correct data type.
<uNewBottom>	The bottom range of key values that will be included in the controlling order's current scope. <uNewBottom> can be an expression that matches the data type of the key expression in the controlling order or a code block that returns the correct data type.
	Note: If <uNewBottom> is not specified, <uNewTop> is taken for both the top and bottom range values.

Description

SET SCOPE, when used with no arguments, clears the top and bottom scopes; this is equivalent to OrdScope(0, NIL) followed by OrdScope(1, NIL). If *<uNewTop>* is specified alone, SET SCOPE sets the top and bottom scope to the indicated value (that is, OrdScope(0, *<uNewTop>*) followed by OrdScope(1, *<uNewTop>*). If both *<uNewTop>* and *<uNewBottom>* are specified, SET SCOPE sets the top and bottom scope as indicated (that is, OrdScope(0, *<uNewTop>*) followed by OrdScope(1, *<uNewBottom>*)..

Assembly

XSharp.RT.DLL

See Also

OrdScope()

1.8.4.10.10 SET SCOPEBOTTOM Command

Purpose

Change the bottom boundary for scoping key values in the controlling order.

Syntax

```
SET SCOPEBOTTOM TO [<uNewBottom>]
```

Description

SET SCOPEBOTTOM, when used with the *<uNewBottom>* argument, is functionally equivalent to OrdScope(1, *<uNewBottom>*). SET SCOPEBOTTOM, when used with no argument, is functionally equivalent to OrdScope(1, NIL).

Assembly

XSharp.RT.DLL

See Also

OrdScope()

1.8.4.10.11 SET SCOPETOP Command

Purpose

Change the top boundary for scoping key values in the controlling order.

Syntax

```
SET SCOPETOP TO [<uNewTop>]
```

Description

SET SCOPETOP, when used with the *<uNewTop>* argument, is functionally equivalent to OrdScope(0, *<uNewTop>*). SET SCOPETOP, when used with no argument, is functionally equivalent to OrdScope(0, NIL).

Assembly

XSharp.RT.DLL

See Also

OrdScope()

1.8.4.10.12 SET SOFTSEEK Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines whether a seek action will find a close match when no exact match is found.

Syntax

```
SET SOFTSEEK ON | OFF | (<lToggle>)
```

Arguments

ON
OFF
lToggle   A logical expression which must appear in parentheses.
                            True is equivalent to ON, False to OFF

Description

SET SOFTSEEK is functionally equivalent to SetSoftSeek().

Assembly

XSharp.RT.DLL

See Also

DbSeek(),, Found(), SEEK, SET INDEX, SET ORDER, SET RELATION, SetSoftSeek()

1.8.4.10.13 SET UNIQUE Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines whether to include unique record keys in an order.

Syntax

```
SET UNIQUE ON | OFF | (<lToggle>)
```

Arguments

ON
OFF
lToggle   A logical expression which must appear in parentheses.
                                True is equivalent to ON, False to OFF

Description

SET UNIQUE is functionally equivalent to SetUnique().

Assembly

XSharp.RT.DLL

See Also

DBCreateIndex(), INDEX, PACK, REINDEX, SEEK, SetUnique()

1.8.4.11 International

SET COLLATION
SET INTERNATIONAL

1.8.4.11.1 SET COLLATION Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines the internal collation routine used for string comparisons.

Syntax

```
SET COLLATION TO WINDOWS | clipper
```

Description

SET COLLATION is functionally equivalent to SetCollation(), which you can refer to for more information and examples.

Assembly

XSharp.RT.DLL

See Also

SET INTERNATIONAL, SetCollation()

1.8.4.11.2 SET INTERNATIONAL Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines the international mode for the application.

Syntax

```
SET INTERNATIONAL TO WINDOWS | CLIPPER
```

Description

SET INTERNATIONAL is functionally equivalent to SetInternational(), which you can refer to for more information and examples.

Assembly

XSharp.RT.DLL

See Also

SET COLLATION, SetInternational()

1.8.4.12 Macros

& Command

1.8.4.12.1 & Command

Enter topic text here.

1.8.4.13 Memory Variable

CLEAR MEMORY
DECLARE
MEMVAR
PARAMETERS
PRIVATE
PUBLIC
RELEASE
RESTORE
SAVE
STORE

1.8.4.13.1 CLEAR MEMORY Command

Purpose

Delete both public and private memory variables from the memory variable table.

Syntax

CLEAR MEMORY

Description

It operates in contrast to RELEASE ALL which does not actually delete public and private memory variables, but assigns NIL to those whose scope is the current procedure. CLEAR MEMORY is the only way to delete all public memory variables from current memory. Declared variables and constants, however, are unaffected by CLEAR MEMORY.

Assembly

XSharp.RT.DLL

See Also

MemClear(), RELEASE

1.8.4.13.2 DECLARE / DIMENSION Statement

Purpose

Create and initialize private variables of the FoxPro array type
Note: and Declare are synonyms

```
DIMENSION <arrayName> ( <nRows> [, <nColumns>] ) [, <arrayName>
( <nRows> [, <nColumns>] ) ]     // FoxPro dialect only
DIMENSION <arrayName> [ <nRows> [, <nColumns>] ] [, <arrayName>
[ <nRows> [, <nColumns>] ] ]     // FoxPro dialect only
DECLARE   <arrayName> ( <nRows> [, <nColumns>] ) [, <arrayName>
( <nRows> [, <nColumns>] ) ]     // FoxPro dialect only
```

```
DECLARE   <arrayName> [ <nRows> [, <nColumns>] ] ] [, <arrayName>
[ <nRows> [, <nColumns>] ] ]      // FoxPro dialect only
```

Arguments

arrayName>   Variable name of array . The array will have the dimensions
                              as declared with <nRows> and <nColumns>. The array
                              may be declared with parentheses as delimiters but also
                              with square brackets.
                              <nColumns> is optional
                              We recommend the use of square brackets.

See Also

[PUBLIC](#), [LOCAL](#)

1.8.4.13.3 MEMVAR Statement

Note   This command is not available in the Core and Vulcan dialects

Purpose
Declare one or more memory variable names to be used by the current routine.

Syntax

```
MEMVAR <idMemvarList>
```

Arguments

<idMemvarList>   A list of public and private variable names to declare to the
                              compiler.

Description

When you use the MEMVAR statement to declare variables, unaliased references to
variables in *<idMemvarList>* are treated as if they were preceded by the special memory
variable alias
(_MEMVAR->).

Like other variable declaration statements (such as LOCAL and FIELD), you must place
MEMVAR statements before any executable statements (including PRIVATE, PUBLIC,
and PARAMETERS) in the routine you are defining. The MEMVAR statement has no
effect on the macro operator, which always assumes memory variables.

The MEMVAR statement neither creates the variables nor verifies their existence. Its primary purpose is to ensure correct references to variables that are known to exist at runtime. Attempting to access variables before they are created will raise a runtime error.

Examples

This example demonstrates the relationship between a private and field variable with the same name. The private variable is declared with the MEMVAR statement:

```
FUNCTION Example()
   MEMVAR Amount, Address
   PRIVATE Amount := 100
   USE customer NEW

   ? Amount            // Refers to Amount private variable
   ? Customer->Amount      // Refers to Amount field variable
```

See Also

FIELD, LOCAL, PARAMETERS, PRIVATE, PUBLIC, STATIC

1.8.4.13.4 PARAMETERS Statement

Note   This command is not available in the Core and Vulcan dialects

Purpose

Create private variables to receive passed values or references.

Syntax

```
PARAMETERS <idParameterList>
PARAMETERS <Parameter1> [ AS <Type> [ OF <ClassLibrary> ] ] [,
<Parameter2> [ AS <Type> [ OF <ClassLibrary> ] ] ] // FoxPro only
```

Arguments

<idParameterList>	One or more parameter variables separated by commas. These variables are used to receive arguments that you pass when you call the routine. The variables will be dynamic memory variables
<Type> & <ClassLibrary>	The compiler recognizes the AS <Type> and the AS

<Type> of <Classlibrary> clauses in the FoxPro dialect.

Description

When a PARAMETERS statement executes, all variables in the parameter list are created as private variables and all public or private variables with the same names are hidden until the current procedure or function terminates. A PARAMETERS statement is an executable statement and can, therefore, occur anywhere in a procedure or function. Parameters can also be declared as local variables if specified as a part of the PROCEDURE or FUNCTION declaration statement (see the example). Parameters specified in this way are referred to as formal parameters. Note that you cannot specify both formal parameters and a PARAMETERS statement within a procedure or function definition.
Attempting to do this results in a compiler error.
The number of receiving variables does not have to match the number of arguments passed by the calling routine. If you specify more arguments than parameters, the extra arguments are ignored. If you specify fewer arguments than parameters, the extra parameters are created with a NIL value. If you skip an argument, the corresponding parameter is initialized to NIL.
The PCount() function returns the position of the last argument passed in the list of arguments. This is different than the number of parameters passed, since it includes skipped parameters.

Examples

This function receives values passed into private parameters with a PARAMETERS statement:

```
FUNCTION MyFunc()
PARAMETERS cOne, cTwo, cThree
? cOne, cTwo, cThree
```

The next example is similar, but receives values passed into local variables, by declaring the parameter variables within the FUNCTION declaration:

```
FUNCTION MyFunc(cOne, cTwo, cThree)
? cOne, cTwo, cThree
```

See Also

LPARAMETERS, PRIVATE

1.8.4.13.5 PRIVATE statement

Note   This command is not available in the Core and Vulcan dialects

Purpose

Create variables and arrays visible within current and invoked routines.

Syntax

```
PRIVATE <idVar> [:= <uValue>] | <ArraySpec> [, ...]
PRIVATE <idVar> [:= <uValue>] [AS <Type> [OF <ClassLibrary>] ]  //
FoxPro dialect
```

Arguments

<idVar>	A valid identifier name for the private variable to create.
<uValue>	The initial value to assign to the variable. If not specified, the variable is initialized to NIL.
<ArraySpec>	The specification for a dynamic array to create. <ArraySpec> is one of the following: <idArray>[<nElements>, <nElements>, <nElements>] <idArray>[<nElements>][<nElements>][<nElements>] All dimensions except the first are optional. <idArray> is a valid identifier name for the array to create. Array elements are initialized to NIL. <nElements> defines the number of elements in a particular dimension of an array. The number of dimensions is determined by how many <nElements> arguments you specify.
<Type> & <ClassLibrary>	The compiler recognizes the AS <Type> and the AS <Type> of <Classlibrary> clauses in the FoxPro dialect.

Description

PRIVATE is an executable statement which means you must specify it after any variable declaration statements (such as FIELD, LOCAL, and MEMVAR) in the routine that you are defining.

Warning! *Any reference to a variable created with this statement will produce a compiler error unless the Undeclared Variables compiler option is checked.*

When you create a private variable or array, existing and visible private and public variables of the same name are hidden until the current routine terminates or the private variable is explicitly released.

Attempting to specify a private variable that conflicts with a visible declared variable (for example, LOCAL, GLOBAL, or DEFINE) of the same name is not recognized by the compiler as an error because PRIVATE is not a compiler declaration statement.   Instead, the declared variable will hide the public variable at runtime. This means that you will not be able to access the public variable at all until the declared variable is released.

In class methods, instance variables (with the exception of access/assign variables) are always more visible than private variables of the same name. Use the _MEMVAR-> alias to access a private variable within a method if there is a name conflict. For access/assign variables, use the SELF: prefix to override a name conflict with a private variable.

In addition to the PRIVATE statement, you can create private variables by:
- Assigning to a variable that does not exist or is not visible will create a private variable
- Receiving parameters using the PARAMETERS statement

Private variables are dynamically scoped. They exist until the creating routine returns to its caller or until explicitly released with CLEAR ALL, or CLEAR MEMORY.

Notes

Compatibility: The ALL, LIKE, and EXCEPT clauses of the PRIVATE statement supported by other Xbase dialects are not supported.

Examples

The following example creates two PRIVATE arrays and three other PRIVATE variables:

```
PRIVATE aArray1[10], aArray2[20], var1, var2, var3
```

The next example creates a multi-dimensional private array using each element addressing convention:

```
PRIVATE aArray[10][10][10], aArray2[10, 10, 10]
```

This example uses PRIVATE statements to create and initialize arrays and variables:

```
PRIVATE aArray := { 1, 2, 3, 4 }, ;
     aArray2 := ArrayNew(12, 24)
PRIVATE cChar := Space(10), cColor := SetColor()
```

See Also

[LOCAL](), [MEMVAR](), [PARAMETERS](), [PUBLIC](), [DIMENSION](), [DECLARE]()

1.8.4.13.6 PUBLIC Statement

Purpose
Creates variables and arrays visible to all routines in an application.

Syntax

```
PUBLIC <memVarList>
PUBLIC <idVar> [:= <uValue>] | <ArraySpec> [, ...]
PUBLIC <idVar> [:= <uValue>] [AS <Type> [OF <ClassLibrary>] ]
                            // FoxPro dialect only
PUBLIC ARRAY <arrayName> ( <nRows> [, <nColumns>] ) [, <arrayName>
( <nRows> [, <nColumns>] ) ]    // FoxPro dialect only
PUBLIC ARRAY <arrayName> [ <nRows> [, <nColumns>] ] [, <arrayName>
[ <nRows> [, <nColumns>] ] ]    // FoxPro dialect only
```

Arguments

<memVarList>	One or more variable names separated by commas.
<idVar>	A valid identifier name for the public variable to create. The idVar may be prefixed with an ampersand (such as PUBLIC &name). In that case the compiler the idVar should contain a string with the name of the variable that is declared and initialized.
<uValue>	The initial value to assign to the variable. This can be any valid expression. If <uValue> is not specified, the variable is initialized to FALSE. There are exceptions: In the FoxPro dialect the PUBLIC FOX and FOXPRO are initialized with TRUE. In the other dialects the PUBLIC CLIPPER is initialized with TRUE.
<ArraySpec>	The specification for a dynamic array to create. <ArraySpec> is one of the following: <idArray>[<nElements>, <nElements>, <nElements>] <idArray>[<nElements>][<nElements>][<nElements>] All dimensions except the first are optional. <idArray> is a valid identifier name for the array to create. Array elements are initialized to NIL. <nElements> defines the number of elements in a particular dimension of an array. The number of dimensions is determined by how many <nElements> arguments you specify.
<Type> & <ClassLibrary>	The compiler recognizes the AS <Type> and the AS <Type> of <Classlibrary> clauses in the FoxPro dialect.

	The <Type> and <ClassLibrary> clauses are ignored because dynamic memory variables are always of type USUAL
<arrayName>	Variable name of array . The array will have the dimensions as declared with <nRows> and <nColumns>. The array may be declared with parentheses as delimiters but also with square brackets. <nColumns> is optional We recommend the use of square brackets.

Description

PUBLIC is an executable statement which means you must specify it after any variable declaration statements (i.e., FIELD, LOCAL, and MEMVAR) in the routine that you are defining.

Warning! *Any reference to a variable created with this statement will produce a compiler error unless the Undeclared Variables compiler option is checked.*

Any declared variables, such as LOCALs, with the same names as existing public or private variables temporarily hide the public or private variables until the overriding variables are released or are no longer visible.

An attempt to create a public variable with the same name as an existing and visible private variable is simply ignored; however the assignment portion of the PUBLIC statement is not ignored. For example, the following lines of code change the value of the variable *x* but do not change its scope from private to public:

```
PRIVATE x := 1000
...
PUBLIC x := "New value for x"
? x                                // Result:  "New value for x"
```

The explanation for this behavior is that, internally, the PUBLIC statement and the assignment are treated as separate statements. Thus, this code would be treated as follows:

```
PRIVATE x := 1000
...
PUBLIC x
x := "New value for x"
? x                                // Result:  "New value for x"
```

The PUBLIC statement is ignored, but the assignment statement is executed, changing the value of the private variable *x*.

This behavior has an interesting repercussion when you declare a public array using a variable name that already exists as private. For example:

```
PRIVATE x := 1000
...
PUBLIC x[10]
? x[1]                  // Result:  NIL
```

In this case, the PUBLIC statement is also treated as two separate statements:

```
PRIVATE x := 1000
...
PUBLIC x
x := ArrayNew(10)
? x[1]                  // Result:  NIL
```

Again, the PUBLIC statement is ignored, and the assignment changes *x* from a private numeric variable to a private reference to a ten element array.

Attempting to specify a public variable that conflicts with a visible declared variable (for example, LOCAL, GLOBAL, or DEFINE) of the same name is not recognized by the compiler as an error because PUBLIC is not a compiler declaration statement. Instead, the declared variable will hide the public variable at runtime. This means that you will not be able to access the public variable at all until the declared variable is released.

In class methods, instance variables (with the exception of access/assign variables) are always more visible than public variables of the same name. Use the _MEMVAR-> alias to access a public variable within a method if there is a name conflict. For access/assign variables, use the SELF: prefix to override a name conflict with a public variable. Public variables are dynamically scoped. They exist for the duration of the application or until explicitly released with CLEAR ALL or CLEAR MEMORY.

Notes

PUBLIC Clipper: To include XSharp extensions in an application and still allow the application to run under dBASE III PLUS, the special public variable, Clipper, is initialized to TRUE when created with PUBLIC.

Examples

This example creates two PUBLIC arrays and one PUBLIC variable:

```
PUBLIC aArray1[10, 10], var2
PUBLIC aArray2[20][10]
```

The following PUBLIC statements create variables and initialize them with values:

```
PUBLIC cString := Space(10), cColor := SetColor()
PUBLIC aArray := {1, 2, 3}, ;
   aArray2 := ArrayNew(12, 24)
```

See Also

GLOBAL, MEMVAR, PARAMETERS, PRIVATE, DIMENSION, DECLARE

1.8.4.13.7 RELEASE Command

Note	This command is not available in the Core and Vulcan dialects
Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Releases public and private memory variables visible to the current routine by assigning a NIL value to them.

Syntax

```
RELEASE <idMemvarList>
RELEASE ALL [LIKE | EXCEPT <Skeleton>]
```

Arguments

<idMemvarList>   A list of private or public variables to release. Specifying a variable name that does not exist or is not visible raises a runtime error.

ALL   Releases all private variables and leaves public variables intact.

LIKE | EXCEPT <Skeleton>

                        Specifies a set of visible private variables to release (LIKE) or keep (EXCEPT) and leaves public variables intact. <Skeleton> can include literal characters as well as the standard wildcard characters, * and ?. If no variables match the <Skeleton>, nothing happens.

Description

This command does not actually delete the specified variables from memory like CLEAR ALL or CLEAR MEMORY. Instead, it releases the value of the variables by assigning NIL to them. For this reason, variables that are hidden do not become visible until termination of the routine initiating the RELEASE operation.

Note: Declared variables and constants are not affected by the RELEASE command.

Assembly

XSharp.RT.DLL

See Also

CLEAR MEMORY, LOCAL, PRIVATE, PUBLIC, QUIT

1.8.4.13.8 RESTORE Command

Note	This command is not available in the Core and Vulcan dialects
Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Recreate public and private variables previously saved to a file and initialize them with their former values.

Syntax

```
RESTORE FROM <xcSourceFile> [ADDITIVE]
```

Arguments

<xcSourceFile>   The name of the memory file (created, for example, with SAVE), including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension is .MEM.

This command attempts to open <xcSourceFile> in shared mode. If the file does not exist, a runtime error is raised. If the file is successfully opened, the operation proceeds. If access is denied because, for example, another process has exclusive use of the file, NetErr() is set to TRUE.

ADDITIVE Causes memory variables loaded from the memory file to be added to the existing pool of memory variables.

Description

The scope of the variable is not saved with the variable but is instead established when the variable is restored. Arrays and declared variables cannot be saved or restored.

When memory variables are restored, they are recreated as private variables with the scope of the current procedure or function unless they exist as public variables and you specify the ADDITIVE clause . If ADDITIVE is specified, public and private variables with the same names are overwritten unless hidden with PRIVATE. If ADDITIVE is not specified, all public and private variables are released before the memory file is loaded.

When restoring variables that were saved in a CLIPPER program, the variable names are truncated to 10 characters. This is because CLIPPER honors only the first 10 characters and generates the .MEM file using only these characters. XSharp, however, honors all characters. For example, in CLIPPER, the two variable names ThisIsALongVariable and ThisIsALon refer to the same variable; this is not the case in XSharp.

Note: Declared variables are not affected by the RESTORE command. If a variable has been declared in the current routine, and a variable with the same name is restored, only the declared variable is visible unless references to the restored variable are prefaced with the _MEMVAR-> alias.

Examples

This example demonstrates a typical application of SAVE and RESTORE. Here memory variables containing screens are created using SAVE TO and RESTORE FROM:

```
// Create and use a pseudo array of screens
SAVE SCREEN TO cScreen1
SAVE ALL LIKE cScreen* TO Screens

<Statements>...

RESTORE FROM Screens ADDITIVE
nNumber = "1"
RESTORE SCREEN FROM ("cScreen" + nNumber)
```

Assembly

XSharp.RT.DLL

See Also

LOCAL, PRIVATE, PUBLIC, SAVE, SetDefault(), SetPath()

1.8.4.13.9 SAVE Command

Purpose

Copy public and private memory variables visible within the current routine to a disk file.

Syntax

```
SAVE TO <xcTargetFile> [ALL [LIKE | EXCEPT <Skeleton>]
```

Arguments

<xcTargetFile>   The name of the file, including an optional drive, directory, and extension. See SetDefault() and SetPath() for file searching and creation rules. The default extension is .MEM.

If <xcTargetFile> does not exist, it is created. If it exists, this command attempts to open the file in exclusive mode and, if successful, the file is overwritten without warning or error. If access is denied because, for example, another process is using the file, NetErr() is set to TRUE.

ALL   Saves all private and public variables.

LIKE | EXCEPT <Skeleton>

Specifies a set of visible public and private variables to save (LIKE) or exclude (EXCEPT). <Skeleton> can include literal characters as well as the standard wildcard characters, * and ?. If no variables match the <Skeleton>, nothing happens.

Description

The scope of the variable is not saved but is instead established when the variable is restored. Arrays and declared variables cannot be saved or restored.

Examples

This example saves all visible private and public variables to TEMP.MEM:

```
PRIVATE cOne := "1"
SAVE ALL TO temp
```

This example saves all visible private and public variables with names beginning with c to MYVARS.MEM:

```
SAVE ALL LIKE c* TO myvars
```

This example saves all visible private and public variables with names that do not begin with c to MYVARS2.MEM:

```
SAVE ALL EXCEPT c* TO myvars2
```

Assembly

XSharp.RT.DLL

See Also

PRIVATE, PUBLIC, RESTORE, SetDefault(), SetPath()

1.8.4.13.10 STORE Command

Note   This command is not available in the Core and Vulcan dialects
Note   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Assign a value to one or more variables.

Syntax

```
STORE <uValue> TO <idVarList>
```

Arguments

<uValue>	A value to assign to the specified variables.
TO <idVarList>	Defines a list of one or more variables that are assigned the value <uValue>. If any variable reference in the list is ambiguous (that is, not declared at compile time or not explicitly qualified with an alias), it is assumed to be MEMVAR. If any variable in the list is not visible or does not exist, a private variable is created using <uValue>.

Description

The STORE command is defined using the assignment operator (:=).

Notes

Assigning a value to an entire array: In XSharp, neither the STORE command nor the assignment operators can assign a single value to an entire array. Use the AFill() function for this purpose.

Examples

These statements create and assign values to undeclared private variables:

```
STORE "string" TO cVar1, cVar2, cVar3
cVar1 := "string2"
cVar2 := _MEMVAR->cVar1
```

These statements assign multiple variables using both STORE and the inline assignment operator (:=). The methods produce identical code:

```
STORE "value" TO cVar1, cVar2, cVar3
cVar1 := cVar2 := cVar3 := "value"
```

These statements assign values to the same field referenced explicitly with an alias. The first assignment uses the field alias (_FIELD->), where the second uses the actual alias name:

```
USE sales NEW
STORE 1200.98 TO _FIELD->CustBal
STORE 1200.98 TO Sales->CustBal
```

See Also

AFill(), , LOCAL, PRIVATE, PUBLIC, RELEASE, REPLACE, RESTORE, SAVE, STATIC

1.8.4.14 Numeric

SET DECIMALS
SET DIGITFIXED
SET DIGITS
SET FIXED

1.8.4.14.1 SET DECIMALS Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that determines the number of decimal places used to display numbers.

Syntax

SET DECIMALS TO [<*nDecimals*>]

Arguments

nDecimals Specifies the minimum number of decimal places to display. The default is two decimal places. The maximum number of decimal places is 18; the minimum is zero

Description

SET DECIMALS TO with no argument is equivalent to SET DECIMALS TO 0. SET DECIMALS is functionally equivalent to SetDecimal().

Assembly

XSharp.RT.DLL

See Also

SetDecimal(), SetDecimalSep(), SetFixed(),

1.8.4.14.2 SET DIGITFIXED Command

Purpose

Change the setting that fixes the number of digits used to display numeric output.

Syntax

SET DIGITFIXED ON | OFF | (<*lToggle*>)

Arguments

ON
OFF
lToggle   A logical expression which must appear in parentheses.
                                 True is equivalent to ON, False to OFF

Description

SET DIGITFIXED is functionally equivalent to SetDigitFixed().

Assembly

XSharp.RT.DLL

See Also

SetDigitFixed()

1.8.4.14.3 SET DIGITS Command

Purpose

Change the setting that determines the number of digits that will be shown to the left of the decimal point when a number is displayed.

Syntax

SET DIGITS TO [<*nDigits*>]

Description

SET DIGITS is functionally equivalent to SetDigit().

Assembly

XSharp.RT.DLL

See Also

SetDigit()

1.8.4.14.4 SET FIXED Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Change the setting that fixes the number of decimal digits used to display numbers.

Syntax

SET FIXED ON | OFF | (<*lToggle*>)

Arguments

ON, OFF, lToggle   A logical expression which must appear in parentheses. True is equivalent to ON, False to OFF

Description

SET FIXED is functionally equivalent to SetFixed().

Assembly

XSharp.RT.DLL

See Also

Exp(), Log(), SetDecimal), SetFixed(), SQrt(), Val()

1.8.4.15 Program Control

1.8.4.15.1 #ifdef Statement

Purpose

Build a section of code if a constant exists and is not equal to FALSE or 0.

Syntax

```
#ifdef <idConstant>
    <Statements>...
[#else]
    <Statements>...
#endif
```

Arguments

<idConstant>   The name of a constant whose existence is being verified.

Description

#ifdef...#endif lets you perform a conditional build by identifying a section of source code to include if a specified constant exists and is not equal to FALSE or 0. The #else statement specifies the code to include if the #ifdef condition fails, and the #endif terminates the conditional build block.

Note: You can use #ifdef with compiler entities other than constants, such as functions and globals. In these cases, the statement tests for existence only, and does not look at the value of the entity.

Examples

This code fragment is a general skeleton for a conditional build with #ifdef. Because the *lDemo* constant is defined as TRUE, the code between the #ifdef and #endif statements will be built:

```
DEFINE lDemo := TRUE
FUNCTION Start()
   #IFDEF lDemo
         <Demo-specific statements>...
   #ENDIF
   ...
```

To build the real (non-demo) version of this application, you would change the DEFINE statement to:

```
DEFINE lDemo := FALSE
```

See Also

#ifndef, DEFINE

1.8.4.15.2 #ifndef Statement

Purpose

Build a section of code if a constant is FALSE, 0, or not defined.

Syntax

```
#ifndef <idConstant>
   <Statements>...
```

```
[#else]
    <Statements>...
#endif
```

Arguments

<idConstant>   The name of a constant whose absence is being verified.

Description

#ifndef...#endif lets you perform a conditional build by identifying a section of source code to include if a specified constant is defined as FALSE or 0 or is not defined. The #else statement specifies the code to include if the #ifndef condition fails, and the #endif terminates the conditional build block.

Note: You can use #ifndef with compiler entities other than constants, such as functions and globals. In these cases, the statement tests for existence only, and does not look at the value of the entity.

Examples

This code fragment is a general skeleton for a conditional build with #ifndef. Since the constant *lDebug* is set to FALSE, the code between the #ifndef and #else statements will be built, and the code between the #else and #endif statements will be ignored:

```
DEFINE lDebug := FALSE

FUNCTION Start()
  #IFNDEF lDebug
        <Optimized version OF code>...
  #ELSE
        <Debugging version OF code>...
  #ENDIF
```

Changing the DEFINE statement as follows will build the code between the #else and #endif statements instead.

```
DEFINE lDebug := TRUE
```

See Also

#ifdef, DEFINE

1.8.4.15.3 ASYNC .. AWAIT

Purpose

ASYNC await are strictly speaking not statements, but modifiers.

ASYNC is a method modifier used to indicate that a method / function contains asynchronous code

AWAIT is used before an expression to indicate that an operation may take a while to process.

The compiler will (behind the scenes) construct a complicated mechanism in which the body of the method is split into a part before and after the await. When the expression returns then the code jumps to the point where it has to continue. As such this belongs to the JUMP statements.

Example

```
//
// This example shows that you can call an async task and wait for
it to finish
// The result of the async task (in this case the size of the file
that has been downloaded)
// will be come available when the task has finished
// The calling code (The Start()) function will not have to wait
until the async task has
// finished. That is why the line "2....." will be printed before
the results from TestClass.DoTest()
// The sample also shows an event and displays the thread id's.
You can see that the DownloadFileTaskAsync() method
// starts multiple threads to download the web document in
multiple pieces.

USING System
USING System.Threading.Tasks

FUNCTION Start() AS VOID
    ? "1. calling long process"
    TestClass.DoTest()
    ? "2. this should be printed while processing"
    Console.ReadKey()

CLASS TestClass
    STATIC PROTECT oLock AS OBJECT      // To make sure we
synchronize the writing to the screen
    STATIC CONSTRUCTOR
```

```
        oLock := OBJECT{}

    ASYNC STATIC METHOD DoTest() AS VOID
        LOCAL Size AS INT64
        Size := AWAIT LoooongProcess()
        ? "3. returned from long process"
        ? Size, " Bytes downloaded"

    ASYNC STATIC METHOD LoooongProcess() AS Task<INT64>
        VAR WebClient := System.Net.WebClient{}
        VAR FileName := System.IO.Path.GetTempPath()+"temp.txt"
        webClient:DownloadProgressChanged += OnDownloadProgress
        webClient:Credentials :=
System.Net.CredentialCache.DefaultNetworkCredentials
        AWAIT
webClient:DownloadFileTaskAsync("http://www.xsharp.info/index.php"
, FileName)
        VAR dirInfo      :=
System.IO.DirectoryInfo{System.IO.Path.GetTempPath()}
        VAR Files        := dirInfo:GetFiles("temp.txt")
        IF Files:Length > 0
            System.IO.File.Delete(FileName)
            RETURN Files[1]:Length
        ENDIF
        RETURN 0

    STATIC METHOD OnDownloadProgress (sender AS OBJECT, e AS
System.Net.DownloadProgressChangedEventArgs) AS VOID
        BEGIN LOCK oLock
          ? String.Format("{0,3} % Size: {1,8:N0} Thread {2}",
100*e:BytesReceived / e:TotalBytesToReceive , e:BytesReceived, ;
            System.Threading.Thread.CurrentThread:ManagedThreadId)
        END LOCK
        RETURN

END CLASS
```

1.8.4.15.4 BEGIN SEQUENCE Statement

Purpose

Define a sequence of statements for a BREAK.

Syntax

```
BEGIN SEQUENCE
    <Statements>...
[BREAK [<uValue>]]
    <Statements>...
[RECOVER [USING <idVar>]]
    <Statements>...
END [SEQUENCE]
```

Arguments

BREAK <uValue>

Branches execution to the statement immediately following the nearest RECOVER statement if one is specified, or the nearest END SEQUENCE statement. <uValue> is the value returned into the <idVar> specified in the USING clause of the RECOVER statement.

RECOVER USING <idVar>

A recover point in the SEQUENCE construct where control branches after a BREAK statement. If this clause is specified, <idVar> receives the value returned by the BREAK statement. In general, this is an error object. <idVar> must be a declared variable and cannot be strongly typed.

END

The end point of the SEQUENCE control structure. If no RECOVER statement is specified, control branches to the first statement following the END statement after a BREAK.

Description

BEGIN SEQUENCE...**END** is a control structure used for exception and runtime error handling. It delimits a block of statements defining a discrete operation, including invoked procedures and functions. With the exception of the **BREAK** statement, the entire construct must fall within the same entity definition.

When a BREAK is encountered anywhere in a block of statements following the **BEGIN SEQUENCE** statement up to the corresponding **RECOVER** statement, control branches to the program statement immediately following the **RECOVER** statement. If a **RECOVER** statement is not specified, control branches to the statement following the **END** statement, terminating the **SEQUENCE**. If control reaches a **RECOVER** statement without encountering a **BREAK**, it branches to the statement following the corresponding **END**.

The **RECOVER** statement optionally receives a parameter passed by a **BREAK** statement that is specified with a return value. This is usually an error object, generated

and returned by the current error handling block defined by ErrorBlock(). If an error object is returned, it can be sent messages to query information about the error. With this information, a runtime error can be handled within the context of the operation rather than in the current runtime error handler.

You cannot **RETURN**, **LOOP**, or **EXIT** between a **BEGIN SEQUENCE** and **RECOVER** statement

Control structures can be nested to any depth. The only requirement is that each control structure be properly nested.

Examples

This code fragment demonstrates a **SEQUENCE** construct in which the BREAK occurs within the current procedure:

```
BEGIN SEQUENCE
    <Statements>...
    IF lBreakCond
        BREAK
    ENDIF
RECOVER
    <Recovery Statements>...
END

<Recovery Statements>...
```

This example demonstrates an error handler returning an error object to the variable specified in the USING clause of the RECOVER statement:

```
LOCAL oLocal, bLastHandler
// Save current and set new error handler
bLastHandler := ErrorBlock({|oErr| ;
    MyHandler(oErr, TRUE)})

BEGIN SEQUENCE
    .
    .  <Operation that might fail>...
    .
RECOVER USING oLocal

    // Send messages to oLocal & handle the error
    ? "Error:  "
    IF oLocal:GenCode != 0
       ?? oLocal:Description
```

```
        ENDIF
        .
        .
        .
    END

    // Restore previous error handler
    ErrorBlock(bLastHandler)

    FUNCTION MyHandler(oError, lLocalHandler)
        // Handle locally returning the error object
        IF lLocalHandler
            BREAK oError
        ENDIF

        <Other statements to handle the error>...
```

This example re-executes a **SEQUENCE** statement block by issuing a **LOOP** from within the **RECOVER** statement block:

```
    DO WHILE TRUE
        BEGIN SEQUENCE

            <Operation that may fail>...

        RECOVER
            IF PrintRecover()
                    // Repeat the SEQUENCE statement block
                LOOP
            ENDIF
        END
        EXIT        // Escape from the operation

    ENDDO
```

See Also

_Break(), CanBreak(), Error Class, ErrorBlock(), RETURN

1.8.4.15.5 BREAK statement

Purpose

The BREAK statement raises a runtime exception.

Syntax

BREAK [*expression*]

Arguments

expression   An optional expression to throw.

Remarks

BREAK throws a runtime exception, causing execution to branch to the nearest
RECOVER, CATCH or FINALLY block in a BEGIN SEQUENCE-RECOVER USING or
TRY construct. If execution is not within a BEGIN SEQUENCE or TRY construct, the
application will terminate.
The specified expression will be evaluated and received by the nearest RECOVER USING
statement, if any, as a value of type USUAL. If the nearest RECOVER statement does not
have a USING clause, the result of expression is discarded.
If expression is not specified, it defaults to NIL.

Example

```
FUNCTION foo
LOCAL e AS USUAL
BEGIN SEQUENCE
   bar( 1 )
RECOVER USING e
   ? "An exception has occurred, exception value is:", e
END SEQUENCE

FUNCTION bar( x )
  IF Valtype(x) != STRING
    BREAK "Argument not a string!"
  ENDIF
  ...
RETURN
```

1.8.4.15.6 CANCEL Command

Purpose

Terminate the application, close all open files, and return control to the operating system.

Syntax

```
CANCEL | QUIT
```

Description

You can use either command from anywhere in an application. A RETURN executed from the Start() routine performs the same action.

Notes

Usage: Both commands can be used from anywhere in an application. A RETURN executed at the highest level procedure or a BREAK, with no pending SEQUENCE, can also be used to QUIT an application.

Examples

This example uses QUIT in a dialog box:

```
IF DialogYesNo(10, 10, "Quit application", ;
        BG+/B,B/W", 2)
    QUIT
ENDIF
```

Assembly

XSharp.RT.DLL

See Also

BEGIN SEQUENCE, ErrorLevel(), RETURN

1.8.4.15.7 DEFAULT Command

Purpose

Assign a default value to a NIL argument.

Syntax

```
DEFAULT <idVar> TO <uDefault> [, <idVar> TO <uDefault>...]
```

Description

DEFAULT is functionally equivalent to Default(), except that DEFAULT lets you assign multiple default values.

Assembly

XSharp.RT.DLL

1.8.4.15.8 DO CASE Statement

Purpose

Conditionally execute a block of statements.

Syntax

```
DO CASE
CASE <lCondition>
   <Statements>...
[CASE <lCondition>]
   <Statements>...
[OTHERWISE]
   <Statements>...
END[CASE]
```

Arguments

<lCondition>	If this expression evaluates to TRUE, the statements following it up until the next CASE, OTHERWISE, or ENDCASE are executed. Afterwards, control branches to the statement following the next ENDCASE statement.
OTHERWISE	If all preceding CASE conditions evaluate to FALSE, the statements following the OTHERWISE up until the next ENDCASE are executed. Afterwards, control branches to the statement following the next ENDCASE statement.

Description

DO CASE works by branching to the statement following the first CASE <lCondition> that evaluates to TRUE. If all CASE conditions evaluate to FALSE, it branches to the statement following the OTHERWISE statement (if specified).

Execution proceeds until the next CASE, OTHERWISE, or ENDCASE is encountered, and control then branches to the first statement following the next ENDCASE statement. Control structures can be nested to any depth. The only requirement is that each control structure be properly nested.

Note: DO CASE...ENDCASE is identical to IF...ELSEIF...ENDIF, with neither syntax having a performance advantage over the other.

Examples

This example uses DO CASE in a menu structure to branch control based on user selection:

```
FUNCTION ActonChoice(nChoice as LONG) AS VOID
DO CASE
CASE nChoice = 0
   RETURN
CASE nChoice = 1
   ChoiceOne()
CASE nChoice = 2
   ChoiceTwo()
ENDCASE
```

See Also

BEGIN SEQUENCE, DO WHILE, FOR, FOREACH IF SWITCH

1.8.4.15.9 DO Statement

Purpose

Call a procedure or function, and optionally pass arguments to the called routine.

Syntax

```
DO <idProcedure> [WITH <uValueArgList>]
```

Arguments

<idProcedure>   The name of the procedure or function to execute.

WITH <uValueArgList>   A comma-separated list of arguments to pass to
                                 <idProcedure>.

Description

DO performs the same action as a function or procedure specified on a line by itself with the exception that variables other than field variables are passed by reference as the default.
In order to pass a field variable as an argument, enclose it in parentheses, unless you declare it with the FIELD statement or specify it with an alias.

Examples

This example executes a procedure with no parameters:

```
DO AcctsRpt
AcctsRpt()                    // Preferred method
```

The next example executes a procedure passing two constants:

```
DO QtrRpt WITH "2nd", "Sales Division"
// Preferred method
QtrRpt("2nd", "Sales Division")
```

In this example, a procedure is executed with the first argument passed by value and the second passed by reference:

```
nNumber := 12
DO YearRpt WITH nNumber + 12, nNumber
// Preferred method
YearRpt(nNumber + 12, @nNumber)
```

Here, a procedure is invoked with skipped arguments embedded in the list of arguments:

```
DO DisplayWindow WITH ,,,,"My Window"
// Preferred method
DisplayWindow(,,,,"My Window")
```

See Also

FIELD, FUNCTION, LOCAL, PARAMETERS, PRIVATE, PROCEDURE, PUBLIC, RETURN

1.8.4.15.10 DO WHILE Statement

Purpose

Execute a loop while a condition is TRUE.

Syntax

[DO] WHILE <*lCondition*>
    <*Statements*>...
[EXIT]
    <*Statements*>...
[LOOP]
    <*Statements*>...
END[DO]

Arguments

<lCondition>	The logical control expression for the DO WHILE loop.
EXIT	Unconditionally branches control from within a FOR, FOREACH , REPEAT or DO WHILE statement to the statement immediately following the corresponding ENDDO or NEXT statement.
LOOP	Branches control to the most recently executed FOR, FOREACH , REPEAT or DO WHILE statement.

Description

When the condition evaluates to TRUE, control passes into the structure and proceeds until an EXIT, LOOP, or ENDDO is encountered. ENDDO returns control to the DO WHILE statement and the process repeats itself. If the condition evaluates to FALSE, the DO WHILE construct terminates and control passes to the statement immediately following the ENDDO.

Use EXIT to terminate a DO WHILE structure based on a condition other than the DO WHILE condition. LOOP, by contrast, prevents execution of statements within a DO

WHILE based on an intermediate condition, and returns to the most recent DO WHILE statement.

Control structures can be nested to any depth. The only requirement is that each control structure be properly nested.

Examples

This example demonstrates a typical control structure for a simple grouped report:

```
LOCAL cOldSalesman, nTotalAmount
USE sales INDEX salesman NEW
DO WHILE .NOT.  EOF()
   cOldSalesman := Sales->Salesman
   nTotalAmount := 0
   DO WHILE cOldSalesman = Sales->Salesman ;
            .AND.  (.NOT.  EOF())
      ? Sales->Salesman, Sales->Amount
      nTotalAmount := nTotalAmount + Sales->Amount
      SKIP
   ENDDO
   ? "Total:  ", nTotalAmount, "for", cOldSalesman
ENDDO
CLOSE sales
```

This code fragment demonstrates how LOOP can be used to provide an intermediate processing condition:

```
DO WHILE <lCondition>
   <Initial Processing>...
   IF <Intermediate Condition>
        LOOP
   ENDIF
   <Continued Processing>...
ENDDO
```

The next example demonstrates the use of DO WHILE to emulate a "repeat until looping" construct:

```
LOCAL lMore := TRUE
DO WHILE lMore
   <Statements>...
   lMore := <lCondition>
ENDDO
```

Here, a DO WHILE loop moves sequentially through a database file:

```
DO WHILE .NOT. EOF()
    <Statements>...
    SKIP
ENDDO
```

See Also

BEGIN SEQUENCE, DBEval(), DO CASE, FOR, IF, RETURN, EXIT, LOOP

1.8.4.15.11 EXIT Statement

Purpose

Unconditionally branches control from within a FOR, FOREACH , REPEAT or DO WHILE statement to the statement immediately following the corresponding ENDDO or NEXT statement.

Syntax

```
EXIT
```

Remarks

The EXIT statement is only valid within a DO WHILE, FOR ... NEXT, FOREACH .. NEXT or REPEAT .. UNTIL construct. An EXIT statement outside of either construct will raise a compile time error.

See Also

FOR
FOREACH
DO WHILE
REPEAT UNTIL

1.8.4.15.12 EXTERNAL Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Declare a list of routines (procedures or functions) to be linked into the application.

Syntax

```
EXTERNAL <idRoutineList>
```

Arguments

<idRoutineList>   A comma-separated list of routine names to link. The names should not include parentheses.

Description

EXTERNAL is a declaration statement that specifies one or more routines to be linked into the application. It should be placed after the variable declaration statements (such as LOCAL)

During the compilation of X# source code, all explicit references to routines are automatically linked. In some instances however, there can be no references made to a routine until runtime. EXTERNAL resolves this by forcing the named routines to be linked even if they are not explicitly referenced in the source code. This is important in several instances:

- Routines referenced in macro expressions or variables
- Functions used in index keys and not otherwise referenced in the source code

Examples

This example forces the code for HardCR(), Tone(), MemoTran(), and StrTran() to be linked into the application, regardless of whether these functions are referenced explicitly in the source code:

```
EXTERNAL HardCR, Tone, MemoTran, StrTran
```

1.8.4.15.13 FOR Statement

Purpose

Execute a block of statements a specified number of times.

Syntax

```
FOR [<idCounter> := <nStart> | VAR <idCounter> := <nStart> | LOCAL
<idCounter> := <nStart> AS <idType> ] [TO | UPTO | DOWNTO] <nEnd>
[STEP <nDelta>]
    <Statements>...
    [EXIT]
    <Statements>...
    [LOOP]
NEXT
```

Note

In the FoxPro and Xbase++ dialect ENDFOR is allowed as alternative for NEXT

Arguments

<idCounter>	The name of the loop control or counter variable. IF a LOCAL or VAR clause is included then the local is created for the duration of the loop. With the VAR clause the datatype is inferred from the usage.
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
<nStart>	The initial value assigned to <idCounter>. If the loop is counting up, <nStart> must be less than <nEnd>. If the loop is counting down, <nStart> must be greater than <nEnd>.
TO <nEnd>	The final value of <idCounter>. The TO clause can be used for counting up or down, depending on whether the STEP clause gives a positive or negative value for <nDelta>. Note, however, that your code will be more efficient if you avoid the TO clause and specify UPTO or DOWNTO instead.
UPTO <nEnd>	The final value of <idCounter>. The UPTO clause is used for counting up.
DOWNTO <nEnd>	The final value of <idCounter>. The DOWNTO clause is used for counting down.
STEP <nDelta>	The amount <idCounter> is changed for each iteration of the loop. If used with the TO clause, <nDelta> can be either positive or negative. With UPTO and DOWNTO, <nDelta> should be positive. If the STEP clause is not specified, <idCounter> is incremented (or decremented in the case of DOWNTO) by one for each iteration of the loop.

X#

EXIT	Unconditionally branches control from within a FOR, FOREACH , REPEAT or DO WHILE statement to the statement immediately following the corresponding ENDDO or NEXT statement.
LOOP	Branches control to the most recently executed FOR, FOREACH , REPEAT or DO WHILE statement.

Description

The control structure loops from the initial value of *<idCounter>* to the boundary specified by *<nEnd>*, moving through the range of values of the control variable for an increment specified by *<nDelta>*. All expressions in the FOR statement are reevaluated for each iteration of the loop. The *<nStart>* and *<nEnd>* values, therefore, can be changed as the control structure operates.

A FOR loop operates until *<idCounter>* is greater than or less than *<nEnd>* (depending on whether you are counting up or down) or an EXIT statement is encountered. Control then branches to the statement following the corresponding NEXT statement. If a LOOP statement is encountered, control branches back to the current FOR statement. Control structures can be nested to any depth. The only requirement is that each control structure be properly nested.

Tip: Although FOR loops are useful for traversing arrays (as demonstrated in the examples below), your code will be more efficient if there is a corresponding array function designed to do what you want.

Examples

This example traverses an array in ascending order:

```
nLenArray := ALen(aArray)
FOR i := 1 UPTO nLenArray
    <Statements>...
NEXT
```

To traverse an array in descending order:

```
nLenArray := ALen(aArray)
FOR i := nLenArray DOWNTO 1
    <Statements>...
NEXT
```

See Also

AEval(), <u>BEGIN SEQUENCE</u>, <u>DO CASE</u>, <u>DO WHILE</u>, <u>IF</u>, <u>EXIT</u>, <u>LOOP</u>

1.8.4.15.14 FOREACH Statement

Purpose

Execute a block of statements for all elements in a collection

Syntax

```
FOREACH  [IMPLIED <idElement> | VAR <idElement> | <idElement> AS
<idType>] IN <container>
    <Statements>...
    [EXIT]
    <Statements>...
    [LOOP]
NEXT
```

Note

In the FoxPro dialect FOR EACH as 2 separate words is also allowed.
In the FoxPro and Xbase++ dialect ENDFOR is allowed as alternative for NEXT

Arguments

<idElement>	The name of the variable that will receive the values of the elements in <container> When the IMPLIED or VAR clause is used then the datatype of the variable is inferred from the container. When the AS <idType> clause is used then this will be the datatype of the variable
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
<container>	A DotNet object that supports IEnumerable(), such as a XBase array, .Net array or a collection like List<>
EXIT	Unconditionally branches control from within a FOR, FOREACH , REPEAT or DO WHILE statement to the statement immediately following the corresponding ENDDO or NEXT statement.

LOOP	Branches control to the most recently executed FOR, FOREACH , REPEAT or DO WHILE statement.

Description

The FOREACH statement is a convenient way to enumerate variable in an array or collection.
Preferrably the iteration variable should not be changed inside the loop. A compiler warning will be shown when you do this.
It is also not recommended to change the container in the loop as this may often result in a runtime error.

See Also

AEval(), BEGIN SEQUENCE, DO CASE, DO WHILE, IF, EXIT, LOOP

1.8.4.15.15 IF Statement

Purpose

Conditionally execute a block of statements.

Syntax

```
IF <lCondition> [THEN]
    <Statements>...
[ELSEIF <lCondition>]
    <Statements>...
[ELSE]
    <Statements>...
END[IF]
```

Arguments

[THEN]	The THEN keyword is optional and has been added because Visual FoxPro allows this keyword.
<lCondition>	If this expression evaluates to TRUE, the statements following it up until the next ELSEIF, ELSE, or ENDIF are executed. Afterwards, control branches to the statement following the next ENDIF statement.
ELSE	If all preceding IF and ELSEIF conditions evaluate to FALSE, the statements following the ELSE up until the next ENDIF are executed. Afterwards, control branches to the statement following the next ENDIF statement.

Description

IF works by branching to the statement following the first *<lCondition>* that evaluates to TRUE. If all conditions evaluate to FALSE, it branches to the statement following the ELSE statement (if specified). Execution proceeds until the next ELSEIF, ELSE, or ENDIF is encountered, and control then branches to the first statement following the next ENDIF statement.

Control structures can be nested to any depth. The only requirement is that each control structure be properly nested.

Note: IF...ELSEIF...ENDIF is identical to DO CASE...ENDCASE, with neither syntax having a performance advantage over the other. The IF construct is also similar to the IIf() operator which can be used within expressions.

Examples

This example evaluates a number of conditions using an IF...ELSEIF...ENDIF construct:

```
LOCAL nNumber := 0

IF nNumber < 50
   ? "Less than 50"
ELSEIF nNumber = 50
   ? "Is equal to 50"
ELSE
   ? "Greater than 50"
ENDIF
```

See Also

BEGIN SEQUENCE, DO CASE, DO WHILE, FOR, IIf()

1.8.4.15.16 LOOP Statement

Purpose

The LOOP statement causes control to jump back to the beginning of the innermost FOR, FOREACH , REPEAT or DO WHILE statement.

Syntax

```
LOOP
```

Remarks

The LOOP statement is only valid within a DO WHILE, FOR ... NEXT, FOREACH .. NEXT or REPEAT .. UNTIL construct. A LOOP statement outside of either construct will raise a compile time error.

See Also

DO WHILE
FOR

FOREACH
REPEAT UNTIL

1.8.4.15.17 NOP Statement

Purpose

The NOP statement inserts an empty statement in the code

Syntax

```
NOP
```

Remarks

The NOP statement is an empty statement. It can be used to indicate that a certain code section is left intentionally empty

Examples

```
FUNCTION Start(aArgs as string[]) AS VOID
   IF aArgs != NULL .AND. aArgs:Length > 0
       ? "You passed ", aArgs:Length, "command line parameters"
   ELSE
       NOP    // This indicates that the else branch is not
```

```
forgotten but intentionally empty
    ENDIF
    .....
    RETURN
```

1.8.4.15.18 QUIT Command

Purpose

Terminate application processing, close all open files, and return control to the operating
system.

Syntax

QUIT | CANCEL

Description

QUIT is the same as CANCEL. See CANCEL for a complete explanation of this
command.

Assembly

XSharp.RT.DLL

See Also

BEGIN SEQUENCE, CanBreak(), CANCEL, ErrorLevel(), RETURN

1.8.4.15.19 REPEAT UNTIL Statement

Purpose

Execute a loop until a condition is TRUE.

Syntax

```
REPEAT
    <Statements>...
```

```
[EXIT]
    <Statements>...
[LOOP]
    <Statements>...
UNTIL <lCondition>
```

Arguments

<lCondition>   The logical control expression for the REPEAT UNTIL loop.

EXIT   Unconditionally branches control from within a FOR, FOREACH , REPEAT or DO WHILE statement to the statement immediately following the corresponding ENDDO or NEXT statement.

LOOP   Branches control to the most recently executed FOR, FOREACH , REPEAT or DO WHILE statement.

Description

As long as the condition evaluates to FALSE then the loop will continuet. When the condition evaluates to false, the REPEAT construct terminates and control passes to the statement immediately following the UNTIL.

Use EXIT to terminate a REPEAT UNTIL structure based on a condition other than the REPEAT UNTIL condition. LOOP, by contrast, prevents execution of statements within a REPEAT UNTIL based on an intermediate condition, and returns to the most recent REPEAT UNTIL statement.

Control structures can be nested to any depth. The only requirement is that each control structure be properly nested.

Examples

This example demonstrates a typical control structure for a simple grouped report:

```
LOCAL cOldSalesman, nTotalAmount
USE sales INDEX salesman NEW
DO WHILE .NOT. EOF()
   cOldSalesman := Sales->Salesman
   nTotalAmount := 0
   DO WHILE cOldSalesman = Sales->Salesman ;
            .AND. (.NOT. EOF())
      ? Sales->Salesman, Sales->Amount
      nTotalAmount := nTotalAmount + Sales->Amount
```

```
        SKIP
    ENDDO
    ? "Total:  ", nTotalAmount, "for", cOldSalesman
ENDDO
CLOSE sales
```

This code fragment demonstrates how LOOP can be used to provide an intermediate processing condition:

```
DO WHILE <lCondition>
    <Initial Processing>...
    IF <Intermediate Condition>
        LOOP
    ENDIF
    <Continued Processing>...
ENDDO
```

The next example demonstrates the use of DO WHILE to emulate a "repeat until looping" construct:

```
LOCAL lMore := TRUE
DO WHILE lMore
    <Statements>...
    lMore := <lCondition>
ENDDO
```

Here, a DO WHILE loop moves sequentially through a database file:

```
DO WHILE .NOT.  EOF()
    <Statements>...
    SKIP
ENDDO
```

See Also

BEGIN SEQUENCE, DBEval(), DO CASE, FOR, IF, RETURN, DO WHILE, EXIT, LOOP

1.8.4.15.20 RETURN Statement

Purpose

Terminate a routine by returning control to the calling routine or operating system if executed from the Start() routine.

Syntax

RETURN [<*uValue*>]

Arguments

<uValue>   May be specified in a function or method definition to designate its return value. Procedure definitions do not allow <uValue> as part of the RETURN statement. See the FUNCTION and METHOD entries in this guide for information about default return values if <uValue> is not specified.

Description

All private variables created and local variables declared in the current routine are released from memory when control returns.

Examples

These examples illustrate the general form of the RETURN statement in a procedure and in a function:

```
PROCEDURE <idProcedure>()
    <Statements>...
    RETURN

FUNCTION <idFunction>()
    <Statements>...
    RETURN <uValue>
```

The next example returns an array, created in a function, to a calling routine:

```
FUNCTION PassArrayBack()
    PRIVATE aArray[10][10]
    aArray[1][1] = "myString"
    RETURN aArray
```

See Also

[BEGIN SEQUENCE](#), [FUNCTION](#), [LOCAL](#), [PRIVATE](#), [PROCEDURE](#), [PUBLIC](#), [QUIT](#)

1.8.4.15.21 RUN Command

Note	This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header ([-nostddefs](#)) files then this command will not be available. If you tell the compiler to use a different standard header file ([-stddef](#)) then this command may also be not available

Purpose

Execute a Windows or DOS application, a batch file, or a DOS command.

Syntax

RUN <*xcCommandLine*>

Arguments

<xcCommandLine>	The command line is made up of two parts. The first part is one of the following:
•	An executable Windows or DOS program
•	A .PIF file
•	A .BAT file
•	Any resident DOS command
•	COMMAND.COM
	The second part is the command line parameters that will be passed to the specified application.
	NOTE: When running DOS programs, it is best to specify the file extension (for example, .EXE) rather than allow DOS to determine the default; otherwise, the RUN command will generate a temporary file named VODOSRUN.BAT to run the specified program.

Description

RUN executes a Windows or DOS program or a DOS command from within an application.
The application name in <*xcCommandLine*> may optionally contain a path. If it does not, Windows will search for the application in the following order:

- The current directory
- The Windows directory
- The Windows SYSTEM subdirectory
- The directory that contains the currently executing module (.EXE or .DLL)
- The directories in the PATH environment variable

X#

- All network drives

If you use RUN to start a Windows application, the two applications will be run asynchronously. In other words, your XSharpapplication will not wait for the called application to finish but will continue to execute the instructions following the RUN command.

If you specify a DOS application, a .BAT file, or a DOS command, Windows will create a DOS task and switch to it, thereby stopping your application. During the execution of the DOS task, however, the user may switch back to the calling application.. The calling application will automatically resume execution after the DOS task terminates.
If you run a DOS program, you might consider setting up a .PIF file (using the PIF Editor provided by Windows) to fine tune the execution environment. For example you might specify that the DOS application is to be executed in a window rather than in full-screen mode. If you are using a .PIF file, pass the name of the .PIF file to the RUN command instead of the name of your application or give it the same base name as your application and put it into the same directory. In the latter case, Windows will pick it up automatically if you pass the application name to the RUN command.

Examples

This example starts the Windows Notepad editor on a file called DATA.TXT:

```
RUN notepad data.txt
```

One of the options you can give your users is direct access to DOS. Do this with:

```
RUN command.com
```

Assembly

XSharp.RT.DLL

1.8.4.15.22 SWITCH Statement

Purpose

Conditionally execute a block of statements.

Syntax

```
SWITCH <expression>
CASE <constantvalue> [WHEN <whenexpression>]
   <Statements>...
[CASE <constantvalue>]
[CASE <constantvalue>]
   <Statements>...
[CASE <constantvalue>]
   <Statements>...
[CASE <variablename> AS <datatype>] [WHEN <whenexpression>]
   <Statements>...
[OTHERWISE]
   <Statements>...
END [SWITCH]
```

Arguments

<constantvalue>	A constant value that can be evaluated at compile time. You can also have 2 consecutive CASE lines without statements between them. In that case both CASE lines share the same statementblock. If you want a case block without statements, then insert a NOP statement as its statement.
<whenexpression>	A logical expression that determines if the particular CASE block should be active This is sometimes called an expression filter.
<variablename>	A variablename that gets declared by the pattern matching expression
<datatype>	The datatype of the variable in the pattern matching expression
OTHERWISE	If none of the preceding CASE conditions match, the statements following the OTHERWISE up until the next END SWITCH are executed. Afterwards, control branches to the statement following the next END SWITCH statement.

Description

SWITCH works by branching to the statement following the first CASE <constantvalue> that evaluates to TRUE. If all CASE conditions evaluate to FALSE, it branches to the statement following the OTHERWISE statement (if specified).
In general there are 2 types of SWITCH statements:
1. Switch statements with constant values (CASE <constantvalue>)
2. Switch statements with pattern matching expressions (CASE <varName> AS <datatype>)
Both types of SWITCH statements can be enhanced with WHEN filters.

If you do not have a WHEN filter then each CASE line must be "unique", so no two CASEs can have the same constantvalue or same datatype. If you add a WHEN filter then this is allowed.

The compiler may (and will) rearrange the order of the CASE labels when generating code for example to combine two CASEs with the same constant value and a WHEN filter. These may be combined into one CASE label with an embedded IF statement.

Execution proceeds until the next CASE, OTHERWISE, or END SWITCH is encountered, and control then branches to the first statement following the next ENDCASE statement. Control structures can be nested to any depth. The only requirement is that each control structure be properly nested.

Examples

This example uses SWITCH in a menu structure to branch control based on user selection:

```
FUNCTION ActonChoice(nChoice as LONG) AS VOID
SWITCH nChoice
CASE  0
    RETURN
CASE 1
    ChoiceOne()
CASE 2
    ChoiceTwo()
END SWITCH
```

See Also

BEGIN SEQUENCE, DO WHILE, FOR, FOREACH IF DO CASE

1.8.4.15.23 THROW Statement

The THROW statement raises a runtime exception.

Syntax

```
THROW [expression]
```

Arguments

expression   An optional expression to throw.

Remarks

THROW throws a runtime exception, causing execution to branch to the nearest **CATCH** or **FINALLY** block in a **TRY** construct. If execution is not within a TRY construct the application will terminate.

The specified expression is passed to the **CATCH** statement, if any, and must be of type System.Exception or a class derived from it . See **TRY**-**CATCH**-**FINALLY** fro more information.

Using **THROW** within a **CATCH** block without any arguments re-throws the exception, passing it unchanged to the next highest **TRY**-**CATCH** block.

Example

```
USING System.IO
FUNCTION ReadFile( filename AS STRING ) AS STRING
   LOCAL s AS STRING
   TRY
      s := File.ReadAllText( filename )
   CATCH e AS DirectoryNotFoundException
      ? "Directory not found", e
   CATCH e AS IOException
      ? "IO exception occurred", e
   CATCH e AS UnauthorizedAccessException
      ? "Access denied", e
   CATCH
      ? "Some other exception"
      // Escalate error to next TRY-CATCH
      THROW
   FINALLY
      ? "All done!"
   END TRY
   RETURN s
```

See Also

[TRY-CATCH-FINALLY](#)

1.8.4.15.24 TRY CATCH Statement

Purpose

TRY, **CATCH** and **FINALLY** are used to declare an exception handling block.

```
TRY
    guardedStatements
[CATCH [[variableName] AS exceptionType] [WHEN whenexpression]
    exceptionHandlingStatements
]
[FINALLY
    cleanupStatements
]
END TRY
```

Arguments

variableName	The name of a variable that will receive the exception. The variable name is optional. If you only specify the type then the exception will still be caught but not stored in a local variable.
exceptionType	The exception type that will be caught by the CATCH block.
whenexpression	A logical expression that determines if the particular CATCH block should be active
exceptionHandlingStatements	Zero or more statements that handle the exception condition.
cleanupStatements	Zero or more statements that perform any necessary cleanup before the TRY block is exited..

Remarks

A **TRY**-**CATCH**-**FINALLY** block is used to trap and handle exceptions that may be thrown within a block of code. Exceptions may be generated by the CLR, the Vulcan.NET runtime library, third-party libraries or by application code using the **THROW** statement.

TRY Block

The statements within the **TRY** block are sometimes referred to as "guarded" statements. These are the statements that potentially can cause exceptions that you want to handle.

CATCH Blocks

A exception handling block may contain any number of **CATCH** blocks (including zero). Each **CATCH** block that declares a variable name and a type will receive exceptions of that type. CATCH blocks that receive exceptions implicitly declare a local variable that will contain the caught exception. This implicitly declared local variable is only valid within the scope of the CATCH block. The name given to the variable must not be the same as any explicitly declared LOCAL or parameter, or a compile-time error will occur. However, it is legal to use the same variable name in multiple CATCH blocks. Since each **CATCH** block's variable is only visible within its enclosing block, there is no conflict.

A **CATCH** block may also be declared without a variable name but WITH an exception type. In that case the exception will still be caught but not stored in a local variable.

A **CATCH** block may also be declared without any variable name and exception type. This type of CATCH block will catch any exception, and is equivalent to declaring a CATCH block with an exception type of System.Exception.

If multiple **CATCH** blocks are declared, the order in which they appear is important. The CLR will examine the CATCH clauses in order, and invoke the first one that matches the exception being thrown. This includes not only the specific exception class that was specified, but any derived classes. For this reason, you should catch the more specific exception types before less specific ones.

The exception type declared in a **CATCH** block must always be System.Exception, or a class derived from it.

If no suitable **CATCH** block was declared for the exception that has been thrown, control will be passed to the next highest exception handling block. If there is no higher exception handling block, or none that can handle the exception, the application will terminate.

Exceptions may be explicitly passed on to the next highest exception handling block by using the THROW keyword.

Finally Block

If a **FINALLY** block is declared, any statements within it are executed regardless of how the TRY block exits. This provides a mechanism to perform any cleanup such as releasing resources. The code within a finally block will be executed even if there is no suitable CATCH block to handle the exception.

Compatibility Note:

TRY-**CATCH**-**FINALLY** blocks are similar to, but much more flexible than **BEGIN SEQUENCE**-**RECOVER** blocks. However, **BEGIN SEQUENCE** and **RECOVER** are still supported for backwards compatibility.

Note that exceptions thrown with **BREAK** will not be caught with a **TRY**-**CATCH**-**FINALLY** block because the data thrown by **BREAK** is encapsulated in a USUAL, which does not inherit from System.Exception.

However, exceptions thrown with **THROW** will be caught by the next highest BEGIN SEQUENCE block (if any) and the exception will be encapsulated in a USUAL if a RECOVER USING variable has been declared.

Also note that the CanBreak() runtime function does not detect whether execution is currently within a **TRY** block. CanBreak() is provided only for compatibility with existing Visual Objects code and **BEGIN SEQUENCE** blocks, and should not be relied upon to determine whether execution is within an exception handling block. There is no way to determine whether execution is within an exception handling block because this functionality is not present in the CLR, and execution may currently be within code that is compiled in an language other than Vulcan.NET.

Example

The following example tests for division by zero and catches the exception that will be thrown by the CLR if the divisor is zero. Any other exceptions would propagate to the next highest exception handling block (if any). Without the exception handling block, the application would terminate with an unhandled DivideByZeroException.

```
FUNCTION DivisionTest( x AS INT, y AS INT ) AS INT
   TRY
      RETURN x / y
   CATCH e AS System.DivideByZeroException
      ? "Divide by zero!", e
      RETURN 0
   END TRY
```

The following example demonstrates multiple CATCH blocks and a FINALLY block:

```
USING System.IO

FUNCTION ReadFile( filename AS STRING ) AS STRING
   LOCAL s AS STRING

   TRY
      s := File.ReadAllText( filename )
   CATCH e AS DirectoryNotFoundException
      ? "Directory not found", e
   CATCH e AS IOException
      ? "IO exception occurred", e
   CATCH e AS UnauthorizedAccessException
      ? "Access denied", e
   CATCH
      ? "Some other exception"
   FINALLY
      ? "All done!"
   END TRY

   RETURN s
```

See Also

BEGIN SEQUENCE
THROW

1.8.4.15.25 WITH command

Purpose

Specifies multiple properties for an object.

Syntax

```
WITH ObjectName
    [cStatements] or
    [:Statements]
END WITH
```

Arguments

ObjectName   Specifies the name of the object. ObjectName can be the name of the object or a reference to the object.

1.8.4.15.26 YIELD Statement

Purpose

When you use the yield keyword in a statement, you indicate that the method, operator, or get accessor in which it appears is an iterator.

Syntax

```
YIELD RETURN <expression>
YIELD (EXIT|BREAK)
```

Arguments

YIELD RETURN   You use YIELD RETURN <expression> to return each element one at a time.

YIELD [BREAK|EXIT]   You use YIELD BREAK or YIELD EXIT to end the iteration.

Examples

```
using System.Collections.Generic

// The Yield return statement allows you to create code that
returns a
// collection of values without having to create the collection in
memory first.
// The compiler will create code that "remembers" where you were
inside the
// loop and returns to that spot.
FUNCTION Start AS VOID
   FOREACH nYear AS INT IN GetAllLeapYears(1896, 2040)
      ? "Year", nYear, "is a leap year."
   NEXT
   Console.ReadLine()
RETURN

FUNCTION GetAllLeapYears(nMin AS INT, nMax AS INT) AS
IEnumerable<INT>
   FOR LOCAL nYear := nMin AS INT UPTO nMax
      IF nYear % 4 == 0 .and. (nYear % 100 != 0 .or. nYear % 400
== 0)
         YIELD RETURN nYear
      END IF
      IF nYear == 2012
         YIELD EXIT      // Exit the loop
      ENDIF
   NEXT
```

1.8.4.16 Terminal Window

?|??
ACCEPT
SET COLOR
WAIT

Note: XSharp does not support the following commands:

@...BOX
@...CLEAR
@...SAY
@...TO
@..GET
DISPLAY
INPUT
LIST
READ

SET CONSOLE
SET DEVICE
SET PRINTER
TEXT ... ENDTEXT
TYPE

1.8.4.16.1 ?|?? Statement

Purpose

Display the results of one or more expressions, separated by a space, in the terminal window.

Syntax

? | ?? [<*uValueList*>]

Arguments

<uValueList>	A list of values to display. If no argument is specified with the ? statement, a carriage return/linefeed is sent to the terminal window. If you use the ?? statement without arguments, nothing happens.

Description

? and ?? are synonyms for the QOut() and QQOut() functions, respectively.
Although functionally similar to one another, ? and ?? differ slightly. ? sends a carriage return/linefeed before displaying the results of the expression list. ?? displays output at the current position. This lets you use ?? statements to display successive output on the same line.

A ? or ?? statement locates the cursor or print head one position to the right of the last character displayed. Row() and Col() are updated to reflect the new cursor position.

If output from a ? or ?? statement reaches the edge of the terminal window it wraps to the next line. If the output reaches the bottom of the window the window scrolls up one line. To format any expression specified, use Transform(). If you need to pad a variable length value for column alignment, use any of the Pad() functions to left-justify, right-justify, or center the value as illustrated in the examples below.

Examples

This example displays a record from a database file using ? and ?? statements with PadR() to assure column alignment:

```
LOCAL nPage := 0, nLine := 99
USE salesman INDEX salesman NEW
```

```
        DO WHILE !EOF()
           IF nLine > 55
                   IF nPage != 0
                           EJECT
                   ENDIF
                   ? PadR("Page", LTRIM(STR(++nPage)), 72)
                   ?? DTOC(TODAY())
                   ?
                   ?
                   ? PadC("Sales Listing", 79)
                   ?
                   nLine := 5
           ENDIF
           ? Name, Address, PadR(RTrim(City) + "," + State, 20), ZipCode
           ++nLine
           SKIP
        ENDDO
        CLOSE salesman
```

Assembly

XSharp.RT.DLL

See Also

QOut(), QQOut()

1.8.4.16.2 \|\\ Statement

Purpose

Prints or displays lines of text

Syntax

\	TextLine
\\	TextLine

Arguments

\ TextLine	When you use \, the text line is preceded by a carriage return and a line feed.
\\ TextLine	When you use \\, the text line is not preceded by a carriage return and a line feed.
	Any spaces preceding \ and \\ are not included in the output line, but spaces following \ and \\ are included.

You can embed an expression in the text line. If the expression is enclosed in the text merge delimiters (<< >> by default) and SET TEXTMERGE is ON, the expression is evaluated and its value is output as text.

Description

The \ and \\ commands facilitate text merge in X#.
Text merge makes it possible for you to output text to a file to create form letters or programs.
Use \ and \\ to output a text line to the current text-merge output file and the screen.
SET TEXTMERGE is used to specify the text merge output file. If text merge is not directed to a file, the text line is output only to the main Visual FoxPro window or the active user-defined output window. SET TEXTMERGE NOSHOW suppresses output to the main Visual FoxPro window or the active user-defined window.

Examples

This example displays a record from a database file using ? and ?? statements with PadR() to assure column alignment:

```
CLOSE DATABASES
OPEN DATABASE (C:\Test\Data\testdata')
USE Customer // Open customer table
SET TEXTMERGE ON
SET TEXTMERGE TO letter.txt
\<<CDOW(DATE( ))>>, <<CMONTH(DATE( ))>>
\\ <<DAY(DATE( ))>>, <<YEAR(DATE( ))>>
\
\
\Dear <<contact>>
\Additional text
\
\Thank you,
\
\XYZ Corporation
CLOSE ALL
```

Assembly

XSharp.VFP.DLL

See Also

[SET TEXTMERGE](#)

1.8.4.16.3 ACCEPT Command

Purpose

Take input from the keyboard and assign it to a memory variable.

Syntax

ACCEPT [<*uValuePrompt*>] TO <*idVar*>

Arguments

<uValuePrompt>	An optional prompt displayed before the input area.
TO <idVar>	The variable that will hold input from the keyboard. If there is no variable named <idVar> that is visible to the current routine, a private variable is created.

Description

When ACCEPT is executed, it first performs a carriage return/linefeed, displays the prompt in the terminal window, and begins taking characters from the keyboard at the position immediately following the prompt.

Up to 255 characters can be entered. When input reaches the edge of the window, as defined by MaxCol(), the cursor moves to the next line.

ACCEPT supports only two editing keys: Backspace and Enter (Esc is not supported). Backspace deletes the last character typed. Enter confirms entry and is the only key that can terminate an ACCEPT. If Enter is the only key pressed, ACCEPT assigns a NULL_STRING to <*idVar*>.

Examples

This example uses ACCEPT to get keyboard input from the user:

```
LOCAL cVar
ACCEPT "Enter a value:  " TO cVar
IF cVar = NULL_STRING
```

```
    ? "User pressed Enter"
ELSE
    ? "User input:", cVar
ENDIF
```

Assembly

XSharp.RT.DLL

1.8.4.16.4 CLEAR SCREEN Command

Purpose

Take input from the keyboard and assign it to a memory variable.

Syntax

CLEAR SCREEN

Description

The command CLEAR erases the screen.
After execution of the CLEAR command, the cursor is positioned at 0, 0 and the functions Row() and Col() are updated.

Examples

This example uses ACCEPT to get keyboard input from the user:

```
FUNCTION Start AS VOID
SET COLOR TO gr+/B // Yellow on Blue
CLEAR SCREEN
? "Hello world"
```

Assembly

XSharp.RT.DLL

1.8.4.16.5 SET ALTERNATE Command

Purpose

Write screen output to a text file.

Syntax

SET ALTERNATE TO [<cFileName> [ADDITIVE]]
SET ALTERNATE ON | OFF | <lToggle>

Arguments

<cFileName>	Specifies the name of the ASCII file where screen output is recorded. The name must contain the drive and path. The file name can be specified either as a literal file name or as a character expression in parentheses. When the file name is specified without a file extension, the extension ".TXT" is used by default.
ADDITIVE	The option ADDITIVE adds the screen output to the current contents of the file <cFileName> . Without this option, the contents of the file are deleted if the file already exists. If a file with the name <cFileName> does not exist, it is created.
<lToggle>	<lToggle> is a logical expression which must appear in parentheses. Instead of the logical expression, the option ON can be specified for the value .T. (true) or OFF for the value .F. (false). When .T. or ON is specified, screen output is recorded in the file <cFileName> .

Description

The command SET ALTERNATE opens an ASCII file to record screen output. Only screen output performed using commands like ? or LIST and functions like QOut() and QQOut() is written into the file. Commands containing the option TO FILE work like SET ALTERNATE. .

The alternate file (the ASCII file <cFileName>) is not tied to a single work area, but can be used from all work areas. Output to the file is turned on or off by the options ON | OFF or the logical value of <lToggle> . An alternate file is defined by including a file name in the command SET ALTERNATE. When SET ALTERNATE TO is called without a specified file name, the currently open alternate file is closed and recording screen output to a file is no longer possible. The alternate file is also closed by the commands CLOSE ALTERNATE and CLOSE ALL.

Examples

```
FUNCTION Start
     USE Customers NEW
     SET ALTERNATE TO CustomerList.TXT
     SET CONSOLE OFF
     DO WHILE .NOT. Eof()
        ? Customers->LastName, Customers->FirstName
        ? Customers->Street
        ? Customers->City + ",", Customers->State, Customers-
>Zip
        ?
        SKIP
     ENDDO
     SET ALTERNATE TO
     SET CONSOLE ON
     USE

     RETURN
```

Assembly

XSharp.RT.DLL

See Also

SetAlternate(),SetAltFile(), SET CONSOLE

1.8.4.16.6 SET COLOR Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a
> function call. If you disable the standard header (-nostddefs) files then this command will not be
> available. If you tell the compiler to use a different standard header file (-stddef) then this
> command may also be not available

Purpose

Change the setting that determines the current color for the terminal window.

Syntax

SET COLOR | COLOUR TO [*<xcColorString>*]

Description

SET COLOR is functionally equivalent to SetColor(). SET COLOR TO with no arguments is the same as SetColor(NULL_STRING), returning to the default color settings.

Examples

This example uses the unselected setting to make the current GET red on white while the rest are black on white:

```
FUNCTION Start AS VOID
SET COLOR TO gr+/B  // Yellow on Blue
CLEAR SCREEN
? "Hello world"
```

Assembly

XSharp.RT.DLL

See Also

SetColor()

1.8.4.16.7 SET CONSOLE Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Turn the screen display on/of

Syntax

SET CONSOLE ON | off | <lToggle>

Arguments

<lToggle>   A logical expression which must appear in parentheses. Instead of the logical expression, the option ON can be specified for the value .T. (true) or OFF for the value .F. (false). When .T. or ON is specified (the default value), all output is displayed on the screen. If the setting is set to

X#

OFF, output on the screen using commands like ? or ?? is suppressed.

Description

The command SET CONSOLE deactivates or activates output of characters on the screen. This includes commands like ? and functions like QOut() or QQOut(), which always begin output at the current cursor position.

When SET CONSOLE is set to ON, output is displayed on the screen. There can also be parallel output to a file (command SET ALTERNATE).

Examples

```
FUNCTION Start
     USE Customers NEW
     SET ALTERNATE TO CustomerList.TXT
     SET CONSOLE OFF
     DO WHILE .NOT. Eof()
        ? Customers->LastName, Customers->FirstName
        ? Customers->Street
        ? Customers->City + ",", Customers->State, Customers-
>Zip
        ?
        SKIP
     ENDDO
     SET ALTERNATE TO
     SET CONSOLE ON
     USE

     RETURN
```

Assembly

XSharp.RT.DLL

See Also

SetConsole(), [SET ALTERNATE](#)

1.8.4.16.8 SET TEXTMERGE Command

> **Note**   This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header ([-nostddefs](#)) files then this command will not be available. If you tell the compiler to use a different standard header file ([-stddef](#)) then this command may also be not available

X#

Purpose

Enables or disables the evaluation of fields, variables, array elements, functions, or expressions that are surrounded by text-merge delimiters, and lets you specify text-merge output.

Syntax

```
SET TEXTMERGE [ON | OFF] [TO [FileName] MEMVAR VarName [ADDITIVE]]
    [WINDOW WindowName] [SHOW | NOSHOW]
```

Arguments

ON	Specifies that any fields, variables, array elements, functions, or expressions surrounded by the text-merge delimiters be evaluated and output when placed after \ or \\, or between TEXT and ENDTEXT.
OFF	(Default) Specifies that any fields, variables, array elements, functions, or expressions be literally output along with the text-merge delimiters surrounding them.
TO [FileName]	Specifies that output from \, \\, and TEXT ... ENDTEXT be directed to a text file in addition to the main Visual FoxPro window, which is the default. You can also direct the output to a text file by including FileName. If a file with that name doesn't exist, a new file is created. If a file with the same name already exists and SET SAFETY is set to ON, you are given the option of overwriting the existing file. The text file is opened as a low-level file, and its file handle is stored to the _TEXT system variable. You can close the file by issuing SET TEXTMERGE TO without additional arguments. If the file handle of another file was previously stored in _TEXT, that file is closed.
MEMVAR VarName	Specifies a variable to contain data from TEXTMERGE output. Because SET TEXTMERGE is a global setting and can span several procedures or methods, it is possible for MEMVAR VarName to lose scope. The command will function even when the variable is out of scope, but will return no content. You can control scoping of MEMVAR VarName by declaring the variable.
ADDITIVE	Specifies that output from \, \\, and TEXT ... ENDTEXT be appended to an existing file or memory variable.
SHOW | NOSHOW	(Default) SHOW displays text-merge output. NOSHOW suppresses display of text-merge output. By default, output generated by \, \\, and TEXT ... ENDTEXT is sent to the Console window.

Description

Specifies that any fields, variables, array elements, functions, or expressions surrounded by the text-merge delimiters be evaluated and output when placed after \ or \\, or between TEXT and ENDTEXT.
The following short program example demonstrates how the contents of the variable gcTodayDate and the DATE( ) and TIME( ) functions are evaluated when SET TEXTMERGE is set to ON.

Examples

```
CLEAR
CLOSE DATABASES
SET TEXTMERGE ON
SET TEXTMERGE TO ContactList.TXT
CLOSE DATABASES
OPEN DATABASE ( 'C:\test\Data\testdata')
USE customer
TEXT
        CONTACT NAMES
   <<DATE( )>>    <<TIME( )>>
ENDTEXT
WAIT "Press a key to generate the first ten names."
SCAN NEXT 10
   TEXT
      <<contact>>
   ENDTEXT
ENDSCAN
CLOSE ALL
```

Assembly

XSharp.VFP.DLL

1.8.4.16.9 TEXT Command

Purpose

Declare a block of text that can be assigned to a variable or send to an output device.
There are several different variations of the TEXT Statement:
- Core TEXT Command
- Non-Code TEXT Command

- [FoxPro TEXT Command](#)

The different variations of the TEXT commands are implemented as User Defined commands in XSharpDefs.xh or one of the other header files.
All of these commands map to the preprocessor directives [#text](#) and [#endtext](#) where the actual work is done.

1.8.4.16.9.1 TEXT Command (Core)

Syntax

```
TEXT TO <VariableName> [ADDITIVE]
      TextLines
ENDTEXT
```

Arguments

TextLines	Specifies text to assign to the variable VarName
TO <VariableName>	Specifies the variable name to use for passing the contents of the TEXT...ENDTEXT. This variable should be of type string, or should allow that a string can be added to it. It does not have to be a local variable, it can also be a field in the current class or an expression (someVar:SomeField)
ADDITIVE	Determines whether the contents of the TO variable are overwritten or added to existing contents.

Example

```
FUNCTION Start() AS VOID
LOCAL cValue AS STRING
TEXT TO cValue
Line 1
Line 2
Line 3
ENDTEXT
? cValue
TEXT TO cValue ADDITIVE
```

```
      Line 4
      Line 5
   ENDTEXT
   ? cValue
   RETURN
```

The first time the variable will contain 3 lines of text delimited with CR/LF. The second time there will be 5 lines.

```
The code produced by the compiler will somewhat look like this:
```

```
   FUNCTION Start() AS VOID
   LOCAL cValue AS STRING
   cValue := ""
   cValue += "Line 1" +  chr(13)+chr(10)
   cValue += "Line 2" +  chr(13)+chr(10)
   cValue += "Line 3" +  chr(13)+chr(10)

   ? cValue

   cValue += "Line 4" +  chr(13)+chr(10)
   cValue += "Line 5" +  chr(13)+chr(10)

   ? cValue
   RETURN
```

See also

[#text directive](#)
[#endtext directive](#)

1.8.4.16.9.2 TEXT Command (Non-Core)

Syntax

```
   TEXT [INTO <VariableName> [TRIMMED]] |
   [INTO <VariableName> WRAP] |
   [INTO <VariableName> WRAP <cLineBreak> [TRIMMED]]
```

```
        TextLines
    ENDTEXT
```

or

```
    TEXT [TO PRINTER] | [TO FILE <cFilename>]
        TextLines
    ENDTEXT
```

Arguments

TextLines	Specifies text to assign to the variable VarName
INTO <VariableName>	Specifies the variable name to use for passing the contents of the TEXT...ENDTEXT. This variable should be of type string, or should allow that a string can be added to it. It does not have to be a local variable, it can also be a field in the current class or an expression (someVar:SomeField)
WRAP <cLineBreak>	The optional parameter <cLineBreak> specifies a character string that is used to wrap lines. The default value is CHR(13)+CHR(10) (carriage return and line feed). If WRAP is not specified, no line breaks will be inserted.
TRIMMED	This optional parameter tells the preprocessor to remove all spaces from the beginning of the line if TEXT INTO .. is used.
TO FILE <cFilename>	<cFilename> indicates the name of a file where the text <Text> is optionally written. The name must contain the drive and path, if necessary. The file name can be specified either as a literal file name or as a character expression in parentheses. When the file name is specified without file extension, ".TXT" is used by default.
TO PRINTER	Indicates that the text should be sent to the printer instead to a file.

See also

#text directive

[#endtext directive](#)

1.8.4.16.9.3 TEXT Command (FoxPro)

Syntax

```
TEXT [TO <VariableName> [ADDITIVE] [TEXTMERGE] [NOSHOW] [FLAGS
nValue] [PRETEXT eExpression]]
     TextLines
ENDTEXT
```

Arguments

TextLines	Specifies text to send to the current output device. TextLines can consist of text, memory variables, array elements, expressions, functions, or any combination of these. Note X# evaluates expressions, functions, memory variables, and array elements specified with TextLines only if you set SET TEXTMERGE to ON and enclose them with the delimiters specified by SET TEXTMERGE DELIMITERS. If SET TEXTMERGE is OFF, Visual FoxPro outputs expressions, functions, memory variables, and array elements as string literals along with their delimiters. For example, X# evaluates and outputs the current date when you specify the DATE( ) function as TextLines only if SET TEXTMERGE is ON, and TextLines contains the function and the appropriate delimiters, such as <<DATE( ) >>. If SET TEXTMERGE is OFF, X#outputs <<DATE( )>> as a string literal. If you place comments within TEXT...ENDTEXT or after the single backslash character (\) or double backslash characters (\\), X# outputs the comments.
TO <VariableName>	Specifies the memory variable name to use for passing the contents of the TEXT...ENDTEXT. This variable can already exist. If the variable has not yet been declared, X# automatically creates it as a private variable. The TO clause operates regardless of how SET TEXTMERGE is set. If SET TEXTMERGE is set to a file, and the TO statement is included, Visual FoxPro outputs both the file and variable.
ADDITIVE	Determines whether the contents of the TO variable are overwritten or added to existing contents.

If the contents of TO VarName is not a string, X# always overwrites the contents in VarName.

TEXTMERGE	Enables evaluation of contents surrounded by delimiters without setting SET TEXTMERGE to ON.
NOSHOW	Disables display of the text merge to the screen.
FLAGS nValue	Specifies a numerical value that determines if output is suppressed to an output file, or if blank lines preceding any text are included in the output.

Value (additive)	
1	Suppresses output to the file specified with the _TEXT System Variable.
2	When the NOSHOW clause is included, preserves blank lines preceding text that appears within TEXT...ENDTEXT. Setting nValue to 2 will separate current TEXT...ENDTEXT output from previous TEXT...ENDTEXT output with a line feed. Combining an nValue setting of 2 and PRETEXT of 4 will separate current TEXT…ENDTEXT output from previous TEXT…ENDTEXT output with a line feed while removing empty lines in the TEXT...ENDTEXT output.

PRETEXT eExpression	Specifies a character string to insert before each line of the text merge contents between TEXT...ENDTEXT or a numeric expression. The following table describes behaviors of the PRETEXT clause depending on the expression specified by eExpression.

eExpression	**PRETEXT behavior**
Character expression	Insert the expression before each line of the text merge contents appearing between the TEXT...ENDTEXT statement. When using PRETEXT with TEXT...ENDTEXT, eExpression is limited to a maximum length of 255 characters. eExpression overrides the contents of the _PRETEXT system variable. When eExpression contains an expression that needs to be evaluated, for example, a user-defined

function (UDF), Visual FoxPro evaluates it only once when the TEXT command first appears.

Numeric expression

Specify additive flag values to determine behavior for the text merge contents appearing between the TEXT...ENDTEXT statement.

For example, a value of 7 specifies that Visual FoxPro eliminate all white space including spaces, tabs, and carriage returns. A value falling outside of the range of 0-15 produces an error.

Note
Specifying a value of zero does not eliminate white space.

When eExpression is a numeric expression, you can use the _PRETEXT system variable to insert additional text after Visual FoxPro eliminates white space.
The following table lists numeric additive flags that you can use in eExpression to specify additional behavior.

Value (Additive)	Description
1	Eliminate spaces before each line.
2	Eliminate tabs before each line
4	Eliminate carriage returns, for example, blank lines, before each line.
8	Eliminate line feeds.

Note
Unlike the _PRETEXT system variable, the PRETEXT clause does not have global scope and applies only to the TEXT...ENDTEXT statement in which it appears.

Characters removed using the PRETEXT clause apply only to text within the TEXT...ENDTEXT statement and not to evaluated text merged from cExpression. In the following example, the spaces in the memory variable, myvar, are not removed when merged with the text in TEXT...ENDTEXT:

```
myvar = "   AAA"
TEXT TO x NOSHOW ADDITIVE TEXTMERGE PRETEXT 7
Start Line
<<myvar>>
     BBB
   CCC
ENDTEXT
```

By default, TEXT ... ENDTEXT sends output to the terminal window. To suppress output to the terminal window, issue SET CONSOLE OFF. To send output to a printer or a text file, use SET PRINTER. To send output from TEXT ... ENDTEXT to a low-level file that you created or opened using FCREATE( ) or FOPEN( ), store the file handle returned by FCREATE( ) or FOPEN( ) to the _TEXT system variable, which you can use to direct output to the corresponding low-level file.

The text merge process usually includes any white space that might appear before each line in a TEXT...ENDTEXT statement. However, the inclusion of white space might cause the text merge to fail, for example, when XML is used in a Web browser. You must remove such white space to avoid incorrectly formatted XML.

Nesting TEXT...ENDTEXT statements is not recommended, especially if using the PRETEXT clause because the nested statements can affect the format of the outer statements.

Example 1

The following example demonstrates creating a low-level file called myNamesFile.txt and storing its file handle in the _TEXT system variable. The program exits if the myNamesFile.txt file cannot be created.

The code opens the customer table and outputs the names of the first ten contacts to CotactList.txt.
The code outputs the text and results of the functions to the text file.

```
CLEAR
CLOSE DATABASES
SET TEXTMERGE ON
SET TEXTMERGE TO ContactList.TXT
CLOSE DATABASES
OPEN DATABASE ( 'C:\test\Data\testdata')
USE customer
TEXT
         CONTACT NAMES
   <<DATE( )>>    <<TIME( )>>
ENDTEXT
WAIT "Press a key to generate the first ten names."
SCAN NEXT 10
   TEXT
      <<contact>>
   ENDTEXT
ENDSCAN
CLOSE ALL
```

Example 2

The following example shows a custom procedure that uses TEXT...ENDTEXT to store an XML DataSet to a variable. In the example, all spaces, tabs, and carriage returns are eliminated.

```
PROCEDURE myProcedure
   DO CASE
   CASE nValue = 1
      TEXT TO myVar NOSHOW TEXT PRETEXT 7
         <?xml version="1.0" encoding="utf-8"?>
         <DataSet xmlns="http://tempuri.org">
         <<ALLTRIM(STRCONV(leRetVal.item(0).xml,9))>>
         </DataSet>
      ENDTEXT
   OTHERWISE
   ENDCASE
ENDPROC
```

See Also

FOPEN( ) Function
_PRETEXT System Variable
SET TEXTMERGE Command
SET TEXTMERGE DELIMITERS Command
_TEXT System Variable
#text directive
#endtext directive

1.8.4.16.10 WAIT Command

> **Note** This command is defined in a header file and will be preprocessed by the X# preprocessor to a function call. If you disable the standard header (-nostddefs) files then this command will not be available. If you tell the compiler to use a different standard header file (-stddef) then this command may also be not available

Purpose

Display a prompt after sending a carriage return/linefeed to the terminal window, then wait for a key to be pressed.

Note: WAIT is a compatibility command and is no longer recommended.

Syntax

WAIT [<*uValuePrompt*>] [TO <*idVar*>]

Arguments

<uValuePrompt>   An optional prompt displayed before the input area. If omitted, "Press any key to continue..." is displayed. Specify NULL_STRING if you do not want to display a prompt.

TO <idVar>   The variable that will hold input from the keyboard. If there is no variable named <idVar> that is visible to the current routine, a private variable is created. <idVar> is assigned the keystroke as a string. If an Alt or Ctrl key combination is pressed, WAIT assigns Chr(0) to <idVar>.

Non-alphanumeric values entered by pressing an Alt+key combination assign the specified character. If the character can be displayed, it is echoed to the screen.

Example

This example illustrates how to store the WAIT keystroke as an array element:

```
FUNCTION Start AS VOID
LOCAL aVar[2]
WAIT "Press a key..." TO aVar[1]
? aVar[1]              // Result:  key pressed in
                       // Response to WAIT
? aVar[2]              // Result:  NIL
? ValType(aVar)        // Result:  A
? ValType(aVar[1])     // Result:  C
```

Assembly

XSharp.RT.DLL

See Also

ACCEPT

1.8.4.17 Variable Declaration

FIELD
LOCAL
MEMVAR
STATIC

1.8.4.17.1 FIELD Statement

Purpose

Declare one or more database field names to be used by the current routine.

Syntax

```
FIELD <idFieldList> [IN <idAlias>]
```

Arguments

<idFieldList>   A list of names to declare as fields to the compiler.

IN <idAlias>   An alias to assume when there are unaliased references to the names specified in the <idFieldList>.

Description

When you use the FIELD statement to declare fields, unaliased references to variables in *<idFieldList>* are treated as if they were preceded by the special field alias (_FIELD->) or *<idAlias>->* if the IN clause is specified.

Like other variable declaration statements (i.e., LOCAL and MEMVAR), you must place FIELD statements before any executable statements (including PRIVATE, PUBLIC, and PARAMETERS) in the routine you are defining. The FIELD statement has no effect on the macro operator, which always assumes memory variables.

The FIELD statement neither opens a database file nor verifies the existence of the specified fields. It is useful primarily to ensure correct references to fields that are known to exist at runtime. Attempting to access the fields when the associated database is not in use will raise a runtime error.

Examples

This function includes statements to declare database field names in both the current and Employee work areas:

```
FUNCTION DisplayRecord()
   FIELD CustNo, OrderNo, Orders
   FIELD EmpName, EmpCode IN Employee
   USE employee NEW
   USE orders NEW

   ? CustNo                    // Refers to Orders->CustNo
```

```
? EmpName                        // Refers to Employee->EmpName

CLOSE orders
CLOSE employee
```

See Also

DBFieldInfo(), LOCAL, MEMVAR, STATIC

1.8.4.17.2 LOCAL Statement

Purpose

Declare and initialize local variables and arrays.

Syntax

```
[STATIC] LOCAL <idVar> [:= <uValue>] [,...] [AS | IS <idType>]
[, ...]
[STATIC] LOCAL DIM <ArraySpec> [, ...] AS | IS <idType> [, ...]
[STATIC] LOCAL <ArraySpec> [, ...] [AS ARRAY] [, ...]
LOCAL ARRAY <arrayName> ( <nRows> [, <nColumns>] ) [, <arrayName>
( <nRows> [, <nColumns>] ) ]     // FoxPro dialect only
LOCAL ARRAY <arrayName> [ <nRows> [, <nColumns>] ] [, <arrayName>
[ <nRows> [, <nColumns>] ] ]     // FoxPro dialect only
```

Note: The LOCAL statement is shown using several syntax diagrams for convenience.
You can declare variables, dynamic arrays, and dimensioned arrays using a single
LOCAL statement if each definition is separated by a comma.

Arguments

STATIC   Causes the local variable to retain its value across
                              invocations of the declaring entity but does not affect its
                              visibility.

<idVar>   A valid identifier name for the local variable to declare.

<uValue>   The initial value to assign to the variable.

                              For LOCAL, this can be any valid expression.

	For STATIC LOCAL, this value can be a literal representation of one of the data types listed below or a simple expression involving only operators, literals, and DEFINE constants; however, more complicated expressions (including class instantiation) are not allowed.
	Note: Although <uValue> can be a literal array, it must be one-dimensional. Multi-dimensional literal arrays are not allowed. For example, {1, 2, 3} is allowed, but {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}} is not.
	Note: Although the Chr() function cannot be used in <uValue>, the _Chr() operator can. _Chr() is otherwise identical in functionality to Chr().
	If <uValue> is not specified, the initial value of the variable depends on the data type you declare (e.g., NIL if you do not use strong typing, 0 for AS INT, etc.)
DIM <ArraySpec>	The specification for a dimensioned array to declare.
<ArraySpec>	The specification for a dynamic array to declare. In both cases, <ArraySpec> is one of the following: <idArray>[<nElements>, <nElements>, <nElements>] <idArray>[<nElements>][<nElements>][<nElements>] All dimensions except the first are optional.
	<idArray> is a valid identifier name for the array to declare. For dynamic arrays, array elements are initialized to NIL. For dimensioned arrays, the initial value of the elements depends on the data type as explained above for <uValue>.
	<nElements> defines the number of elements in a particular dimension of an array. The number of dimensions is determined by how many <nElements> arguments you specify.
	<nElements> can be a literal numeric representation or a simple numeric expression involving only operators, literals, and DEFINE constants; however, more complicated expressions (such as function calls) are not allowed.
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
IS <idType>	Specifies a structure data type in which the memory needed to hold the structure is allocated on the stack (<idStructure> is the only <idType> allowed with the IS keyword.) See the VOSTRUCT entry in this guide for more information on data

structure memory allocation.

AS ARRAY	For dynamic array declarations, specifies the data type of the entire array.
<arrayName>	Variable name of array . The array will have the dimensions as declared with <nRows> and <nColumns>. The array may be declared with parentheses as delimiters but also with square brackets. We recommend the use of square brackets.

Description

LOCAL is a declaration statement that declares one or more variables or arrays as local to the current routine (i.e., function, method, or procedure). Like other variable declaration statements (such as FIELD and MEMVAR), you must place LOCAL statements before any executable statements (including PRIVATE, PUBLIC, and PARAMETERS) in the routine you are defining.

Local variable declarations hide all inherited private variables, visible public variables, instance variables, global variables, and constants with the same name. The search order for a variable name is as follows:
1.   LOCALs, local parameters, MEMVARs, and FIELDs
2.   SELF instance variables (i.e., without <*idObject*>: prefix in class methods)
3.   GLOBALs and DEFINEs

A LOCAL statement that declares a variable name which is already declared within the same routine (with FIELD, LOCAL, or MEMVAR) causes a compiler error.

Local variables are visible only within the current routine, and unlike private variables, are not visible within invoked routines. If a routine is invoked recursively, each recursive activation creates a new set of local variables.

Unless you specify the STATIC keyword, local variables are created automatically each time the routine in which they were declared begins executing. They continue to exist and retain their values until the declaring routine returns control to the routine that invoked it.

The STATIC keyword serves as a lifetime modifier for a local variable, preventing the variable from being released from memory when the creating entity returns to its calling routine.

Important! When an application containing static variable declarations is invoked, the variables are created and initialized before the beginning of program execution. Thus, initial values are assigned only once per application run, not each time the creator is called.

Notes

Local parameters: The FUNCTION, METHOD, and PROCEDURE statements allow you to declare a list of local parameters enclosed in parentheses following the entity name. For example:

```
FUNCTION <idFunction>(<idParamList>)
```

Exporting locals through code blocks: When you create a code block, you can access local variables defined in the creating entity within the code block definition without passing them as parameters (because local variables are visible to the code block). This, along with the fact that you can pass a code block as a parameter, allows you to export local variables. For example:

```
FUNCTION One()
    LOCAL nVar := 10 AS INT, cbAdd AS CODEBLOCK
    cbAdd := {|nValue| nValue + nVar}

    ? NextFunc(cbAdd)        // Result:  210

FUNCTION NextFunc(cbAddEmUp)
    RETURN EVAL(cbAddEmUp, 200)
```

When the code block is evaluated in NextFunc(), *nVar*, which is local to function One(), becomes visible even though it is not passed directly as a parameter.

Macro expressions: You cannot refer to local variables within macro variables and expressions. If you refer to a local variable within a macro variable, a private or public variable with the same name will be referenced instead. If no such variable exists, a runtime error will be raised.

Type of a local variable: Since Type() uses the macro operator (&) to evaluate its argument, you cannot use it to determine the type of a local variable or an expression containing a local variable reference. You can, however, use ValType() which evaluates its argument and returns the type of the return value

Memory files: You cannot SAVE or RESTORE local variables.

Examples

The following example declares two local arrays and two local variables:

```
LOCAL aArray1[20, 10], aArray2[20][10], var1, var2
```

This example declares two local variables with initializers. The first is initialized to a date value and the second to an array:

```
LOCAL dWhen := TODAY()
LOCAL aVegies := {"Tomato", "Chickadee", "Butterbean"}
```

In this example, the variable *x* and the dimensioned array *z* are typed as INT, while the variables *cName* and *cAddr* are typed as STRING:

```
LOCAL x, DIM z[100] AS INT, cName, cAddr AS STRING
```

The next example declares static variables both with and without initializers:

```
STATIC LOCAL aArray1[20, 10], aArray2[20][10]
STATIC LOCAL cVar, cVar2
STATIC LOCAL cString := "my string", var
STATIC LOCAL aArray := {1, 2, 3}
```

Here a static variable is manipulated within a function. In this example, a count variable increments itself each time the function is called:

```
FUNCTION MyCounter(nNewValue)
            // Initial value assigned once
    STATIC LOCAL nCounter := 0
    IF nNewValue != NIL
            // New value for nCounter
    nCounter := nNewValue
    ELSE
            // Increment nCounter
        ++nCounter
    ENDIF
    RETURN nCounter
```

See Also

FIELD, FUNCTION, DEFINE, GLOBAL, MEMVAR, METHOD, PROCEDURE, STATIC, DIMENSION, DECLARE, PUBLIC

1.8.4.17.3 LPARAMETERS Statement

The LPARAMETERS Statemenent is identical to the PARAMETERS statement.
The Variables however will be created as LOCAL variables and not as Dynamic Memory
Variables and the optional <Type> clause is respected.

Purpose

Create local variables to receive passed values or references.

Syntax

```
LPARAMETERS <idParameterList>
LPARAMETERS <Parameter1> [ AS <Type> [ OF <ClassLibrary> ] ] [,
<Parameter2> [ AS <Type> [ OF <ClassLibrary> ] ] ]
```

Arguments

`<idParameterList>`	One or more parameter variables separated by commas. These variables are used to receive arguments that you pass when you call the routine. The variables will be dynamic memory variables
`<Type>` & `<ClassLibrary>`	The compiler recognizes the AS <Type> and the AS <Type> of <Classlibrary> clauses in the FoxPro dialect. The <ClassLibrary> is ignored but the <Type> is enforced.

Description

When a LPARAMETERS statement executes, all variables in the parameter list are
created as local variables.
Parameters can also be declared as local variables if specified as a part of the
PROCEDURE or FUNCTION declaration statement (see the example). Parameters
specified in this way are referred to as formal parameters. Note that you cannot specify
both formal parameters and a PARAMETERS statement within a procedure or function
definition.
Attempting to do this results in a compiler error.
The number of receiving variables does not have to match the number of arguments
passed by the calling routine. If you specify more arguments than parameters, the extra
arguments are ignored. If you specify fewer arguments than parameters, the extra
parameters are created with a NIL value. If you skip an argument, the corresponding
parameter is initialized to NIL.
The PCount() function returns the position of the last argument passed in the list of
arguments. This is different than the number of parameters passed, since it includes
skipped parameters.

Examples

This function receives values passed into private parameters with a PARAMETERS statement:

```
FUNCTION MyFunc()
LPARAMETERS cOne, cTwo, cThree
? cOne, cTwo, cThree
```

The next example is identical, but receives values passed into local variables, declared within the FUNCTION declaration:

```
FUNCTION MyFunc(cOne, cTwo, cThree)
? cOne, cTwo, cThree
```

See Also

PARAMETERS

1.8.4.17.4 STATIC Statement

Purpose

Declare and initialize static variables and arrays.

Syntax

```
STATIC [LOCAL] <idVar> [:= <uValue>] [, ...] [AS | IS <idType>] [,
...]
STATIC [LOCAL] DIM <ArraySpec> [, ...] AS | IS <idType> [, ...]
STATIC [LOCAL] <ArraySpec> [, ...] [AS ARRAY] [, ...]
```

Note: The STATIC statement is shown using several syntax diagrams for convenience only. You can declare variables, dynamic arrays, and dimensioned arrays using a single STATIC statement if each definition is separated by a comma.

<idVar>	A valid identifier name for the local variable to declare.
<uValue>	The initial value to assign to the variable.

For LOCAL, this can be any valid expression.

For STATIC LOCAL, this value can be a literal representation of one of the data types listed below or a simple expression involving only operators, literals, and DEFINE constants; however, more complicated expressions (including class instantiation) are not allowed.

Note: Although <uValue> can be a literal array, it must be one-dimensional. Multi-dimensional literal arrays are not allowed. For example, {1, 2, 3} is allowed, but {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}} is not.

Note: Although the Chr() function cannot be used in <uValue>, the _Chr() operator can.   _Chr() is otherwise identical in functionality to Chr().

If <uValue> is not specified, the initial value of the variable depends on the data type you declare (e.g., NIL if you do not use strong typing, 0 for AS INT, etc.)

DIM <ArraySpec>	The specification for a dimensioned array to declare.
<ArraySpec>	The specification for a dynamic array to declare. In both cases, <ArraySpec> is one of the following: <idArray>[<nElements>, <nElements>, <nElements>] <idArray>[<nElements>][<nElements>][<nElements>] All dimensions except the first are optional. <idArray> is a valid identifier name for the array to declare. For dynamic arrays, array elements are initialized to NIL. For dimensioned arrays, the initial value of the elements depends on the data type as explained above for <uValue>. <nElements> defines the number of elements in a particular dimension of an array. The number of dimensions is determined by how many <nElements> arguments you specify. <nElements> can be a literal numeric representation or a simple numeric expression involving only operators, literals, and DEFINE constants; however, more complicated expressions (such as function calls) are not allowed.
AS <idType>	Specifies the data type. If omitted, then depending on the compiler options the type will be either USUAL or determined by the compiler.
IS <idType>	Specifies a structure data type in which the memory needed to hold the structure is allocated on the stack (<idStructure>

is the only <idType> allowed with the IS keyword.) See the
[VOSTRUCT](#) entry in this guide for more information on data
structure memory allocation.

AS ARRAY | For dynamic array declarations, specifies the data type of the entire array.

<arrayName> | Variable name of array . The array will have the dimensions as declared with <nRows> and <nColumns>. The array may be declared with parentheses as delimiters but also with square brackets.
We recommend the use of square brackets.

Description

The STATIC keyword serves as a lifetime modifier for variables declared with the LOCAL statement, preventing them from being released from memory when the creating entity returns to its calling routine. It is listed here as a separate entry because, the LOCAL keyword being optional, STATIC can stand alone in a routine as a variable declaration statement. Refer to the LOCAL statement for a description and notes concerning static variable declarations.

See Also

[LOCAL](#)

1.8.4.17.5 STACKALLOC

The STACKALLOC keyword allows you allocate a block of memory on the stack of the current function / method
The syntax to use STACKALLOC is

```
VAR x := StackAlloc <dword>{1,2,3,4,5,6,7,8,9,10}
VAR y := StackAlloc int[]{10}
```

or more generic

```
VAR x := StackAlloc <typed Literal array>
VAR y := StackAlloc <typed array>
```

The compiler will generate variables of the type "Typed PTR". So in the first example x will be of type DWORD PTR and y will be of type INT PTR.

You can also declare the variables with a normal LOCAL keyword. In that case the type must be <Type> PTR

```
LOCAL x := StackAlloc <dword>{1,2,3,4,5,6,7,8,9,10} AS DWORD PTR
LOCAL y := StackAlloc INT[]{10} AS INT PTR
```

You can also use STACKALLOC for an expression that is not a variable declaration. In that case the compiler will resolve the STACKALLOC expression to an allocation of an object of type System.Span<T>. This type is not available in the .Net Framework, but only in .Net 5 and later.

1.8.4.17.6 VAR Statement

1.8.5 Expressions

Expressions are an important element of the language. There are many types of expressions. The expression rule in the compiler is the biggest rule
The table below lists the various expression types in the order they are recognized by the compiler. Some expression types are covered in a separate topic.
As you can see most expression types are recursive. They contain one or more references to sub expressions

Expressions

Expression type	Syntax
Member Access	expression? (.|:) identifier
Member Access with Ampersand (resolved at runtime)	expression? (.|:) &identifier expression? (.|:) &(expr)
Method Call	expression (argumentlist?)
Arrayelement Access	expression [argumentlist]
Conditional Access	expression ? boundexpression
Typecast	(datatype)
Postfix	expression (++ | --)
Await	AWAIT expression
StackAlloc	STACKALLOC expression
Prefix	(+|-|~|@|++|--|^) expression
Typecheck	expression IS datatype VAR newVariable
As Typecheck	expresion ASTYPE datatype
Powerof	expression (^|**) expression
MulDiv	expression (*|/|%) expression
PlusMinus	expression (+|-) expression
LShift	expression << expression
RShift	expression >> expression
Comparison	expression OPERATOR expression, where OPERATOR is

Expression type	Syntax
	<,<=, >, >=, =, ==, $, !=, <>, #
Bitwise And	expression & expression
Bitwise XOR	expression ~ expression
Bitwise OR	expression | expression
Not Expression	(.NOT. | !) expression
Logical AND expression	expression (.AND.|&&) expression
Logical OR expression	expression (.OR. | ||) expression
Default expression	expression DEFAULT expression
Assignment expression	expression ASSIGN_OP expression where ASSIGN_OP is :=, +=, -=, *=, ^=, /=, %=, &=, |=, <<=, >>=, ~=, ??=
Primary expression	[See other rule](#)

1.8.5.1 Bound Expressions

Bound expressions are expressions that access properties or members from objects or structures

Expression type	Syntax
Bound Member Access	boundexpression (.|:) simpleName
Bound Method Call	boundexpression (argumentlist?)
Bound Array Access	boundexpression [argumentlist]

Expression type	Syntax
Bound Conditional Access	expression ? boundexpression
Bind Member Access	(.|:) simpleName
Bind Array Access	[argumentlist]

1.8.5.2 Primary Expressions

Primary Expressions are the simplest building blocks in the expression rule in X#

Primary Expression type	Syntax
Self	SELF
Super	SUPER
LiteralArray	{......}
Anonymous Type	CLASS { .... }
Tuple	TUPLE { .... }
CodeBlock	{|| ....}
Lambda Expression	{ .. => }
LINQ query	FROM .. SELECT ..
DelegateConstructor call	datatype { expression, @Identifier() }
Constructor call with optional initializer	datatype { argumentlist? } initializer? }
Checked	CHECKED (expression)
Unchecked	UNCHECKED (expression)
Typeof	(_typeof | TYPEOF) (expression)
Sizeof	(_sizeof | SIZEOF) (expression)
Nameof	NAMEOF (identifier)
Default	DEFAULT (expression)
Name	identifier

Primary Expression type	Syntax
Conversion	nativetype (expression): LONG(1+2)
XBase Conversion	xbasetype (expression): FLOAT(1+2)
VOCast	datatype (_CAST , exprsssion)
VoCastPtr	PTR (datatype , expression)
VOTypeName	typeName
IIF expression	(IIF|IF) (expression , expression , expression)
Bitwise	(_AND, _OR, _XOR, _NOT) (expressionlist)
AliasedField	FIELD -> identifier | identifier -> identifier
Aliased Expression	FIELD -> expression | expression -> expression
MacroExpr	& (expression)
MacroVar	& identifier
Parenthesized	(expression)
ArgList	_ARGLIST

1.8.5.3 Codeblocks

Codeblocks are an important part of the X# language.

Traditionally the codeblock looked like

```
codeblock          : { | codeblockParamList? | expression }
                   ;
codeblockParamList : identifier (, identifier)*
                   ;
```

For example

```
{|a,b| a*b}
```

X# has extended the Codeblock rule. We now not only accept a single expression, but also a statement list and an expressionlist:

```
codeblock          : { | codeblockParamList? |
                     ( expression
                     | eos statementblock
                     | codeblockExpressionList )
                     }
                     ;



codeblockExprList  : (expression? ,)+ expression          //
The last expression is the return value of the block
                  ;
```

Examples of this are

```
{|a,b| a:= Sqrt(a), a*b}
{|a,b|
    ? a
    ? b
}
```

The second example can be seen as an anonymous method with 2 parameters

1.8.5.4 Lamda Expressions

Lambda Expressions are a bit like a codeblock. it is a block of code with optional parameters and a return value.
They are used to create anonymous functions.
Unlike a codeblock they do NOT inherit from a common class. They also cannot be stored inside the USUAL type.

Code that expects a lambda expression often declares a delegate that describes the parameter types and return value of the lambda expression.
Lambda expressions can also be used to declare an event handler.

The return value of the lambda expression is the last expression in the expression list, or the value returned by the return statement in the statement list.

```
{ [pars] => expression [, expressionlist] }
```

or

```
{ [pars] =>
   statements     // these are NOT separated with semi colons!
}
```

pars   Comma separated list of arguments. They can optionally be
                                          typed.

Some examples:

```
// single expression, untyped parameters
{ a, b => a * b}

// single expression, typed parameters
{ a as int, b as int => a / b}

// typed parameters, expression list, last expression is return
value
{ a as int, b as int => a := iif(a == 0, 1, a), b / a}

// statement list, no semi colons needed, return statement is
return value
{ a as int, b as int =>
   if a == 0
        a := 1
   endif
   return b / a
}

// Register a click event handler. No need to type the args
// The signature is derived from the signature of the Click Event
btnOk:Click += { sender, args => SELF:Close() }
```

1.8.5.5   LINQ Expressions

The LINQ expression rule is:

```
linqquery          :  fromClause queryBody
                   ;

fromClause         : FROM identifier (AS typeName)? IN expression
                   ;

queryBody          : (queryBodyClause)* selectOrGroupclause
(queryContinuation)?
                   ;

queryBodyClause    : fromClause

                   | LET identifier ASSIGN_OP expression

                   | WHERE expression

                   | JOIN identifier (AS typeName)? IN expression
ON expression EQUALS expression joinIntoClause?
```

```
                              | ORDERBY ordering (, ordering)*

                              ;

    joinIntoClause      : INTO identifier
                              ;

    ordering            : expression (ASCENDING|DESCENDING)?
                              ;

    selectOrGroupclause : SELECT expression
                          | GROUP expression BY expression
                              ;

    queryContinuation   : INTO identifier queryBody
```

An example of a LINQ Query

```
    VAR oAll := FROM D IN oDev  ;
            JOIN C IN oC ON D:Country EQUALS C:Name   ;
            ORDERBY D:LastName  ;
            SELECT CLASS {D:Name, D:Country, C:Region}
```

In this example you will see

```
    fromclause:             FROM D in oDev
    querybody 1:            JOIN C IN oC ON D:Country EQUALS C:Name
    querybody 2:            ORDERBY D:LastName
    selectOrGroupClause:    SELECT CLASS {...}
```

Other examples can be found in the LINQ Example topic

1.8.5.6 Initializers

X# has added two types of initializers to the language: collection Initializers and Object Initializers. The syntax for these is:

```
    constructorcall         : datatype { parameterlist? }
    initializer?
                              ;

    initializer             : objectinitializer
                          | collectioninitializer
                              ;
```

```
objectinitializer          : { (memberinitializer (,
memberinitializer)*)? }
                           ;

memberinitializer          : Name=identifierName :=
Expr=initializervalue
                           ;

initializervalue           : objectOrCollectioninitializer
                           | expression
                           ;

collectioninitializer      : { expression (, expression)* }
                           ;
```

Note:

- The initializer which is also delimited by curly braces immediately follows the closing curly brace of the constructor call

An example of an object initializer:

```
    VAR oPerson := Person{}{FirstName := "John", LastName :=
"Smith"}
    VAR oPerson := Person{"John", "Smith"} {Age := 35 }
```

An example of a collection initializer

```
    oList := List<Int>{} {1,2,3,4,5}
```

And combined:

```
    Var oPeople := List<Person> {} {;
                                Person{}{FirstName :=
"John", LastName := "Smith"}, ;
                                Person{}{FirstName :=
"Jane", LastName := "Doe"} ;
                           }
```

The LINQ example topic shows different initializers in action.

1.8.5.7 Compiler Macros

The following defines can be used in your code and will be replaced by the compiler with a literal value:

Name	Type	Value

__ARRAYBASE__	Integer	0 or 1 depending on the /az compiler option
__CLR2__	String Literal	__CLR2__ (only for compatibility with Vulcan, x# does not implement the /clr compiler option). See comment below.
__CLR4__	String Literal	__CLR4__ (only for compatibility with Vulcan, x# does not implement the /clr compiler option). See comment below.
__CLRVERSION__	String Literal	"2" or "4"depending on the version. (only for compatibility with Vulcan, x# does not implement the /clr compiler option). See comment below.
__DATE__	String Literal	Current date in YYYYMMDD format
__DATETIME__	String Literal	Current date/time in format from regional settings
__DIALECT__	String	Name of the current dialect
__DIALECT_CORE__	Logical	Defined with TRUE when Core dialect is selected
__DIALECT_FOXPRO__	Logical	Defined with TRUE when FoxPro dialect is selected
__DIALECT_HARBOUR__	Logical	Defined with TRUE when Harbour dialect is selected
__DIALECT_VO__	Logical	Defined with TRUE when VO dialect is selected
__DIALECT_VULCAN__	Logical	Defined with TRUE when Vulcan dialect is selected
__DIALECT_XBASEPP__	Logical	Defined with TRUE when Xbase++ dialect is selected
__DEBUG__	Logical Literal	TRUE when compiling in debug mode. Undefined in Release mode
__ENTITY__	String Literal	Name of current entity
__FILE__	String Literal	Current source file name
__FOX1__, __FOX2__	Logical	Current value of the FoxPro compatibility compiler options /fox1 and /fox2
__FUNCTION__	String Literal	Current function/method name without class prefix and in original case
__FUNCTIONS__	String Literal	Returns the name of the current Functions class
__HARBOUR__	Logic literal	TRUE when the Harbour dialect is selected. Otherwise not defined.
__LINE__	String Literal	Current source line number
__MEMVAR__	Logical	TRUE when the /memvar compiler option is used

__MODULE__	String Literal	Current source file name
__SIG__	String Literal	Signature of current entity
__SRCLOC__	String Literal	Filename and line number in the source
__SYSDIR__	String Literal	Systemdir (on developers machine!)
__TIME__	String Literal	Compile time in HH:mm:ss format
__UNDECLARED__	Logical Literal	TRUE when the /undeclared compiler option is used
__UNSAFE__	Logical Literal	TRUE when the /unsafe compiler option is used
__UTCTIME__	String Literal	UTC Compile time in HH:mm:ss format
__VERSION__	String Literal	Version of the compiler
__VO__	Logic literal	TRUE when the VO dialect is selected. Otherwise not defined.
__VO1__ , __VO2__ ... __VO17__	Logical	Current value of the matching VO compatibility compiler option, /vo1, /vo2 ... /vo17
__VULCAN__	Logic literal	TRUE when the Vulcan dialect is selected. Otherwise not defined.
__WINDIR__	String Literal	Windows dir (on developers machine!)
__WINDRIVE__	String Literal	Windows drive (on developers machine!)
__XPP__	Logic literal	TRUE when the Xbase++ dialect is selected. Otherwise not defined.
__XPP1__	Logical	Current value of the /xpp1 compiler option. Only defined in the Xbase++ dialect.
__XSHARP__	Logical Literal	Always TRUE
__XSHARP_RT__	Logical Literal	TRUE when compiling against the X# runtime. Not defined otherwise.

Note

The CLR2 and CLR4 version is determined by the X# compiler by looking at the version of either system.dll or mscorlib.dll

1.8.5.8 Pseudo Functions

The following pseudo functions are supported by the X# compiler

Function	Description
PCOUNT()	This pseudo function is only available in methods or functions with a CLIPPER calling convention. It returns the number of argument passed to the function. The

	function does not expect and does not allow any arguments. ***Not available in Core.***
ARGCOUNT()	This pseudo function returns the number of arguments defined for the current method or function.
_GETMPARAM() and **_GETFPARAM**()	These pseudo functions are only available in methods or functions with a CLIPPER calling convention. You can use them to retrieve a function parameter by position. You must pass a numeric expression to these functions. If you pass a number that is larger than the actual number of parameters at runtime then you will get an array access exception. ***Not available in Core.***
String2Psz() and **Cast2Psz**()	These pseudo functions are used to convert DotNet strings to unmanaged Ansi PSZ strings. Not only is a PSZ created, but the functions also change the code generation and set up code to clear the allocated PSZ variable on exit of the function in which they are created. ***Not available in Core.***
ALTD()	This function will insert a call to System.Diagnostics.Debugger.Break inside a check to see if the debugger is attached (System.Diagnostics.Debugger.IsAttached)
_GetInst()	This function will return the module handle for the current module. Behind the scenes this is translated to System.Runtime.InteropServices.Marshal.GetHINSTANCE(TypeOf(FunctionsClass):Module)
PCALL() and **CCALL()**	The methods are used to call an API function for a strongly typed PTR. The function expects a first parameter of type PTR and the other parameters must match the parameters defined in the function to which the typed PTR points The compiler creates a delegate with the proper prototype and uses Marshal.GetDelegateForFunctionPointer() to call the function.
PCallNative<Type>() and **CCallNative<Type>()**	The methods are used to call an API function for an untyped PTR The function expects a generic type parameter which indicates the return type and a first parameter of type PTR. Other parameters are allowed and must not point to managed memory. The compiler creates a delegate with the proper prototype and uses Marshal.GetDelegateForFunctionPointer() to call the function.
_ARGS()	This pseudo function returns is replaced by the compiler to a reference to the generated array of parameters for functions/methods with clipper calling convention

SLen()	This function is translated by the compiler to a call of the Length property of the string, with a built-in check for NULL.
Chr(), _Chr()	When the numeric parameter of this function is a literal number between 0 and 127 then the compiler replaces the function call with a literal string with a character of that value. Larger values are not converted at compile time but at runtime because these numbers are codepage dependent. So an expression like `"Hello world"+Chr(13)+Chr(10)` will be translated into a literal string containing "Hello world" followed by the **CRLF** characters (the compiler concatenates the strings at compile time).

1.8.6 Operators

X# has many operators. These are
- [Binary operators](#)
- [Assignment operators](#)
- [Logical operators](#)
- [Bitwise operators](#)
- [Relational operators](#)
- [Shift operators](#)
- [Unary operators](#)
- [Workarea operators](#)
- [IIF() Operator](#)
- [SizeOf() Operator](#)
- [TypeOf() Operator](#)
- [NameOf() Operator](#)

1.8.6.1 Binary

X# uses the following binary operators:

Operator	Example	Meaning
+	x + y	addition
-	x - y	subtraction
*	x * y	multiplication
/	x / y	division. If the operands are integers, the result is an integer truncated toward zero (for example, -7 / 2 is -3).
%	x % y	modulus. If the operands are integers, this returns the remainder of dividing x by y. If q = x / y and r = x % y, then x = q * y + r.
^ or **	x ^ y or x ** y	power of. x ^ y returns x to the power of y
default or ??	x DEFAULT y x ?? y	When x is not null then x, otherwise y. Can be chained: x DEFAULT y DEFAULT z or x ?? y ?? z

Binary operators and XBase types

The following XBase types support binary operators.

Type	Operator	Description
ARRAY	none	binary operators are not supported for arrays
DATE	+ and -	You can add and subtract a numeric to a date, which is the equivalent of adding days.

You can subtract a date from a date which will return the number of days between the dates

FLOAT	all	You can perform all binary operations on FLOATs when both operands are numeric. The compiler will automatically insert a conversion from <any numeric type> to FLOAT when the right hand side of the binary operator is not a float
SYMBOL	none	binary operators are not supported for arrays
STRING	+ and -	The + operator will add 2 strings. The - operator will add the RHS to the LHS and move all trailing spaces of the LHS to the end of the resulting string. You can also add STRING values and SYMBOL values. The SYMBOLs will automatically be converted to strings
USUAL	all	You can use all binary operators on USUALs. The code in the runtime will check to see if the 2 sides are "compatible" and will produce a runtime error when the operation is not available.

1.8.6.2 Assignment operators

X# uses the following Assignment operators:

Operator	Example	Meaning
:=	x := y	Store the value of the second operand in the object specified by the first operand (simple assignment).
=	x = y	Store the value of the second operand in the object specified by the first operand (simple assignment). **This is allowed in the VFP dialect only ! In all other dialects assigning a value with a '=' operator will generate a warning.**
+=	x += y	Add the value of the second operand to the value of the first operand; store the result in the object specified by the first operand
-=	x -= y	Subtract the value of the second operand from the value of the first operand; store the result in the object specified by the first operand.
/=	x /= y	Divide the value of the first operand by the value of the second operand; store the result in the object specified by the first operand
%=	x %= y	Take modulus of the first operand specified by the value of the second operand; store the result in the object specified by the first operand.
*=	x *= y	Multiply the value of the first operand by the value of the second operand; store the result in the object specified by the first operand.

^= or **=	x ^= y	Calculate the exponent of the first operand and the second operand ; store the result in the object specified by the first operand. **Please note that in languages such as C# the ^= operator performs a Bitwise XOR**
|=	x |= y	Obtain the bitwise inclusive OR of the first and second operands; store the result in the object specified by the first operand.
&=	x &= y	Obtain the bitwise AND of the first and second operands; store the result in the object specified by the first operand.
~=	x ~= y	Obtain the bitwise exclusive OR of the first and second operands; store the result in the object specified by the first operand.
<<=	x <<= y	Shift the value of the first operand left the number of bits specified by the value of the second operand; store the result in the object specified by the first operand.
>>=	x >>= y	Shift the value of the first operand right the number of bits specified by the value of the second operand; store the result in the object specified by the first operand.
?=	a ?= "somevalue"	When a is NULL then it will be assigned "somevalue". Otherwise a is not changed

Assignment operators and XBase types

See the topic about Binary operators to see which complex assignment operators are supported.

1.8.6.3 Logical

X# uses the following Logical operators:

Operator	Example	Meaning
.AND.	x .AND. y	Returns TRUE when both operands are TRUE, otherwise FALSE
.OR.	x .OR. y	Returns TRUE with one or both operands are TRUE. Only when both operands are FALSE then FALSE is returned
.NOT. or !	.NOT. x	Reverses TRUE and FALSE.

1.8.6.4 Bitwise

X# uses the following Bitwise (binary) operators. There are simple character versions of these and also pseudo functions:

Operator	Pseudo Function	Example	Meaning

|	_OR(..)	x | y, _OR(x,y)	Returns the bitwise OR of x and y _OR() may have more than 2 parameters.
~	_XOR(..)	x ~ y, _XOR(x,y)	Returns the bitwise XOR of x and y
&	_AND(..)	x & y, _AND(x,y)	Returns the bitwise AND of x and y. _AND() may have more than 2 parameters.
~	_NOT(..)	~ x, _NOT(x)	Returns the bitwise NOT of x (aka one's complement). _NOT() can only have one parameter

The result of a bitwise operation is best understood via 'truth tables'. If two numbers can, for instance, be defined by 4 bits (expressing numbers 0 up to 15 in decimal value) then, when 'Anding' them, use the AND truth table for each bit in turn. If the values in decimal are 5 and 1 then their bit representations are 0101 and 0001.

AND

A	B	Result
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	Result
0	0	0
0	1	1
1	0	1
1	1	1

XOR

A	B	Result
0	0	0
0	1	1
1	0	1
1	1	0

NOT

A	Result
0	1
1	0

1.8.6.5 Relational

X# uses the following Logical operators:

Operator	Example	Meaning
<	x < y	less than (true if x is less than y). **See below for string comparisons**
<=	x <= y	less than or equal to (true if x is less than or equal to y). **See below for string comparisons**
>	x > y	greater than (true if x is greater than y). **See below for string comparisons**
>=	x >= y	greater than or equals to (true if x is greater than or equal to y). **See below for string comparisons**
=	x = y	equality. ***Note that there is a difference between = and == for strings only. See below***

==	x == y	exact equality. *Note that there is a difference between = and == for strings only. See below*
<>, #, !=	x <> y, x # y, x != y	not equal **Note that for strings this follows the same rules as the single = operator.**
$	x $ y	Is substring of. Returns true if the first string is a substring of the second (case sensitive !)
IS	x IS y	Type compatibility. Returns true if the evaluated left operand can be cast to the type specified in the right operand (a static type).
ASTYPE	x ASTYPE y	Type conversion. Returns the left operand cast to the type specified by the right operand (a static type), but as returns null where (T)x would throw an exception.

String comparisons

The '=' and '==' operators behave differently for strings, and the behavior of the single equals also depends on a runtime setting.

If you call SetExact(FALSE) then '=' equates the characters up to the length of the string on the right-hand side of the operator ignoring the remaining characters on the left. This is the default setting. If you call SetExact(TRUE) then = and == have the same meaning for strings.

The <, <=, > and >= operators for strings have a behavior that depends on a **compiler option** and a **runtime setting**. The -vo13 compiler option 'compatible string comparions' tells the compiler that it needs to use a runtime function for string comparisons. The behavior of this runtime function depends on the setting of SetCollation(). There are 4 possible values for SetCollation():

Setting	Description
Clipper	This setting will convert both strings to OEM strings using the current DOS codepage. After that the strings will be compared using the string comparison / weight tables that are defined with SetNatDLL(). The default comparison uses a wight based on the byte number. Other comparisons available are for example GERMAN, DUTCH, FRENCH, RUSSION, SPANISH, SLOV852 etc. This setting should be used if your application needs to share files with CLIPPER programs.
Windows	This setting will convert both strings to ANSI using the current ANSI codepage. After that the strings will be compared using the normal windows ANSI CompareString() code. This setting should

		be used when your application shares files with VO programs
Unicode		This setting will NOT convert strings and will do a normal Unicode string comparison using the String.Compare() method from the .Net
Ordinal		This setting will NOT convert strings and will do a normal Ordinal string comparison using the String.CompareOrdinal() method from the .Net. This is the **fastest.**

The >= and <= operators for strings also take into account the setting for SetExact(). The 'equalness' of the 2 strings is determined by the same rules as the '=' operator.

1.8.6.6   Shift

X# uses the following shift operators:

Operator	Example	Meaning
>>	x >> y	shift bits right. If the left operand is signed, then left bits are filled with the sign bit. If the left operand is unsigned, then left bits are filled with zero.
<<	x << y	shift bits left and fill with zero on the right.

Note that shift operators are only supported on integral numeric types

1.8.6.7   Unary

X# uses the following unary operators:

Operator	Example	Meaning
+	+x	returns the value of x.
-	-x	numeric negation.
++ (prefix)	++x	prefix increment. Returns the value of x after updating the storage location with the value of x that is one greater (typically adds the integer 1).
-- (prefix)	--x	prefix decrement. Returns the value of x after updating the storage location with the value of x that is one greater (typically adds the integer 1).
~	~x	bitwise complement
++ (postfix)	x++	postfix increment. Returns the value of x and then updates the storage location with the value of x that is one greater (typically adds the integer 1).
-- (postfix)	x--	postfix decrement. Returns the value of x and then updates the storage location with the value of x that is one less (typically subtracts the integer 1).

Note that shift operators are only supported on integral numeric types

1.8.6.8 Workarea

X# uses the following Workarea operators:

Operator	Example	Meaning
->	Customer->LastName FIELD->FirstName _FIELD->Salary M->Name A->City	Field "LastName" in the "Customer" workarea Field "FirstName" in the current workarea Field "Salary" in the current workarea The dynamic memory variable "Name" (public or private) The field City in workarea 1. Allowed single letter aliases are A .. J. Note that FIELD and _FIELD are synonyms. Keywords that appear directly after the -> operator are seen as identifier and not keyword.
. (Dot)	Customer.LastName	In the VFP dialect we also support the DOT (".") syntax for aliased operations. Please note that this is ambiguous because the compiler cannot detect at compile time if CUSTOMER is a workarea or for example a MEMVAR. If there is a local variable with the name CUSTOMER then this will access the LASTNAME property of the CUSTOMER local variable. In all other cases this will produce code that looks for either a CUSTOMER workarea or memory variable at runtime. Keywords that appear directly after the "." operator are seen as identifier and not keyword.
. (Dot)	M.Name	In the VFP dialect we also support the DOT (".") syntax for memvar access. This is also ambiguous because NAME can be both a local variable or a dynamic memory variable "NAME" (public or private). However the compiler will be able to detect the local variable at compile time and when this is not found then it will access the dynamic memory variable.

1.8.6.9 IIF Operator

The IIF operator returns one of two values, depending on an expression that returns a value of type LOGIC. The IIF operator is of the form:

```
IIF( conditionExpression, trueExpression, falseExpression )
```

Remarks

conditionExpression may be any expression that returns a value of type LOGIC, or a type that can be implicitly converted to LOGIC.

If *conditionExpression* evaluates to TRUE, *trueExpression* is evaluated and is the result. If *conditionExpression* evaluates to FALSE, *falseExpression* is evaluated and is the result. Only one of the two expressions is ever evaluated.

The return type of the IIF operator is determined by the following rules:
Given:
- tT is the type of trueExpression
- tF is the type of falseExpression
- tR is the return type of the IIF expression

1. If tT and tF are the same type, tR is that type.
2. If tT and/or tF is USUAL, then tT or tF is converted to USUAL if necessary, and tR is USUAL.
3. If tT can be implicitly converted to tF, tR is tF.
4. If tF can be implicitly converted to tT, tR is tT.
5. If tT can be implicitly converted to tF and tF can be implicitly converted to tT, then tR is ambiguous and a compiler error is raised.
6. If tT cannot be implicitly converted to tF and tF cannot be implicitly converted to tT, then tR is indeterminate and a compiler error s raised.
   Note that if tT and tF are VOID, tR is VOID and the IIF operator cannot be used as an operand in another expression, or as a function or method parameter.
   In this case, the IIF operator is essentially the same as an IF...ELSE...ENDIF construct and can only be used in stand-alone expression statements.

In cases 5 and 6, it may be possible to resolve the error by specifying an explicit cast on trueExpression or falseExpression.

This behavior is different than in some other XBase dialects In these dialects the return is usually determined by the following rules
1. If tT and tF are the same type, tR is that type.
2. Otherwise tT and tR are implicitly converted to USUAL and tR is USUAL.

You can use the [/vo10](#) compiler option to enable this behavior in X#, but this is only recommended for code originally written in other environments, such as Visual Objects.

Note

For compatibility with Visual Objects and several other dialects the IF() operator is also supported

1.8.6.10 SizeOf Operator

he sizeof operator returns the number of bytes occupied by a variable of a given type. The argument to the sizeof operator must be the name of an [unmanaged type](#) or a type parameter that is [constrained](#) to be an unmanaged type.

Syntax

```
SizeOf( type )
```

Expression	Constant value
sizeof(sbyte)	1
sizeof(byte)	1
sizeof(short)	2
sizeof(word)	2
sizeof(int)	4
sizeof(dword)	4
sizeof(int64)	8
sizeof(uint64)	8
sizeof(char)	2
sizeof(real4)	4
sizeof(real8)	8
sizeof(decimal)	16
sizeof(logic)	1

For some types the size depends on how the program is running, e.g. in 32 bits mode or 64 bits mode. In that case you may see an error message that the SizeOf operator requires the /unsafe compiler option.

Note

_sizeof() (with a leading underscore and parenthesis) is also supported for compatibility with Visual Objects and is synonymous with sizeof.

1.8.6.11 TypeOf operator

In .Net, all types are inherited from the System.Type.

Syntax

```
TypeOf( type )
```

The TypeOf operator gets the System.Type of a type.
This code sample shows the use case of typeof operator using C#.

```
Console.WriteLine(typeof(String))
Console.WriteLine(typeof(USUAL))
```

The following example demonstrates several ways to obtain a type

Example

```
FUNCTION Start() AS VOID
LOCAL t1 AS Type
LOCAL t2 AS Type
LOCAL t3 AS Type
LOCAL t4 AS Type
LOCAL s AS STRING
s := "Live long and prosper!"
t1 := s:GetType()
t2 := typeof(System.String)
t3 := typeof(STRING)
t4 := Type.GetType( "System.String" )
? "Type objects are equal:", t1 == t2 .and. t1 == t3 .and. t1 ==
t4
? t1:Name
? t1:FullName
? t1:AssemblyQualifiedName
RETURN
```

Note

_typeof() (with a leading underscore and parenthesis) is also supported for compatibility with Visual Objects and is synonymous with sizeof.

1.8.6.12 NameOf Operator

A nameof expression produces the name of a variable, type, or member as the string constant:

You can use a nameof expression to make the argument-checking code more maintainable:

Syntax

```
NameOf( identifier )
```

```
FUNCTION Test(cName as STRING)
IF cName == null
    throw new ArgumentNullException(nameof(cName),
i"{nameof(cName)} cannot be null");
endif
```

1.8.7 X# Preprocessor Directives

The XSharp preprocessor supports the following preprocessor directives

#command
#define
#else
#endif
#endregion
[#if](#if)
#ifdef
#ifndef
#include
#line
#region
#stdout
#translate
#undef

1.8.7.1 #command

Purpose

Specify a user-defined command or translation directive

Syntax

```
#command  <matchPattern> => <resultPattern>  // can be abbreviated
to 4 characters
#xcommand <matchPattern> => <resultPattern>  // cannot be
abbreviated to 4 characters
#ycommand <matchPattern> => <resultPattern>  // cannot be
abbreviated to 4 characters and must match case
```

Arguments

`<matchPattern>` is the pattern the input text should match.

`<resultPattern>` is the text produced if a portion of input text matches the
<matchPattern>.

The => symbol between <matchPattern> and <resultPattern> is, along with #command
or #translate, a literal part of the syntax that must be specified in a #command or
#translate directive. The symbol consists of an equal sign followed by a greater than
symbol with no intervening spaces. Do not confuse the symbol with the >= or the <=

comparison operators in the xBase language.

Description

#command and #translate are translation directives that define commands and pseudofunctions. Each directive specifies a translation rule. The rule consists of two portions: a match pattern and a result pattern.
The match pattern matches a command specified in the program (.prg) file and saves portions of the command text (usually command arguments) for the result pattern to use. The result pattern then defines what will be written to the result text and how it will be written using the saved portions of the matching input text.

#command and #translate are similar, but differ in the circumstance under which their match patterns match input text. A #command directive matches only if the input text is a complete statement, while #translate matches input text that is not a complete statement. #command defines a complete command and #translate defines clauses and pseudofunctions that may not form a complete statement. In general, use #command for most definitions and #translate for special cases.

#command and #translate are similar to but more powerful than the #define directive. #define, generally, defines identifiers that control conditional compilation and manifest constants for commonly used constant values such as SDK codes. Refer to any of the header files in the INCLUDE directory for examples of manifest constants defined using #define.

#command and #translate directives have the same scope as the #define directive. The definition is valid only for the current program (.prg) file unless defined in Std.ch or the header specified with the /U option on the compiler command line. If defined elsewhere, the definition is valid from the line where it is specified to the end of the program file. Unlike #define, a #translate or #command definition cannot be explicitly undefined. The #undef directive has no effect on a #command or #translate definition.

As the preprocessor encounters each source line preprocessor, it scans for definitions in the following order of precedence: #define, #translate, and #command. When there is a match, the substitution is made to the result text and the entire line is reprocessed until there are no matches for any of the three types of definitions. #command and #translate rules are processed in stack-order (i.e., last in-first out, with the most recently specified rule processed first).

In general, a command definition provides a way to specify an English language statement that is, in fact, a complicated expression or function call, thereby improving the readability of source code. You can use a command in place of an expression or function call to impose order of keywords, required arguments, combinations of arguments that must be specified together, and mutually exclusive arguments at compile time rather than at runtime. This can be important since procedures and user-defined functions can now be called with any number of arguments, forcing any argument checking to occur at runtime. With command definitions, the preprocessor handles some of this.

Because directives are processed in stack order, when defining more than one rule for a command, place the most general case first, followed by the more specific ones. This ensures that the proper rule will match the command specified in the program (.prg) file.

Match Markers

See the topic about Match Markers for a detailed discussion

Result Markers

See the topic about Result Markers for a detailed discussion

Notes

- **Less than operator**: If you specify the less than operator (<) in the <resultPattern> expression, you must precede it with the escape character (\).
- **Multistatement lines**: You can specify more than one statement as a part of the result pattern by separating each statement with a semicolon. If you specify adjacent statements on two separate lines, the first statement must be followed by two semicolons.

1.8.7.2 #define

Purpose

Syntax

```
#define identifier [token-string]
```

or

```
#define identifier LPAREN parameters RPAREN
```

#define lets you define a symbol, such that, by using the symbol as the expression in a #ifdef directive, the expression will evaluate to true or in a or #ifndef directive the expression will evaluate to false.
#define also allows you define a symbolic name for a token string, so you can use the symbolic name in your code and the preprocessor will replace all occurences of that name with the token string that you have specified.
For example:

```
#define DEBUG
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

or

```
#define VERSION "1.2.0.0"
#define FILEVERSION "1.2.3.4"
// ...
[assembly: AssemblyVersion(VERSION)]
[assembly: AssemblyFileVersion(FILEVERSION)]
```

A define with parentheses, such as

```
#define MAX(a,b) iif(a>b, a, b)
```

will be treated like a #translate.

Notes

Please note that **defines are CASE SENSITIVE**, so the following code will work:

```
#define TEST 123

FUNCTION Test() AS INT
    RETURN TEST
```

but this will NOT compile:

```
#define TEST 123

FUNCTION TEST() AS INT
    RETURN TEST
```

because the preprocessor will replace the name TEST in the FUNCTION line with the value 123 which is not a valid identifier. After preprocessing the code becomes:

```
FUNCTION 123() AS INT
    RETURN 123
```

1.8.7.3 #else

Purpose

#else lets you create a compound conditional directive, so that, if the expression in the preceding #ifdef directive or #ifndef directive does not evaluate to true, the compiler will evaluate all code between #else and the subsequent #endif.
For example, the following code will show the string "Debug Version" if the symbol DEBUG is defined on the command line, else it will show the text "Release Version"

```
// DEBUG may be defined from the command line
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#else
    Console.WriteLine("Release version");
#endif
```

1.8.7.4 #endif

Purpose

#endif specifies the end of a conditional directive, which began with the #ifdef or #ifndef directive. For example:

```
// DEBUG may be defined from the command line
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#else
    Console.WriteLine("Release version");
#endif
```

1.8.7.5 #endtext

Purpose

Mark the end of a #text .. #endtext region

Syntax

```
#text := <varname>
First Line
Second Line
#endtext
```

Description

The language supports the TEXT .. ENDTEXT construct. These commands are converted by the preprocessor into a #text .. #endtext constract.
#endtext always appears "alone" on a line of code and will be replaced by a call to the (optional) endfunction that is declared with the #text directive and when the block is assigned to a variable then the assignment will be performed on the #endtext line.

Example

Please note that the 2 UDCs below are already defined in XSharpDefs.xh

```
#xcommand ENDTEXT => #endtext


#xcommand  TEXT TO FILE <(file)> ;
      => _TextSave( <(file)> ) ;;
           #text QOut, _TextRestore


TEXT TO FILE EXAMPLE.TXT
line 1
line 2
line 3
line 4
ENDTEXT
```

The TEXT TO FILE command is translated into a call to the function _TextSave(), followed by the #text directive, that specifies that each line must be sent to the QOut() function and that also declares that the #endtext line must be replaced by a call to the _TextRestore() function. The QOut() and _TextRestore function names are specified without parameters. Each line in the block will be sent to the QOut() function as parameter.

So this code is converted to

```
_TextSave("EXAMPLE.TXT");
QOut("line 1")
QOut("line 2")
```

```
QOut("line 3")
QOut("line 4")
_TextRestore()
```

See also

TEXT command
#text Directive

1.8.7.6 #if

Purpose

Mark a region in the source code that will only be included in the compilation when a logical condition evaluates to TRUE.

Syntax

```
#if <logical_expression>
<SourceCode included if <logical_expression> evaluates to .T.>
[ #else
<SourceCode included if <logical_expression> othwerwise>
]
#endif


<logical_expression> : <expression>

<expression>  : <unary_operator> <expression>
   // unary prefix expression
             | <expression> <binary_operator> <expression>
   // binary numeric expression
             | <expression> <shift_operator> <expression>
   // binary shift expression
             | <expression> <comparison_operator>
<expression>   // binary logical expression
             | <expression> <bitwise_operator> <expression>
   // binary bitwise expression
             | <expression> <logical_operator> <expression>
   // binary logical expression
             | <negation_operator> <expression>
```

```
         // negation expression
               | <primary_expression>                    // primary
expression

<unary_operator>       : + | - | ++ | --

<binary_operator>      : ^ | * | / | % | + | -

<shift_operator>       : << | >>

<comparison_operator>  : < | <= | > | >= | = | == | != | <>

<bitwise_operator>     : & | |

<logical_operator>     : .AND. | .OR. | .XOR.

<negation_operator>    : ! | .NOT.

<primary_expression> :
<literal_value>                                 // literal expression
               | ( <expression> )                          //
parenthesized expression

<literal_value>        : <string_literal>
                       | <char_literal>
                       | <logical_literal>
                       | <integer>
                       | <double>
                       | <#define_constant>

<string_literal>       :  "double quoted"
                       | 'single_quoted'
                       | [block_quoted]
                       | e"escaped"

<char_literal>         c'<char>'

<logical_literal>      :  .T. | TRUE | .F. | FALSE
```

Description

The #**if**...#**else**...#**endif** directive forms a control structure for the preprocessor.
When the <logical_expression> evaluates to true (.T.), the preprocessor translates and
outputs the source code located between the directives #**if** and #**else** to the intermediary
file, and the source code between the directives #**else** and #**endif** is ignored.
If no #**else** directive is present, the preprocessor translates and outputs the source code
located between the directives #**if** and #**endif** .

If the <logical_expression> evaluates to false (.F.) the source code between the directives #**else** and #**endif** is included only.

The <logical_expression> term can be formed using operands, compare operators and logical operators. A compare operations always requires two operands and will be evaluated prior to logical operations. The operands must be either string literals, numeric literals or logical literals or a valid #**define** constant that results to one of the mentioned literals.

A string will be recognized when it is enclosed within single or double quote characters. If an undefined constant is encountered the result of that term will be false (.F.).

A logical expression consists either of two expression and one logical operator, or simply of one literal.

Type conversions

When an expression mixes types then the preprocessor automatically converts types in the following order for comparisons and calculations:

1. String
2. Double
3. Integer
4. Logic

Example

```
#if 1 > "abc"
   // the 1 is converted to "1" first before the comparison is
done
   ? "Compare number and string"
#endif
#if 1.2 > FALSE
   // the FALSE is converted to 0 first before the comparison is
done
   ? "Compare number and string"
#endif
```

Conversions

For comparisons and calculation operators the types of the 2 operands are compared. If they are equal then no conversion is needed. If they are not of the same type then the following rules are applied.

If one operand is of type	Then the other operand is converted like this
String	Call .ToString() on the value
Double or Decimal	Integer: ToDouble() Logical: TRUE = 1.0, FALSE = 0.0

Integer	Logic: TRUE = 1, FALSE = 0
Logical (The expression on the #if line)	String: Null or Empty = FALSE, All others = TRUE Integer: 0 = FALSE, All others = TRUE Double: 0.0 = FALSE, All others = TRUE

Comparison operations

String comparisons are done in case sensitive way using an Ordinal comparison. The '=' operator is NOT supported for string comparisons since the preprocessor does not know what the setting for "SetExact() is that you want to use. All comparisons are done with String.Compare().

Dialects

In the FoxPro dialect the operators NOT, AND, OR and XOR are also available

1.8.7.7   #ifdef

Purpose

When the X# compiler encounters an #ifdef directive, followed eventually by an #endif directive, it will compile the code between the directives only if the specified symbol is defined.
The #ifdef statement in X# is Boolean and only tests whether the symbol has been defined or not. For example,

```
// DEBUG may be defined from the command line
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#else
    Console.WriteLine("Release version");
#endif
```

Note

The /vo8 compiler option will change this behavior:
when /vo8 is active then any symbol defined with TRUE or a non-0 numeric value will be seen as "defined", symbols defined with FALSE or a 0 numeric value will be seen as "undefined".

See also

X#

1.8.7.8   #ifndef

Purpose

When the X# compiler encounters an #ifndef directive, followed eventually by an #endif directive, it will compile the code between the directives only if the specified symbol is NOT defined.
The #ifndef statement in X# is Boolean and only tests whether the symbol has been defined or not. For example,

```
// DEBUG may be defined from the command line
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#else
    Console.WriteLine("Release version");
#endif
```

Note

when /vo8 is active then any symbol defined with FALSE or a 0 numeric value will also be seen as "undefined".

1.8.7.9   #include

Purpose

Tells the preprocessor to treat the contents of a specified file as if they appear in the source program at the point where the directive appears.

```
#include "path-spec"
```

You can preprocessor definitions into include files and then use #include directives to add them to any source file.
The path-spec is a file name that may optionally be preceded by a directory specification. The file name must name an existing file.
The preprocessor searches for include files in this order:
1. In the same directory as the file that contains the #include statement.
2. Along the path that's specified by each /I compiler option.
3. In the Include folder inside the XSharp installation folder
4. Along the paths that are specified by the INCLUDE environment variable. (Not supported yet)

1.8.7.10 #line

Purpose

#line sets the current line number for the compiler. It is usually inserted by the preprocessor after preprocessing an include file to synchronize the line numbers with the original source file.

```
#line <number> [FileName]
```

Argumes

line number	the new line number to use by the compiler.
FileName	An optional new filename to use by the compiler

1.8.7.11 #pragma options

Purpose

The #pragma options directive allow you to enable / disable certain compiler options for a range of code.

Syntax

```
#pragma options( "option", state )
```

Note: #pragma directives must appear between before the first entity or between entities and cannot appear in the middle of an entity

"option" can be one of the following (please note that not all compiler options can be used)

Option	Description
"az"	Zero based arrays
"allowdot"	Allow Dot for Instance Members
"enforce override"	Enforce the use of the OVERRIDE keyword
"enforce self"	Enforce SELF to access fields / properties / methods

"fovf"	Overflow checking (duplicate of "ovf")
"initlocals"	Initialize local variables
"lb"	Allow late binding
"memvars"	Allow dyamic memory variables (PUBLIC, PRIVATE, MEMVA
"named args"	**This option cannot be set with the #pragma keyword**
"ovf"	Overflow checking (duplicate of "fovf")
"undeclared"	Allow undeclared variables
"vo1"	**This option cannot be set with the #pragma keyword**
"vo2"	Initialize string variables with empty strings
"vo3"	All instance members virtual
"vo4"	Implicit Signed / Unsigned conversions
"vo5"	Implicit Clipper Calling convention
"vo6"	Implicit pointer conversions
"vo7"	Implicit Casts and Conversions
"vo8"	**This option cannot be set with the #pragma keyword**
"vo9"	Allow missing return statements or missing return values
"vo10"	Compatible IIF Behavior
"vo11"	Compatible numeric conversions
"vo12"	Clipper Integer divisions
"vo13"	Compatible String Comparisons
"vo14"	Embed real constants as float
"vo15"	Allow untyped locals
"vo16"	Generate Clipper Calling Convention default constructors
"vo17"	Clipper/VO Compatible Begin Sequence .. End Sequence
"fox1"	**This option cannot be set with the #pragma keyword**
"fox2"	Foxpro array syntax
"xpp1"	**This option cannot be set with the #pragma keyword**

State can be one of the following
- on
- off
- default

Please note that you can also enable / disable overflow checking with BEGIN CHECKED .. END / BEGIN UNCHECKED .. END

1.8.7.12 #pragma warning(s)

Purpose

The #pragma warning directive allow you to suppress certain compiler warnings for a piece of code.
We support both C# style pragma warnings commands (fully) and Vulcan style pragmas (partially)

Syntax

```
#pragma warnings( number, state )
#pragma warnings ( pop )
#pragma warning state2 [<errornumbers>]
```

Note

#pragma directives must appear between before the first entity or between entities and cannot appear in the middle of an entity

Arguments

number	Warning number to disable. Can be both numeric or in the form of XSnnnn
state	Off | Default Disables a warning or switches it back to the situation from the command line
pop	Switchs all warnings back to their default value
state2	Disable | Restore Disables or restores the warning numbers that follow. When no error numbers are specified, then disable disables all warning and restore restores all warning to their default value
errornumbers	(Optional) comma separated list of numbers or names (XSnnnn)

Note:

The compiler does NOT check if the numbers are valid or if they are indeed warnings. So you can specify non existent numbers and/or numbers that represent errors in stead of warnings. The compiler will not warn you when that is the case.

C# style syntax

Example	Description
#pragma warning disable 1234	Disable warning 1234
#pragma warning disable 1234, XS2345	Disable 2 warnings 1234 and XS2345
#pragma warning restore 1234	Reset warning 1234 to the state from the command line
#pragma warning restore 1234, XS2345	Reset 2 warnings 1234 and XS2345 to the state from the command line
#pragma warning disable	Disables all warnings
#pragma warning restore	Restores all warnings to the settings from the command line

Vulcan style syntax

Example	Description
#pragma warnings (1234, off)	Disable warning 1234
#pragma warnings (1234, default)	Reset warning 1234 to the state from the command line
#pragma warnings (pop)	Restores all warnings to the settings from the command line

NOT supported:

#pragma warnings (1234, on)	
#pragma warnings (push)	

1.8.7.13 #region - #endregion

Purpose

#region lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor. In longer code files, it is convenient to be able to collapse or hide one or more regions so that you can focus on the part of the file that you are currently working on. The following example shows how to define a region:

```
#region MyClass definition
CLASS MyClass
    EXPORT Name as STRING
END CLASS
#endregion
```

Comments

A #region block must be terminated with a #endregion directive.
A #region block cannot overlap with a [#ifdef](#) block or an [#ifndef](#) block. However, a #region block can be nested in a #ifdef or #ifndef block, and a #ifdef or #ifndef block can be nested in a #region block.
For now the compiler does not enforce this, but it will in the not too far future.
A #region and #endregion block may have optional comments after the #region and #endregion keyword. The compiler will ignore everything after the keyword until the end of the line

1.8.7.14 **#stdout**

Purpose

Send data to the output stream during compilation

Syntax

```
#stdout <message>
```

Description

The `#stdout` directive sends output to the standard output during compilation followed by a CRLF.
If this output will be shown or not in the IDE that you are using depends on the settings of this IDE.

Example

```
// example of #stdout
// The example demonstrates the use of #stdout
FUNCTION Start AS VOID
#ifndef DEBUG  // output at compile time
```

```
        #stdout Compiling a debug version of the program
    #endif
    ? "Hello world"
    RETURN
```

1.8.7.15 #text

Purpose

Mark the start of a #text .. #endtext region.
The #text directive also defines the nature of the region. This region can either assign a value to a local variable or process the contents of the region.
There are 2 variations of the #text directibe

```
1. #text [:= | +=] VarName [, LineDelimiter [, LineFunc, [,
EndFunc]] ]
2. #text LineFunc [, EndFunc]
```

1. This variation can be recognized by the := or += operator that follows the #text directive. This declares a #text .. #endtext region that stores the value to a local variable for which the name is specified behind the operator. The text declaration may also contain (optional) tokens that will be used as "line delimiters", an optional function that will be used to calculate each line and an optional function name that will be called from the #endtext line
2. This variation does not have a variable name and declares up to 2 function names for each line and the #endtext line

Arguments

VarName	is the Name of the variable that should be assigned the value of the Text block
LineDelimiter	is the delimiter that should be added to the end of every line in the block

LineFunc   is the Name of a function that should be called on every line
                                  in the block. The function gets passed the line and should
                                  return a string

EndFunc   is the Name of a function that gets called after all the lines
                                  were created. This function receives the string value of the
                                  block when the block is assigned to a variable and should
                                  return a string. When the block is not assigned to a variable
                                  then this function gets called without parameters.

Example 1

Please note that the 3 UDCs below are already defined in XSharpDefs.xh

```
#xcommand ENDTEXT => #endtext

#xcommand  TEXT TO <varname>  ;
      =>  #text := <varname>, chr(13)+chr(10)

#xcommand  TEXT TO <varname> ADDITIVE ;
      =>  #text += <varname>, chr(13)+chr(10)



LOCAL cResult AS STRING
TEXT TO cResult
line 1
line 2
line 3
line 4
TEXT
? cResult
```

This code is converted to

```
LOCAL cResult AS STRING
var tempLocal := System.Text.StringBuilder{}
tempLocal:Append("Line 1"+chr(13)+chr(10) )
tempLocal:Append("Line 2"+chr(13)+chr(10) )
tempLocal:Append("Line 3"+chr(13)+chr(10) )
cResult := tempLocal:ToString()
? cResult
```

Please note that the compiler declares different TEXT commands for different dialects.
The TEXT command above works in all dialects.
The TEXT command below is for the Non-Core dialects (with the exception of FoxPro):

```
#xcommand  TEXT INTO <varname> WRAP [<lnbreak>] TRIMMED ;
      => #text := <varname>, iif(<.lnbreak.>,<!lnbreak!>, CRLF) ,
LTrim
```

This command allows a user defined end of line character and calls the LTrim() function on each string before assigning it to the variable

FoxPro declares a special TEXT command that looks like this:

```
#xcommand  TEXT TO <varname> [<tm:TEXTMERGE>] [<noshow:NOSHOW>]
[FLAGS <flags>] [PRETEXT <expression> ]       ;
      => __TextInit(<.tm.>, <.noshow.>, iif(<.flags.>, <!flags!>,
0), <!expression!>  ) ;;
         #text := <varname>,  ,__TextWriteLine , __TextEnd
```

As you can see the command gets translated into a function call to __TextInit() with the values of the various TEXT command options. Each line is send to the __TextWriteLine function and the #endtext directive is replaced with a call to __TextEnd(). There are NO delimiters added for each line. This is handled inside __TextWriteLine. This function is also responsible for expanding expressions inside the text when the TEXTMERGE option is chosen, or when the global SET TEXTMERGE is enabled.

Example 2

Please note that the 2 UDCs below are already defined in XSharpDefs.xh

```
#xcommand ENDTEXT => #endtext

#xcommand  TEXT TO FILE <(file)> ;
      => _TextSave( <(file)> ) ;;
         #text QOut, _TextRestore

TEXT TO FILE EXAMPLE.TXT
line 1
line 2
line 3
line 4
ENDTEXT
```

The TEXT TO FILE command is translated into a call to the function _TextSave(), followed by the #text directive, that specifies that each line must be sent to the QOut() function and that also declares that the #endtext line must be replaced by a call to the _TextRestore() function. The QOut() and _TextRestore function names are specified without parameters. Each line in the block will be sent to the QOut() function as parameter.

So this code is converted to

```
_TextSave("EXAMPLE.TXT");
QOut("line 1")
QOut("line 2")
QOut("line 3")
QOut("line 4")
_TextRestore()
```

See also

TEXT command
Core TEXT Command
Non-Core TEXT Command
FoxPro TEXT Command
#endtext Directive

1.8.7.16 #translate

Purpose

Specify a user-defined translation directive

Syntax

```
#translate    <matchPattern> => <resultPattern> // can be
abbreviated to 4 characters
#xtranslate   <matchPattern> => <resultPattern> // cannot be
abbreviated to 4 characters
#ytranslate   <matchPattern> => <resultPattern> // cannot be
abbreviated to 4 characters and must match case
```

Arguments

`<matchPattern>` is the pattern the input text should match.

`<resultPattern>` is the text produced if a portion of input text matches the
`<matchPattern>`.

The => symbol between <matchPattern> and <resultPattern> is, along with #command
or #translate, a literal part of the syntax that must be specified in a #command or
#translate directive. The symbol consists of an equal sign followed by a greater than
symbol with no intervening spaces. Do not confuse the symbol with the >= or the <=
comparison operators in the xBase language.

Description

#command and #translate are translation directives that define commands and pseudofunctions. Each directive specifies a translation rule. The rule consists of two portions: a match pattern and a result pattern.

The match pattern matches a command specified in the program (.prg) file and saves portions of the command text (usually command arguments) for the result pattern to use. The result pattern then defines what will be written to the result text and how it will be written using the saved portions of the matching input text.

#command and #translate are similar, but differ in the circumstance under which their match patterns match input text. A #command directive matches only if the input text is a complete statement, while #translate matches input text that is not a complete statement. #command defines a complete command and #translate defines clauses and pseudofunctions that may not form a complete statement. In general, use #command for most definitions and #translate for special cases.

#command and #translate are similar to but more powerful than the #define directive. #define, generally, defines identifiers that control conditional compilation and manifest constants for commonly used constant values such as SDK codes. Refer to any of the header files in the INCLUDE directory for examples of manifest constants defined using #define.

#command and #translate directives have the same scope as the #define directive. The definition is valid only for the current program (.prg) file unless defined in Std.ch or the header specified with the /U option on the compiler command line. If defined elsewhere, the definition is valid from the line where it is specified to the end of the program file. Unlike #define, a #translate or #command definition cannot be explicitly undefined. The #undef directive has no effect on a #command or #translate definition.

As the preprocessor encounters each source line preprocessor, it scans for definitions in the following order of precedence: #define, #translate, and #command. When there is a match, the substitution is made to the result text and the entire line is reprocessed until there are no matches for any of the three types of definitions. #command and #translate rules are processed in stack-order (i.e., last in-first out, with the most recently specified rule processed first).

In general, a command definition provides a way to specify an English language statement that is, in fact, a complicated expression or function call, thereby improving the readability of source code. You can use a command in place of an expression or function call to impose order of keywords, required arguments, combinations of arguments that must be specified together, and mutually exclusive arguments at compile time rather than at runtime. This can be important since procedures and user-defined functions can now be called with any number of arguments, forcing any argument checking to occur at runtime. With command definitions, the preprocessor handles some of this.

Because directives are processed in stack order, when defining more than one rule for a command, place the most general case first, followed by the more specific ones. This ensures that the proper rule will match the command specified in the program (.prg) file.

Match Markers

See the topic about Match Markers for a detailed discussion

Result Markers

[See the topic about Result Markers for a detailed discussion](#)

Notes

- **Less than operator**: If you specify the less than operator (<) in the <resultPattern> expression, you must precede it with the escape character (\).
- **Multistatement lines**: You can specify more than one statement as a part of the result pattern by separating each statement with a semicolon. If you specify adjacent statements on two separate lines, the first statement must be followed by two semicolons.

1.8.7.17 #undef

Purpose

Syntax

```
#undef identifier
```

#undef lets you undefine a symbol, such that, by using the symbol as the expression in a [#ifdef](#) directive, the expression will evaluate to false or in a or [#ifndef](#) directive the expression will evaluate to true.

1.8.7.18 Match Markers

Match Pattern

The <matchPattern> portion of a translation directive is the pattern the input text must match. A match pattern is made from one or more of the following components, which the preprocessor tries to match against input text in a specific way:

- Literal values are actual characters that appear in the match pattern. These characters must appear in the input text, exactly as specified to activate the translation directive.

- Words are keywords and valid identifiers that are compared according to the dBASE convention (case-insensitive, first four letters mandatory, etc.). The match pattern must start with a Word.

- #xcommand and #xtranslate can recognize keywords of more than four significant letters.

- Match markers are label and optional symbols delimited by angle brackets (<>) that provide a substitute (idMarker) to be used in the <resultPattern> and identify the clause

for which it is a substitute. Marker names are identifiers and must, therefore, follow the xBase identifier naming conventions. In short, the name must start with an alphabetic or underscore character, which may be followed by alphanumeric or underscore characters.

This table describes all match marker forms:

Match Marker	Name
<idMarker>	Regular match marker
<idMarker,...>	List match marker
<idMarker:word list>	Restricted match marker
<*idMarker*>	Wild match marker
<(idMarker)>	Extended Expression match marker
<#idMarker>	Single match marker.
<%idMarker%>	Wildcard match marker

- **Regular match marker**: Matches the next legal expression in the input text. The regular match marker, a simple label, is the most general and, therefore, the most likely match marker to use for a command argument. Because of its generality, it is used with the regular result marker, all of the stringify result markers, and the blockify result marker.

- **List match marker**: Matches a comma-separated list of legal expressions. If no input text matches the match marker, the specified marker name contains nothing. You must take care in making list specifications because extra commas will cause unpredictable and unexpected results.

  The list match marker defines command clauses that have lists as arguments. Typically these are FIELDS clauses or expression lists used by database commands. When there is a match for a list match marker, the list is usually written to the result text using either the normal or smart stringify result marker. Often, lists are written as literal arrays by enclosing the result marker in curly ({ }) braces.

- **Restricted match marker**: Matches input text to one of the words in a comma-separated list. If the input text does not match at least one of the words, the match fails and the marker name contains nothing.

  A restricted match marker is generally used with the logify result marker to write a logical value into the result text. If there is a match for the restricted match marker, the corresponding logify result marker writes true (.T.) to the result text; otherwise, it writes false (.F.). This is particularly useful when defining optional clauses that consist of a command keyword with no accompanying argument. Std.ch implements the REST clause of database commands using this form.

- **Wild match marker**: Matches any input text from the current position to the end of a statement. Wild match markers generally match input that may not be a legal expression, such as
  #command NOTE <*x*> in Clippers Std.ch, gather the input text to the end of thestatement, and write it to the result text using one of the stringify result markers.

- **Single match marker.** Matches all consecutive tokens until a whitespace token.

- **Wildcard match marker**: Matches a list of tokens that are an Id, Comma, ? or *. This can be used in commands such as SAVE ALL LIKE a*,b*

- **Extended expression match marker**: Matches a regular orextended expression, including a file name or path specification. It is used with the smart stringify result marker to ensure that extended expressions will not get stringified, while normal, unquoted string file specifications will.

- **Optional match clauses** are portions of the match pattern enclosed in square brackets ([ ]). They specify a portion of the match pattern that may be absent from the input text. An optional clause may contain any of the components allowed within a <matchPattern>, including other optional clauses.

  Optional match clauses may appear anywhere and in any order in the match pattern and still match input text. Each match clause may appear only once in the input text. There are two types of optional match clauses: one is a keyword followed by match marker, and the other is a keyword by itself. These two types of optional match clauses can match all of the traditional command clauses typical ofthe xBase command set.

  Optional match clauses are defined with a regular or list match marker to match input text if the clause consists of an argument or a keyword followed by an argument (see the INDEX clause of the USE command in Std.ch). If the optional match clause consists of a keyword by itself, it is matched with a restricted match marker (see the EXCLUSIVE or SHARED clause of the USE command in Std.ch).

  In any match pattern, you may not specify adjacent optional match clauses consisting solely of match markers, without generating a compiler error. You may repeat an optional clause any number of times in the input text, as long as it is not adjacent to any other optional clause. To write a repeated match clause to the result text, use repeating result clauses in the <resultPattern> definition.

1.8.7.19   Result Markers

Result Pattern

The <resultPattern> portion of a translation directive is the text the preprocessor will produce if a piece of input text matches the <matchPattern>. <resultPattern> is made from one or more of the following components:

- **Literal tokens** are actual characters that are written directly to the result text.

- **Words** are xBase keywords and identifiers that are written directly to the result text.

- **Result markers**: refer directly to a match marker name. Input text matched by the match marker is written to the result text via the result marker.

This table lists the Result marker forms:

Result Marker	Name
\<idMarker\>	Regular result marker
#\<idMarker\>	Dumb stringify result marker
\<"idMarker"\>	Normal stringify result marker
\<(idMarker)\>	Smart stringify result marker
\<{idMarker}\>	Blockify result marker
\<.idMarker.\>	Logify result marker
\<!idmarker!\>	Notempty result marker

- **Regular result marker:** Writes the matched input text to the result text, or nothing if no input text is matched. Use this, the most general result marker, unless you have special requirements. You can use it with any of the match markers, but it almost always is used with the regular match marker.

- **Dumb stringify result marker**: Stringifies the matched input text and writes it to the result text. If no input text is matched, it writes a null string (""). If the matched input text is a list matched by a list match marker, this result marker stringifies the entire list and writes it to the result text.

  This result marker writes output to result text where a string is always required. This is generally the case for commands where a command or clause argument is specified as a literal value but the result text must always be written as a string even if the argument is not specified.

- **Normal stringify result marker**: Stringifies the matched input text and writes it to the result text. If no input text is matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker stringifies each element in the list and writes it to the result text.

  The normal stringify result marker is most often used with the blockify result marker to compile an expression while saving a text image of the expression (See the SET FILTER condition and the INDEX key expression in Std.ch).

- **Smart stringify result marker**: Stringifies matched input text only if source text is enclosed in parentheses. If no input text matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker stringifies each element in the list (using the same stringify rule) and writes it to the result text.

  The smart stringify result marker is designed specifically to support extended expressions for commands other than SETs with \<xlToggle\> arguments. Extended expressions are command syntax elements that can be specified as literal text or as an expression if enclosed in parentheses. The \<xcDatabase\> argument of the USE command is a typical example. For instance, if the matched input for the \<xcDatabase\> argument is the word Customer, it is written to the result text as the string "Customer," but the expression (cPath + cDatafile) would be written to the result text unchanged (i.e., without quotes).

- **Blockify result marker**: Writes matched input text as a code block without any arguments to the result text. For example, the input text x + 3 would be written to the result text as {|| x + 3}. If no input text is matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker blockifies each element in the list.

  The blockify result marker used with the regular and list match markers matches various kinds of expressions and writes them as code blocks to the result text. Remember that a code block is a piece of compiled code to execute sometime later. This is important when defining commands that evaluate expressions more than once per invocation. When defining a command, you can use code blocks to pass an expression to a function and procedure as data rather than as the result of an evaluation. This allows the target routine to evaluate the expression whenever necessary.

  In Std.ch, the blockify result marker defines database commands where an expression is evaluated for each record. Commonly, these are field or expression lists, FOR and WHILE conditions, or key expressions for commands that perform actions based on key values.

- **Logify result marker**: Writes true (.T.) to the result text if any input text is matched; otherwise, it writes false (.F.) to the result text. This result marker does not write the input text itself to the result text.

  The logify result marker is generally used with the restricted match marker to write true (.T.) to the result text if an optional clause is specified with no argument; otherwise, it writes false (.F.). In Std.ch, this formulation defines the EXCLUSIVE and SHARED clauses of the USE command.

- **Notempty result marker.** Writes the matched input text to the result text, or NIL if no input text is matched. This may be required instead of the regular result marker if you place the marker inside an IIF() expression.

- **Repeating result clauses** are portions of the <resultPattern> enclosed by square brackets ([ ]). The text within a repeating clause is written to the result text as many times as it has input text for any or all result markers within the clause. If there is no matching input text, the repeating clause is not written to the result text. Repeating clauses, however, cannot be nested. If you need to nest repeating clauses, you probably need an additional
  #command rule for the current command.

  Repeating clauses are the result pattern part of the #command facility that create optional clauses which have arguments. You can match input text with any match marker other than the restricted match marker and write to the result text with any of the corresponding result markers. Typical examples of this facility are the definitions for the STORE and REPLACE commands in Std.ch.

1.9   X# Compiler Options

The X# compiler is a command line tool named xsc.exe. This tool can be controlled using command line options. The following pages describe these command line options.

1.9.1   Command-line Building With xsc.exe

You can invoke the X# compiler by typing the name of its executable file (xsc.exe) at a command prompt.

If you use a standard Command Prompt window, you must adjust your path before you can invoke xsc.exe from any subdirectory on your computer.

You can also use MSBuild to build X# programs programmatically. For more information, see the MSBuild documentation on MSDN.

The xsc.exe executable file usually is located in the {Program Files}\XSharp\Bin folder under the Windows directory.

When you build programs with the Visual Studio IDE then Visual Studio will locate the X# compiler automatically for you and the build output from the compiler will be shown in the Output Windows of Visual Studio.
You can set the verbosity level of the compiler output from the Tools/Options, Projects and Solutions, Build and Run page.

Rules for Command-Line Syntax for the X# Compiler

The X# compiler uses the following rules when it interprets arguments given on the operating system command line:

- Arguments are delimited by white space, which is either a space or a tab.
- The caret character (^) is not recognized as an escape character or delimiter. The character is handled by the command-line parser in the operating system before it is passed to the argv array in the program.
- A string enclosed in double quotation marks ("string") is interpreted as a single argument, regardless of white space that is contained within. A quoted string can be embedded in an argument.
- A double quotation mark preceded by a backslash (\") is interpreted as a literal double quotation mark character (").
- Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
- If an even number of backslashes is followed by a double quotation mark, one backslash is put in the compiler options array for every pair of backslashes, and the double quotation mark is interpreted as a string delimiter.
- If an odd number of backslashes is followed by a double quotation mark, one backslash is put in the compiler options array for every pair of backslashes, and the double quotation mark is "escaped" by the remaining backslash. This causes a literal double quotation mark (") to be added in compiler options array.
- Commandline options can start with a hyphen (-) or a slash (/). On Non windows platforms the hyphen should be used because the slash may be seen as a path delimiter character
- If you do not use the /noconfig compiler option then references to the assemblies listed in xsc.exe will be automatically included.

Sample commands for the X# Compiler

xsc file.prg	Compile File.prg and produce File.exe
xsc -target:library file.prg	Compile File.prg and produce File.dll
xsc -out:Program.exe file.prg	Compile File.prg and produce Program.exe
xsc -define:DEBUG -optimize -out:File.exe *.prg	Compiles all the X# files in the current directory, with optimizations on and defines the DEBUG symbol. The output is File2.exe
xsc /target:library /out:File2.dll /warn:0 /nologo / debug *.prg	Compiles all the X# files in the current directory producing a debug version of File2.dll. No logo and no warnings are displayed
xsc /target:library /out:MyBase.rdd *.prg	Compiles all the X# files in the current directory to MyBase.rdd (a DLL):

1.9.2 X# Compiler Options By Category

The following compiler options are sorted by category. For an alphabetical list, see : X# Compiler Options Listed Alphabetically

Dialect support

Option	Purpose
-dialect	Specifies the dialect to use when compiling. The following values are supported: Core, VO, Vulcan, dBase, FoxPro, Xbase++ and Harbour. Work on the VO and Vulcan dialects has started.

XBase Compatibility

Option	Purpose
-allowdot	Allow the DOT (.) character as operator for instance members
-allowoldstyleassignments	Allow assignments with a single Equals operator (=)
-az	Specifies that arrays are zero-based rather than one-based
-cs	Specifies that the compiler should use case-sensitive type names
-d	Instruct the compiler to emit debugging information. (short for /debug)
-enforceoverride	Enforce the use of the OVERRIDE keyword to override methods from parent classes in subclasses
-enforceself	Enforce the use of SELF: to access fields, properties and methods inside a class
-fovf	Specifies that exceptions will be raised on invalid floating point operations
-fox1	Classes are assumed to inherit from the Custom class
-fox2	Fompatible FoxPro Array support
-i	Specifies a directory to add to the #include file search path
-initlocals	Initialize all local variables
-ins	Use implicit namespace lookup mechanism
-lb	Specifies that the compiler should generate late bound calls when necessary

Option	Purpose
-memvar	Enables support for memory variables
-modernsyntax	Disable certain legacy compiler options
-namedarguments	Specifies whether to allow named arguments in the parser or not.
-noinit	Suppress generation of empty $Init1() and $Exit() functions
-norun	Obsolete compiler option in X#, inherited from Vulcan. To achieve this simply remove the references to the runtime DLLs and compile in the Core dialect
-nostddefs	Suppresses preprocessor definitions in XSharpDefs.xh
-ns	Specify the default namespace for the output assembly
-ovf	Specifies that exceptions will be raised on integer overflows
-showincludes	Lists #include files in compiler output
-snk	Signs assembly with strong name key pair
-undeclared	Enables support for undeclared variables
-vo1	Use Init and Axit methods in stead of Constructors and Destructors
-vo2	Initialize STRING variables, fields and DIM arrays to empty strings
-vo3	Treats All Methods As VIRTUAL
-vo4	Allows implicit signed/unsigned integer conversions
-vo5	Implicit CLIPPER Calling Convention for Zero-Argument Functions
-vo6	Resolves typed function pointers to PTR
-vo7	Allows compatible implicit casts and conversions
-vo8	Enables compatible preprocessor behavior
-vo9	Allows missing RETURN statements
-vo10	Enables compatible IIF behavior
-vo11	Enables Visual Objects compatible arithmetic conversions
-vo12	Enables Clipper compatible integer divisions
-vo13	Enables Visual Objects compatible string comparisons
-vo14	Insert floating point literals as FLOAT

Option	Purpose
-vo15	Allow untyped Locals and return types
-vo16	Automatically generate Clipper calling convention constructors for classes without constructor
-w	This option is not supported. use -NoWarn in stead
-wx	Treats all warnings as errors (alias for -warnaserror)
-xpp1	All classes inherit from the Abstract class

Optimization

Option	Purpose
-filealign	Specifies the size of sections in the output file.
-optimize	Enables/disables optimizations.

Output files

Option	Purpose
-doc	Specifies an XML file where processed documentation comments are to be written.
-modulename	Specify the name of the source module
-out	Specifies the output file.
-pathmap	Specify a mapping for source path names output by the compiler.
-pdb	Specifies the file name and location of the .pdb file.
-platform	Specify the output platform.
-preferreduilang	Specify a language for compiler output.
-target	Specifies the format of the output file using one of five options: /target:appcontainerexe, /target:exe, /target:library, /target:module, /target:winexe, or /target:winmdobj.
-touchedfiles	Specify filename that will be updated with list of files read and written by the compiler

.NET Assemblies

Option	Purpose
-analyzer	Run the analyzers from this assembly (Short form: /a)
-additionalfile	Names additional files that don't directly affect code generation but may be used by analyzers for producing errors or warnings.
-addmodule	Specifies one or more modules to be part of this assembly.
-delaysign	Instructs the compiler to add the public key but to leave the assembly unsigned.
-keycontainer	Specifies the name of the cryptographic key container.
-keyfile	Specifies the filename containing the cryptographic key.
-lib	Specifies the location of assemblies referenced by means of /reference.
-nostdlib	Instructs the compiler not to import the standard library (mscorlib.dll).
-reference	Imports metadata from a file that contains an assembly.

Debugging / Error checking

Option	Purpose
-ast	Dump the abstract syntax tree for each source file (.ast extension)
-checked	Specifies whether integer arithmetic that overflows the bounds of the data type will cause an exception at run time.
-debug	Instruct the compiler to emit debugging information.
-errorendlocation	Output line and column of the end location of each error
-errorreport	Sets error reporting behavior.
-fullpaths	Specifies the absolute path to the file in compiler output.
-lexonly	Tells the compiler to only lex the source code.
-nowarn	Suppresses the compiler's generation of specified warnings.
-parseonly	This compiler option tells the compiler to lex and parse the code.
-ruleset	Specify a ruleset file that disables specific diagnostics.
-warn	Sets the warning level.

Option	Purpose
-warnaserror	Promotes warnings to errors.
-wx	This is an alias for the -warnaserror option

Preprocessor

Option	Purpose
-define	Defines preprocessor symbols.
-i	Specifies a directory to add to the #include file search path
-ppo	Writes preprocessor output to file
-nostddef	Suppresses preprocessor definitions in XSharpDefs.xh
-showdefs	Show the defines that are added from the header files and their usage
-showincludes	Lists #include files in compiler output
-verbose	Shows includes, source file names, defines and more. on the console

Resources

Option	Purpose
-link	Makes COM type information in specified assemblies available to the project.
-linkresource	Creates a link to a managed resource.
-resource	Embeds a .NET Framework resource into the output file.
-usenativeversion	Prefer native resource (if any) over resources generated from managed assembly properties
-win32icon	Specifies an .ico file to insert into the output file.
-win32res	Specifies a Win32 resource to insert into the output file.

Miscellaneous

X#

Option	Purpose
@	Specifies a response file.
-?	Lists compiler options to stdout.
-appconfig	Specify an application configuration file containing assembly binding settings
-baseaddress	Specifies the preferred base address at which to load a DLL.
-checksumalgorithm	Specify the algorithm for calculating the source file checksum stored in PDB. Supported values are: SHA1 (default) or SHA256.
-codepage	Specifies the code page to use for all source code files in the compilation.
-help	Lists compiler options to stdout.
-highentropyva	Specifies that the executable file supports address space layout randomization (ASLR).
-langversion	Specify language version mode: core, VO, Vulcan, Harbour, FoxPro, dBase
-main	Specifies the location of the Main method.
-noconfig	Instructs the compiler not to compile with xsc.rsp.
-nologo	Suppresses compiler banner information.
-parallel	Specifies whether to use concurrent build (+).
-recurse	Searches subdirectories for source files to compile.
-s	Syntax check only
-shared	Use the shared compiler
-subsystemversion	Specifies the minimum version of the subsystem that the executable file can use.
-unsafe	Enables compilation of code that uses the unsafe keyword.
-utf8output	Displays compiler output using UTF-8 encoding.
-win32manifest	Specify a user-defined Win32 application manifest file to be embedded into a executable file.

1.9.3   X# Compiler Options Listed Alphabetically

The following compiler options are sorted by category. For an alphabetical list, see : X#
Compiler Options Listed Alphabetically
The options in **RED** are not supported yet.

Option	Purpose
@	Specifies a response file.
-?	Lists compiler options to stdout.
-additionalfile	Names additional files that don't directly affect code generation but may be used by analyzers for producing errors or warnings.
-addmodule	Specifies one or more modules to be part of this assembly.
-allowdot	Controls if the DOT (".") operator should be allowed to access instance members
-allowoldstyleassignments	Allow assignments with a single Equals operator (=)
-analyzer	Run the analyzers from this assembly (Short form: /a)
-appconfig	Specify an application configuration file containing assembly binding settings
-ast	Dump the abstract syntax tree for each source file (.ast extension)
-az	Specifies that arrays are zero-based rather than one-based
-baseaddress	Specifies the preferred base address at which to load a DLL.
-checked	Specifies whether integer arithmetic that overflows the bounds of the data type will cause an exception at run time.
-checksumalgorithm	Specify the algorithm for calculating the source file checksum stored in PDB. Supported values are: SHA1 (default) or SHA256.
-codepage	Specifies the code page to use for all source code files in the compilation.
-cs	Specifies that the compiler should use case-sensitive type names
-debug -d	Instruct the compiler to emit debugging information.
-define	Defines preprocessor symbols.
-dialect	Specifies the dialect to use when compiling. The following values are supported: Core, VO, Vulcan, dBase, FoxPro,

Option	Purpose
	Xbase++ and Harbour. Work on the VO and Vulcan dialects has started.
-delaysign	Instructs the compiler to add the public key but to leave the assembly unsigned.
-doc	Specifies an XML file where processed documentation comments are to be written.
-enforceoverride	Enforce the use of the OVERRIDE keyword to override methods from parent classes in subclasses
-enforceself	Enforce the use of SELF: to access fields, properties and methods inside a class
-errorendlocation	Output line and column of the end location of each error
-errorreport	Sets error reporting behavior.
-filealign	Specifies the size of sections in the output file.
-fox1	Assume classes inherit from Custom class. This also controls the code generation for properties.
-fox2	Compatible FoxPro Array support
-fovf	Specifies that exceptions will be raised on invalid floating point operations (duplicate of -ovf)
-fullpaths	Specifies the absolute path to the file in compiler output.
-help	Lists compiler options to stdout.
-highentropyva	Specifies that the executable file supports address space layout randomization (ASLR).
-i	Specifies a directory to add to the #include file search path
-initlocals	Initialize all local variables
-ins	Use implicit namespace lookup mechanism
-keycontainer	Specifies the name of the cryptographic key container.
-keyfile	Specifies the filename containing the cryptographic key.
-lb	Specifies that the compiler should generate late bound calls when necessary
-langversion	Specify language version mode: core, VO, Vulcan, Harbour, FoxPro, dBase
-lexonly	Tells the compiler to only lex the source code.
-lib	Specifies the location of assemblies referenced by means of /reference.

Option	Purpose
-link	Makes COM type information in specified assemblies available to the project.
-linkresource	Creates a link to a managed resource.
-main	Specifies the location of the Main method.
-memvar	Enable support for memory variables
-modernsyntax	Disable certain legacy compiler options
-modulename	Specify the name of the source module
-namedargs	Specifies whether to allow named arguments in the parser or not.
-nostddefs	Suppresses preprocessor definitions in XSharpDefs.xh
-ns	Specify the default namespace for the output assembly
-noconfig	Instructs the compiler not to compile with xsc.rsp.
-noinit	Suppress generation of empty $Init1() and $Exit() functions
-nologo	Suppresses compiler banner information.
-norun	Obsolete compiler option in X#, inherited from Vulcan. To achieve this simply remove the references to the runtime DLLs and compile in the Core dialect
-nostdlib	Instructs the compiler not to import the standard library (mscorlib.dll).
-nowarn	Suppresses the compiler's generation of specified warnings.
-optimize	Enables/disables optimizations.
-ovf	Specifies that exceptions will be raised on overflows (duplicate of -fovf)
-out	Specifies the output file.
-parallel	Specifies whether to use concurrent build (+).
-parseonly	Tells the compiler to only lex and parse the source code.
-pathmap	Specify a mapping for source path names output by the compiler.
-pdb	Specifies the file name and location of the .pdb file.
-platform	Specify the output platform.
-ppo	Writes preprocessor output to file
-preferreduilang	Specify a language for compiler output.

X#

Option	Purpose
-recurse	Searches subdirectories for source files to compile.
-reference	Imports metadata from a file that contains an assembly.
-resource	Embeds a .NET Framework resource into the output file.
-ruleset	Specify a ruleset file that disables specific diagnostics.
-s	Syntax check only
-shared	Use the shared compiler
-showdefs	Show the defines that are added from the header files and their usage
-showincludes	Lists #include files in compiler output
-snk	Signs assembly with strong name key pair
-subsystemversion	Specifies the minimum version of the subsystem that the executable file can use.
-target	Specifies the format of the output file using one of five options: /target:appcontainerexe, /target:exe, /target:library, /target:module, /target:winexe, or /target:winmdobj.
-touchedfiles	Specify filename that will be updated with list of files read and written by the compiler
-undeclared	Enables the support for undeclared memory variables
-unsafe	Enables compilation of code that uses the unsafe keyword.
-usenativeversion	Prefer native resource (if any) over resources generated from managed assembly properties
-utf8output	Displays compiler output using UTF-8 encoding.
-vo1	Use Init and Axit methods in stead of Constructors and Destructors
-vo2	Initialize STRING variables, fields and DIM arrays to empty strings
-vo3	Treats All Methods As VIRTUAL
-vo4	Allows implicit numeric conversions
-vo5	Implicit CLIPPER Calling Convention for Zero-Argument Functions
-vo6	Resolves typed function pointers to PTR
-vo7	Allows compatible implicit casts and conversions
-vo8	Enables compatible preprocessor behavior
-vo9	Allows missing RETURN statements

Option	Purpose
-vo10	Enables compatible IIF behavior
-vo11	Enables Visual Objects compatible arithmetic conversions
-vo12	Enables Clipper compatible integer divisions
-vo13	Enables Visual Objects compatible string comparisons
-vo14	Insert floating point literals as FLOAT
-vo15	Allow untyped Locals and return types
-vo16	Automatically generate Clipper calling convention constructors for classes without constructor
-w	This option is not supported. use -nowarn in stead
-warn	Sets the warning level.
-warnaserror	Promotes warnings to errors.
-win32icon	Specifies an .ico file to insert into the output file.
-win32manifest	Specify a user-defined Win32 application manifest file to be embedded into a executable file.
-win32res	Specifies a Win32 resource to insert into the output file.
-wx	Treats all warnings as errors (alias for -warnaserror)
-xpp1	Classes without parent class inherit from the Abstract class

1.9.3.1 @

The @ option lets you specify a file that contains compiler options and source code files to compile.

Syntax

```
@response_file
```

Arguments

response_file   A file that lists compiler options or source code files to compile.

Remarks

The compiler options and source code files will be processed by the compiler just as if they had been specified on the command line.

To specify more than one response file in a compilation, specify multiple response file options. For example:

```
@file1.rsp @file2.rsp
```

In a response file, multiple compiler options and source code files can appear on one line. A single compiler option specification must appear on one line (cannot span multiple lines). Response files can have comments that begin with the # symbol.

Specifying compiler options from within a response file is just like issuing those commands on the command line. See Building from the Command Line for more information.

The compiler processes the command options as they are encountered. Therefore, command line arguments can override previously listed options in response files. Conversely, options in a response file will override options listed previously on the command line or in other response files.

X# provides the xsc.rsp file, which is located in the same directory as the xsc.exe file. See -noconfig for more information on xsc.rsp.

This compiler option cannot be set in the Visual Studio development environment, nor can it be changed programmatically.

Example

The following are a few lines from a sample response file:

```
# build the first output file
-target:exe -out:MyExe.exe source1.prg source2.prg
```

1.9.3.2 -additionalfile

The -additionalfile commandline option allows you to include extra files in the assembly. To support this scenario the X# compiler can accept additional, non-source text files as inputs.

Syntax

```
-additionalfile:file
```

For example, an analyzer may enforce that a set of banned terms is not used within a project, or that every source file has a certain copyright header. The terms or copyright header could be passed to the analyzer as an additional file, rather than being hard-coded in the analyzer itself.

Example

On the command line, additional files can be passed using The -additionalfile option. For example:

```
xsc alpha.prg -additionalfile:terms.txt
```

1.9.3.3 -addmodule

This option adds a module that was created with the target:module switch to the current compilation.

Syntax

```
-addmodule:file[;file2]
```

Arguments

file, file2   An output file that contains metadata. The file cannot contain an assembly manifest. To import more than one file, separate file names with either a comma or a semicolon.

Remarks

All modules added with -addmodule must be in the same directory as the output file at run time. That is, you can specify a module in any directory at compile time but the module must be in the application directory at run time. If the module is not in the application directory at run time, you will get a TypeLoadException.

file cannot contain an assembly. For example, if the output file was created with -target:module, its metadata can be imported with -addmodule.

If the output file was created with a -target option other than -target:module, its metadata cannot be imported with -addmodule but can be imported with -reference.

This compiler option is unavailable in Visual Studio; a project cannot reference a module. In addition, this compiler option cannot be changed programmatically.

Example

Compile source file input.prg and add metadata from metad1.netmodule and metad2.netmodule to produce out.exe:

```
xsc -addmodule:metad1.netmodule;metad2.netmodule -out:out.exe
input.prg
```

1.9.3.4 -allowdot

The -allowdot compiler option allows you to control if the DOT (".") operator should be allowed to access instance members.
- The default for the Core and FoxPro dialects is /allowdot+
- The default for the other dialects is /allowdot-

Syntax

```
-allowdot[+ | -]
```

Arguments

+ | -   Specifying +, or just -

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. [Click here to see the property page](#)

1.9.3.5 -allowoldstyleassignments

The -allowoldstyleassignments compiler option allows you to control if the single Equals ("=") operator should be allowed to assign values.
- The default for the FoxPro dialect is /allowdot+
- The default for the other dialects is /allowdot-

Syntax

```
-allowoldstyleassignments[+ | -]
```

Arguments

+ | -   Specifying +, or just -

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Language tab.
3. Change the value.
4. [Click here to see the property page](#)

1.9.3.6 -analyzer, -a

Run the analyzers from this assembly

1.9.3.7 -appconfig

The -appconfig compiler option enables a X# application to specify the location of an assembly's application configuration (app.config) file to the common language runtime (CLR) at assembly binding time.

Syntax

```
-appconfig:file
```

Arguments

file   Required. The application configuration file that contains assembly binding settings.

Remarks

One use of -appconfig is advanced scenarios in which an assembly has to reference both the .NET Framework version and the .NET Framework for Silverlight version of a particular reference assembly at the same time. For example, a XAML designer written in Windows Presentation Foundation (WPF) might have to reference both the WPF Desktop, for the designer's user interface, and the subset of WPF that is included with Silverlight. The same designer assembly has to access both assemblies. By default, the separate references cause a compiler error, because assembly binding sees the two assemblies as equivalent.

The -appconfig compiler option enables you to specify the location of an app.config file that disables the default behavior by using a `<supportPortability>` tag, as shown in the following example.

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

The compiler passes the location of the file to the CLR's assembly-binding logic.

Note
If you are using the Microsoft Build Engine (MSBuild) to build your application, you can set The -appconfig compiler option by adding a property tag to the .xsproj file. To use the app.config file that is already set in the project, add property tag <UseAppConfigForCompiler> to the .xsproj file and set its value to true. To specify a different app.config file, add property tag <AppConfigForCompiler> and set its value to the location of the file.

Example

The following example shows an app.config file that enables an application to have references to both the .NET Framework implementation and the .NET Framework for Silverlight implementation of any .NET Framework assembly that exists in both implementations. The -appconfig compiler option specifies the location of this app.config file.

```
<configuration>
     <runtime>
       <assemblyBinding>
           <supportPortability PKT="7cec85d7bea7798e"
enable="false"/>
           <supportPortability PKT="31bf3856ad364e35"
enable="false"/>
       </assemblyBinding>
```

```
        </runtime>
    </configuration>
```

1.9.3.8   -ast

Tells the compiler to write abstract syntax trees for each of the source files that was processed.

Syntax

```
-ast[+|-]
```

Arguments

+ | -   Specifying +, or just -ast, tells the compiler to produce
                               abstract syntax trees for each of the source files.

Remarks

This command line option allows you to check the code without generating a binary.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Build tab.
3. Add the option in the "Extra Command Line options" property

1.9.3.9   -az

The -az option specifies that array elements begin with 0, rather than with 1 (the default).

Syntax

```
-az[+|-]
```

Arguments

+ | -   Specifying +, or just -az, directs the compiler to use 0-based array indexing rather than 1-based indexing.

Remarks

This option is off by default, since it would break existing Visual Objects source code. If you prefer to use 0-based arrays with existing code written for Visual Objects, you will need to examine every place in your source code that uses arrays and manually make the appropriate adjustments.

Note: This option does not affect how the assembly being compiled is used from other applications. When The -az option is not used, the compiler generates code to subtract 1 from the array index in order to provide 1-based array indexing semantics at the language level. When The -az option is used, the compiler does not adjust array indexes. Either way, the resulting arrays are always 0-based at the IL level, which allows compatibility with all other .NET languages.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Language tab.
3. In the General section, modify the "Use Zero Based Arrays" property.
4. [Click here to see the property page](#)

Example

```
FUNCTION Start() AS VOID
    LOCAL DIM a[1] AS INT
    ? a[0] -/ runtime error when -az switch is NOT used
    ? a[1] -/ runtime error when -az switch is used
    RETURN
```

1.9.3.10 -baseaddress

The -baseaddress option lets you specify the preferred base address at which to load a DLL.

Syntax

```
-baseaddress:address
```

Arguments

address   The base address for the DLL. This address can be
                                  specified as a decimal, hexadecimal, or octal number.

Remarks

The default base address for a DLL is set by the .NET Framework common language runtime.

Be aware that the lower-order word in this address will be rounded. For example, if you specify 0x11110001, it will be rounded to 0x11110000.

To complete the signing process for a DLL, use SN.EXE with the -R option.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build property page.
3. Add the option in the "User-Defined Command Line options" property

1.9.3.11 -checked

The -checked option specifies whether an integer arithmetic statement that results in a value that is outside the range of the data type, and that is not in the scope of a checked or unchecked keyword, causes a run-time exception.

Syntax

```
-checked[+ | -]
```

Remarks

An integer arithmetic statement that is in the scope of a checked or unchecked keyword is not subject to the effect of The -checked option.

If an integer arithmetic statement that is not in the scope of a checked or unchecked keyword results in a value outside the range of the data type, and -checked+ (/checked) is

used in the compilation, that statement causes an exception at run time. If -checked- is used in the compilation, that statement does not cause an exception at run time.

The default value for this option is -checked-. One scenario for using -checked- is in building large applications. Sometimes automated tools are used to build such applications, and such a tool might automatically set -checked to +. You can override the tool's global default by specifying -checked-.

The VO Compatibility compiler options -ovf and -fovf both set this option

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build property page.
3. Add the option in the "User-Defined Command Line options" property

Example

The following command compiles `t2.prg`. The use of `-checked` in the command specifies that any integer arithmetic statement in the file that is not in the scope of a checked or unchecked keyword, and that results in a value that is outside the range of the data type, causes an exception at run time.

```
xsc t2.prg -checked
```

1.9.3.12 -checksumalgorithm

Specify the algorithm for calculating the source file checksum stored in PDB. Supported values are: SHA1 (default) or SHA256.

Syntax

```
-checksumalgorithm:<alg>
```

Arguments

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build property page.
3. Add the option in the "User-Defined Command Line options" property

1.9.3.13 -codepage

This option specifies which codepage to use during compilation if the required page is not the current default codepage for the system.

Syntax

```
-codepage:id
```

Arguments

id	The id of the code page to use for all source code files in the compilation.

Remarks

If you compile one or more source code files that were not created to use the default code page on your computer, you can use The -codepage option to specify which code page should be used. -codepage applies to all source code files in your compilation.

If the source code files were created with the same codepage that is in effect on your computer or if the source code files were created with UNICODE or UTF-8, you need not use -codepage.

See GetCPInfo for information on how to find which code pages are supported on your system.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

1.9.3.14 -cs

The -cs option directs the compiler to treat all identifiers and type names as case-sensitive.

Syntax

```
-cs[+|-]
```

Arguments

+|-	Specifying +, or just -cs, directs the compiler to treat all type names as case-sensitive.

Remarks

This option is for those users who prefer a case-sensitive language.

This option can also affect case sensitivity of preprocessor symbols, depending on the state of the -vo8 compiler option; if -vo8 is enabled, then -cs controls if also preprocessor symbols are treated as case-sensitive; if -vo8 is disabled, then preprocessor symbols are always case-sensitive and the state of -cs does not affect this behavior of the preprocessor. See the topic about -vo8 for more information.

This option also does not affect keywords. Keywords in X# are always case-insensitive.

Note

If you run the shared compiler you have to pass the -cs compiler option through the commandline and not through a response file.
And when you mix case sensitive and not case sensitive projects then there will be 2 shared compiler processes running, one in case sensitive mode and the other in case insensitive mode.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Language tab.
3. In the General section, modify the Case Sensitive Type Names property in the General.
4. Click here to see the property page

1.9.3.15   -debug

The -debug option causes the compiler to generate debugging information and place it in the output file or files.

Syntax

```
-debug[+ | -]
-debug:{full | pdbonly}
```

Arguments

+ | -	Specifying +, or just -debug, causes the compiler to generate debugging information and place it in a program database (.pdb file). Specifying -, which is in effect if you do not specify -debug, causes no debug information to be created.
full | pdbonly	Specifies the type of debugging information generated by the compiler. The full argument, which is in effect if you do not specify -debug:pdbonly, enables attaching a debugger to the running program. Specifying pdbonly allows source code debugging when the program is started in the debugger but will only display assembler when the running program is attached to the debugger.

Remarks

Use this option to create debug builds. If -debug, -debug+, or -debug:full is not specified, you will not be able to debug the output file of your program.

If you use -debug:full, be aware that there is some impact on the speed and size of JIT optimized code and a small impact on code quality with -debug:full. We recommend -debug:pdbonly or no PDB for generating release code.

Note

One difference between -debug:pdbonly and -debug:full is that with -debug:full the compiler emits a DebuggableAttribute, which is used to tell the JIT compiler that debug information is available. Therefore, you will get an error if your code contains the DebuggableAttribute set to false if you use -debug:full.

For more information on how to configure the debug performance of an application, see Making an Image Easier to Debug.

To change the location of the .pdb file, see -pdb

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build property page.
3. Click the Advanced button.
4. Modify the Debug Info property.

For information on how to set this compiler option programmatically, see DebugSymbols.

Example

Place debugging information in output file `app.pdb`:

```
xsc -debug -pdb:app.pdb test.prg
```

1.9.3.16 -define

The -define compiler option defines name as a symbol in all source code files your program.

Syntax

```
-define:name[;name2]
```

Arguments

name, name2   The name of one or more symbols that you want to define.

Remarks

The -define option has the same effect as using a #define preprocessor directive except that the compiler option is in effect for all files in the project. A symbol remains defined in a source file until an #undef directive in the source file removes the definition. When you use The -define option, an #undef directive in one file has no effect on other source code files in the project.

You can use symbols created by this option with #if, #else, #elif, and #endif to compile source files conditionally.

-d is the short form of -define.

You can define multiple symbols with -define by using a semicolon or comma to separate symbol names. For example:

```
-define:DEBUG;TUESDAY
```

The X# compiler defines a couple of symbols automatically. See the Macros topic elsewhere in this documentation.

To set this compiler option in the Visual Studio development environment
1. Open the project's Properties page.
2. On the Build tab, type the symbol that is to be defined in the Conditional compilation symbols box. For example, if you are using the code example that follows, just type xx into the text box.

For information on how to set this compiler option programmatically, see DefineConstants.

Example

```
-/ preprocessor_define.prg
-/ compile with: -define:xx
-/ or uncomment the next line
-/ #define xx
using System;
public class Test
{
    public static void Main()
    {
        #if (xx)
            Console.WriteLine("xx defined");
        #else
            Console.WriteLine("xx not defined");
        #endif
    }
}
```

1.9.3.17 -delaysign

This compiler option causes the compiler to reserve space in the output file so that a digital signature can be added later.

Syntax

```
-delaysign[ + | - ]
```

Arguments

+ | -   Use -delaysign- if you want a fully signed assembly. Use -delaysign+ if you only want to place the public key in the assembly. The default is -delaysign-.

Remarks

The -delaysign option has no effect unless used with -keyfile or -keycontainer.

When you request a fully signed assembly, the compiler hashes the file that contains the manifest (assembly metadata) and signs that hash with the private key. The resulting digital signature is stored in the file that contains the manifest. When an assembly is delay signed, the compiler does not compute and store the signature, but reserves space in the file so the signature can be added later.

For example, using -delaysign+ allows a tester to put the assembly in the global cache. After testing, you can fully sign the assembly by placing the private key in the assembly using the Assembly Linker utility.

For more information, see Creating and Using Strong-Named Assemblies and Delay Signing an Assembly.

To set this compiler option in the Visual Studio development environment

1. Open the Properties page for the project.
2. Modify the Delay sign only property.
3. Click here to see the property page

For information on how to set this compiler option programmatically, see DelaySign.

1.9.3.18 -dialect

The -dialect compiler option selects the dialect that the compiler should follow. Valid options are: Core, VO, Vulcan, dBase, FoxPro, xBase++ and Harbour.
Note: There should be NO space between the colon and the dialect.

Syntax

```
-dialect: Core | VO | Vulcan | Harbour | XbasePP | FoxPro
```

Arguments

Core	Use the Core dialect. This is like C# with an xBase syntax.
VO	This activates the VO dialect.
Vulcan	This activates the Vulcan dialect.
Harbour	This activates the Harbour dialect.
XBasePP	This activates the Harbour dialect.
FoxPro	This activates the FoxPro dialect.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Application property page.
3. Modify the Dialect property.
4. Click here to see the property page

See the dialect topic for differences between dialects

1.9.3.19 -doc

The -doc compiler option allows you to place documentation comments in an XML file.

Syntax

```
-doc:file
```

Arguments

file   The output file for XML, which is populated with the
                              comments in the source code files of the compilation.

Remarks

In source code files, documentation comments that precede the following can be
processed and added to the XML file:

- Such user-defined types as a class, delegate, or interface
- Such members as a field, event, property, or method

The source code file that contains Main is output first into the XML.

To use the generated .xml file for use with the IntelliSense feature, let the file name of the
.xml file be the same as the assembly you want to support and then make sure the .xml
file is in the same directory as the assembly. Thus, when the assembly is referenced in
the Visual Studio project, the .xml file is found as well. See Supplying Code Comments
and for more information.

Unless you compile with -target:module, file will contain <assembly></assembly> tags
specifying the name of the file containing the assembly manifest for the output file of the
compilation.

Note
The -doc option applies to all input files; or, if set in the Project Settings, all files in the
project. To disable warnings related to documentation comments for a specific file or
section of code, use #pragma warning.

See Recommended Tags for Documentation Comments for ways to generate
documentation from comments in your code.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build tab.
3. Modify the XML documentation file property.
4. Click here to see the property page

X#

For information on how to set this compiler option programmatically, see DocumentationFile.

1.9.3.20 -enforceoverride

This compiler option tells the compiler NOT to automatically add the OVERRIDE modifier to methods in subclasses that override virtual methods in parent classes.

Syntax

```
-enforceoverride[+ | -]
```

Arguments

+ | -   Specifying +, or just -

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Language tab.
3. Change the value.
4. [Click here to see the property page](#)

1.9.3.21 -enforceself

The -enforceself compiler option allows you to control if you can access members of the SELF objects with or without the SELF: prefix.
In Visual Objects and most other original languages this was not allowed. This is equivalent to /enforceself+. In .Net the default is that this is allowed. This is equivalent to /enforceself-.
The effect of this is that the compiler may report an ambiguity when a method exists with the same name as a built in function. If you enable /enforceself then the compiler "knows" that without the SELF: prefix you intend to call the function.

Syntax

```
-enforceself[+ | -]
```

Arguments

+ | -   Specifying +, or just -

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.

2. Click the Language tab.
3. Change the value.
4. [Click here to see the property page](#)

1.9.3.22 -errorendlocation

The -errorendlocation tells the compiler to Output the line and column of the end location of each error in addition to the start location of the error/

Syntax

```
-errorendlocation
```

Remarks

This compiler option is unavailable in Visual Studio and cannot be changed programmatically. It is enabled by default when compiling inside Visual Studio.

1.9.3.23 -errorreport

This compiler option provides a convenient way to report a X# internal compiler error

 Note
On Windows Vista and Windows Server 2008, the error reporting settings that you make for Visual Studio do not override the settings made through Windows Error Reporting (WER). WER settings always take precedence over Visual Studio error reporting settings.

Syntax

```
-errorreport:{ none | prompt | queue | send }
```

Arguments

none   Reports about internal compiler errors will not be collected or sent to Microsoft.

prompt   Prompts you to send a report when you receive an internal compiler error. prompt is the default when you compile an application in the development environment.

queue   Queues the error report. When you log on with administrative credentials, you can report any failures since the last time that you were logged on. You will not be prompted to send reports for failures more than once every

X#

three days. queue is the default when you compile an application at the command line.

send	Automatically sends reports of internal compiler errors to Microsoft. To enable this option, you must first agree to the Microsoft data collection policy. The first time that you specify -errorreport:send on a computer, a compiler message will refer you to a Web site that contains the Microsoft data collection policy.

Remarks

An internal compiler error (ICE) results when the compiler cannot process a source code file. When an ICE occurs, the compiler does not produce an output file or any useful diagnostic that you can use to fix your code.

A user's ability to send reports depends on computer and user policy permissions.

For more information about error debugger, see Description of the Dr. Watson for Windows (Drwtsn32.exe) Tool.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build property page.
3. Click the Advanced button.

1.9.3.24 *-filealign*

The -filealign compiler option lets you specify the size of sections in your output file.

Syntax

```
-filealign:number
```

Arguments

number	A value that specifies the size of sections in the output file. Valid values are 512, 1024, 2048, 4096, and 8192. These values are in bytes.

Remarks

Each section will be aligned on a boundary that is a multiple of The -filealign value. There is no fixed default. If -filealign is not specified, the common language runtime picks a default at compile time.

By specifying the section size, you affect the size of the output file. Modifying section size may be useful for programs that will run on smaller devices.

Use DUMPBIN to see information about sections in your output file.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build property page.
3. Add the option in the "User-Defined Command Line options" property

1.9.3.25 -fovf

The -fovf compiler option is an alias for the <u>-checked</u> command line option for compatibility.

1.9.3.26 -fox1

The -fox1 compiler option tells the compiler that all classes with the FoxPro class syntax are assumed to inherit from the Custom class. This means:
- the AS <idType> CLASS becomes mandatory
- the code generation for properties assumes that the class inherits from the Custom class and creates Get and Set methods that read/write the values to the properties collection inside the Custom class

Syntax

```
-fox1[+ | -]
```

Arguments

+ | -   Specifying +, or just -

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. <u>Click here to see the property page</u>

1.9.3.27 -fox2

The -fox2 compiler option is enabled for foxpro compatible array support. It enables / disables the following features:

- Assigning a single value to an array will fill the whole array with that value
- Support for parenthesized array access.
  If the compiler encounters code that might be an array access such as `Foo(1,2)` then it will check to see if Foo() is a local or field declared as a foxpro array. If that is the case then the array element will be accessed. If Foo is not declared, then this might be an array access for a dynamic memory variable declared outside of the scope of the current method / function. The compiler will then generate code to resolve this at runtime (just like FoxPro does)

Requirements

The -fox2 compiler options MUST be used together with the -memvar compiler option and the FoxPro dialect.

Example

```
    LOCAL a
    Dimension a(10)
    a = 42          // With the /fox2 compiler option this will fill
the array with 42.
                    // Without the option the variable will be
changed from an array to a number
```

or

```
    Function Main()
    Dimension foo(2,5)
      foo = 42          // with /fox2 this fills the array
      ? foo(1,2)        // with /fox2 this returns element 1,2.
Without /fox2 this will call the Foo function below
      ? Foo[1,2]        // this always returns element 1,2

    FUNCTION Foo(n1, n2)
    ? n1, n2
    RETURN n1 * n2
```

Please note

The fox2 compiler option generates some extra code to decide at runtime which action to take.
It is only recommended for code that really needs this feature.
You can use #pragma options to enable / disable fox2 for some source files or even some functions

1.9.3.28 -fullpaths

The -fullpaths option causes the compiler to specify the full path to the file when listing compilation errors and warnings.

Syntax

```
-fullpaths
```

Remarks

By default, errors and warnings that result from compilation specify the name of the file in which an error was found. The -fullpaths option causes the compiler to specify the full path to the file.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically. It is enabled by default when compiling inside Visual Studio.

1.9.3.29 -help, /?

This option sends a listing of compiler options, and a brief description of each option, to stdout.

Syntax

```
-help
-?
```

Remarks

If this option is included in a compilation, no output file will be created and no compilation will take place.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

1.9.3.30 -highentropyva

The -highentropyva compiler option tells the Windows kernel whether a particular executable supports high entropy Address Space Layout Randomization (ASLR).

Syntax

```
-highentropyva[+ | -]
```

Arguments

+ | -   This option specifies that a 64-bit executable or an
                          executable that is marked by The -platform:anycpu compiler
                          option supports a high entropy virtual address space. The
                          option is disabled by default. Use -highentropyva+ or -
                          highentropyva to enable it.

Remarks

The -highentropyva option enables compatible versions of the Windows kernel to use
higher degrees of entropy when randomizing the address space layout of a process as
part of ASLR. Using higher degrees of entropy means that a larger number of addresses
can be allocated to memory regions such as stacks and heaps. As a result, it is more
difficult to guess the location of a particular memory region.

When The -highentropyva compiler option is specified, the target executable and any
modules that it depends on must be able to handle pointer values that are larger than 4
gigabytes (GB) when they are running as a 64-bit process.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build property page.
3. Add the option in the "User-Defined Command Line options" property

1.9.3.31 -i

The -i option adds a directory to the list of directories the compiler searches for include
files.

Syntax

```
-i:dir
```

Arguments

dir   A directory to add to list list of directories the compiler
                          searches for include files.

Remarks

The compiler searches for files included with #include in the following order:

1. The current working directory
2. The include subdirectory of the Vulcan.NET installation directory (when applicable)
3. The include subdirectory of the XSharp installation directory

4. Any directories specified with -i
5. Any directories specified with the INCLUDE environment variable

Multiple directories can be specified by using The -i option multiple times.
The compiler does not search for filenames that are fully qualified in any location other than in the directory specified in the #include directive.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Build tab.
3. In the Preprocessor section, modify the "Additional include paths" property.
4. Click here to see the property page

1.9.3.32 -initlocals

The -initlocals option specifies whether the compiler should automatically initialize local variables

Syntax

```
-initlocals[+|-]
```

Arguments

+ | -   Specifying +, or just -ins, directs the compiler to init local variables or not.

Remarks

This can be used to automatically initialize all local variables and suppress compiler warnings about local variables that are not initialized.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Language tab.
3. In the General section, modify the "Initialize Local Variables" value
4. Click here to see the property page

1.9.3.33 -ins

The -ins option specifies whether the compiler should automatically include namespaces from assemblies compiled in VO/Vulcan dialect marked with the ImplicitNameSpaceAttribute.

Syntax

```
-ins[+|-]
```

Arguments

+ | -   Specifying +, or just -ins, directs the compiler to automatically include namespaces from assemblies marked with the VulcanImplicitNameSpaceAttribute.

Remarks

Class Libraries can be compiled with a special attribute:

```
[assembly: VulcanImplicitNamespaceAttribute( "SomeNameSpace )]
```

This attribute tells the compiler that classes that are placed inside that namespace should be automatically included when searching for classes, as if there was a #using SomeNameSpace statement in the source code.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Language tab.
3. In the Namespaces section, modify the Enable Implicit Lookup of Namespaces property.
4. Click here to see the property page

1.9.3.34 -keycontainer

Specifies the name of the cryptographic key container.

Syntax

```
-keycontainer:string
```

Arguments

string   The name of the strong name key container.

Remarks

When The -keycontainer option is used, the compiler creates a sharable component by inserting a public key from the specified container into the assembly manifest and signing the final assembly with the private key. To generate a key file, type sn -k file at the command line. sn -i installs the key pair into a container.

If you compile with -target:module, the name of the key file is held in the module and incorporated into the assembly when you compile this module into an assembly with -addmodule.

You can also specify this option as a custom attribute (System.Reflection.AssemblyKeyNameAttribute) in the source code for any Microsoft intermediate language (MSIL) module.

You can also pass your encryption information to the compiler with -keyfile. Use -delaysign if you want the public key added to the assembly manifest but want to delay signing the assembly until it has been tested.

For more information, see Creating and Using Strong-Named Assemblies and Delay Signing an Assembly.

To set this compiler option in the Visual Studio development environment

1. This compiler option is not available in the Visual Studio development environment.

You can programmatically access this compiler option with AssemblyKeyContainerName.

1.9.3.35 -keyfile

Specifies the filename containing the cryptographic key.

Syntax

```
-keyfile:file
```

Arguments

file   The name of the file containing the strong name key.

Remarks

When this option is used, the compiler inserts the public key from the specified file into the assembly manifest and then signs the final assembly with the private key. To generate a key file, type sn -k file at the command line.

If you compile with -target:module, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly with -addmodule.

You can also pass your encryption information to the compiler with -keycontainer. Use -delaysign if you want a partially signed assembly.

In case both -keyfile and -keycontainer are specified (either by command line option or by custom attribute) in the same compilation, the compiler will first try the key container. If that succeeds, then the assembly is signed with the information in the key container. If the compiler does not find the key container, it will try the file specified with -keyfile. If that succeeds, the assembly is signed with the information in the key file and the key information will be installed in the key container (similar to sn -i) so that on the next compilation, the key container will be valid.

Note that a key file might contain only the public key.

For more information, see Creating and Using Strong-Named Assemblies and Delay Signing an Assembly.

To set this compiler option in the Visual Studio development environment:

1. Open the Properties page for the project.
2. Click the Signing property page.
3. Modify the Choose a strong name key file property.
4. Click here to see the property page

You can programmatically access this compiler option with AssemblyOriginatorKeyFile.

1.9.3.36 -langversion

this compiler option is ignored. When compared to the C# compiler, X# always compiles in the languageversion Latest.

1.9.3.37 -lb

The -lb option specifies if the compiler should allow code that uses the Late Binding mechnism to call methods and or get/set properties

Syntax

```
-lb[+|-]
```

Arguments

+ | -   Specifying +, or just -lb, directs the compiler to generate a late-bound call to an instance variable or method when compiler cannot generate code for an early bound call.

Remarks

The X# compiler always attempts to generate early bound calls to all class methods, properties and fields. This is true even for methods referred to as "untyped" in Visual Objects. Strictly speaking, nothing is "untyped". If an early bound call cannot be generated, a compile-time error is raised.

In Visual Objects, it is possible to invoke methods and access instance variables on an object without the compiler knowing the exact type of the object. This is done by using a variable of type OBJECT or USUAL to hold the object reference. The "Only Early" option in the VO Application Options dialog must not be checked in order to allow this.

In addition, it is also possible to invoke a method on each element of an ARRAY. Each array element must contain an object that implements the specified method, or a runtime error will occur.

The -lb option is the exact opposite of the Visual Objects Only Early option.

Late binding incurs considerably more runtime overhead than early binding, and prevents compile-time parameter and return value checking. Any late-bound call has the potential to fail at runtime if the object does not support the field or property that is being accessed, the member being invoked or incorrect parameter types or count. Only use early binding for existing code or when there is no viable alternative.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Language tab.
3. In the Visual Objects Compatibility section, modify the Allow Late Binding property.

4. [Click here to see the property page](#)

Example

The following example will compile when The -lb switch is used. Without -lb, errors will be raised.

Note that using late-bound calls can affect the performance of an application and raises the possibility of runtime errors that would otherwise be caught by the compiler if early binding was used. Late binding should only be used for compatibility with existing VO code.

```

```

If The -lb option is enabled, then the above example will be compiled as if it was written:

```

```

Note that while this particular example will compile and execute correctly, if the definition of CLASS foo was changed and INSTANCE i was removed or changed to a method, the example would fail at runtime. Similarly, if METHOD bar was changed or removed, the example may also fail at runtime. For this reason, late-bound programming is strongly discouraged. Consider using subclassing and inheritance instead.

1.9.3.38 -lexonly

Runs the lexer and preprocessor on the source files

Syntax

```
-lexonly[+|-]
```

Arguments

+ | -	Specifying +, or just -lexonly, tells the compiler to only lex the source code and not to parse, bind and generate an output file. When combined with The -ppo option then there will be .ppo files written.

Remarks

This command line option allows you to check the code without generating a binary.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Build tab.
3. Add the option in the "Extra Command Line options" property

1.9.3.39 -lib

The -lib option specifies the location of assemblies referenced by means of the -reference option.

Syntax

```
-lib:dir1[,dir2]
```

Arguments

dir1	A directory for the compiler to look in if a referenced assembly is not found in the current working directory (the directory from which you are invoking the compiler) or in the common language runtime's system directory.
dir2	One or more additional directories to search in for assembly references. Separate additional directory names with a comma, and without white space between them.

Remarks

The compiler searches for assembly references that are not fully qualified in the following order:
1. Current working directory. This is the directory from which the compiler is invoked.
2. The common language runtime system directory.
3. Directories specified by -lib.
4. Directories specified by the LIB environment variable.

Use -reference to specify an assembly reference.

-lib is additive; specifying it more than once appends to any prior values.

An alternative to using -lib is to copy into the working directory any required assemblies; this will allow you to simply pass the assembly name to -reference. You can then delete the assemblies from the working directory. Since the path to the dependent assembly is

not specified in the assembly manifest, the application can be started on the target computer and will find and use the assembly in the global assembly cache.

Because the compiler can reference the assembly does not imply the common language runtime will be able to find and load the assembly at runtime. See How the Runtime Locates Assemblies for details on how the runtime searches for referenced assemblies.

To set this compiler option in the Visual Studio development environment
1. Open the project's Property Pages dialog box.
2. Click the References Path property page.
3. Modify the contents of the list box.

For information on how to set this compiler option programmatically, see ReferencePath.

Example

Compile t2.prg to create an .exe file. The compiler will look in the working directory and in the root directory of the C drive for assembly references.

```
xsc -lib:c:\ -reference:t2.dll t2.prg
```

1.9.3.40 -link

Causes the compiler to make COM type information in the specified assemblies available to the project that you are currently compiling.

Syntax

```
-link:fileList
-/ -or-
-l:fileList
```

Arguments

fileList   Required. Comma-delimited list of assembly file names. If
                            the file name contains a space, enclose the name in
                            quotation marks.

Remarks

The `-link` option enables you to deploy an application that has embedded type information. The application can then use types in a runtime assembly that implement the embedded type information without requiring a reference to the runtime assembly. If

various versions of the runtime assembly are published, the application that contains the embedded type information can work with the various versions without having to be recompiled. For an example, see Walkthrough: Embedding Types from Managed Assemblies.

Using The -link option is especially useful when you are working with COM interop. You can embed COM types so that your application no longer requires a primary interop assembly (PIA) on the target computer. The `-link` option instructs the compiler to embed the COM type information from the referenced interop assembly into the resulting compiled code. The COM type is identified by the CLSID (GUID) value. As a result, your application can run on a target computer that has installed the same COM types with the same CLSID values. Applications that automate Microsoft Office are a good example. Because applications like Office usually keep the same CLSID value across different versions, your application can use the referenced COM types as long as .NET Framework 4 or later is installed on the target computer and your application uses methods, properties, or events that are included in the referenced COM types.

The `-link` option embeds only interfaces, structures, and delegates. Embedding COM classes is not supported.

Note

When you create an instance of an embedded COM type in your code, you must create the instance by using the appropriate interface. Attempting to create an instance of an embedded COM type by using the CoClass causes an error.

To set the `-link` option in Visual Studio, add an assembly reference and set the Embed Interop Types property to true. The default for the Embed Interop Types property is false.

If you link to a COM assembly (Assembly A) which itself references another COM assembly (Assembly B), you also have to link to Assembly B if either of the following is true:
- A type from Assembly A inherits from a type or implements an interface from Assembly B.
- A field, property, event, or method that has a return type or parameter type from Assembly B is invoked.

Like The -reference compiler option, the `-link` compiler option uses the xsc.rsp response file, which references frequently used .NET Framework assemblies. Use The -noconfig compiler option if you do not want the compiler to use the xsc.rsp file.

The short form of `-link` is `-l`.

Generics and Embedded Types

The following sections describe the limitations on using generic types in applications that embed interop types.

Generic Interfaces

Generic interfaces that are embedded from an interop assembly cannot be used.

Types That Have Generic Parameters

Types that have a generic parameter whose type is embedded from an interop assembly cannot be used if that type is from an external assembly. This restriction does not apply to interfaces. For example, consider the Range interface that is defined in the Microsoft.Office.Interop.Excel assembly. If a library embeds interop types from the Microsoft.Office.Interop.Excel assembly and exposes a method that returns a generic type that has a parameter whose type is the Range interface, that method must return a generic interface, as shown in the following code example.

Example

The following code compiles source file `OfficeApp.prg` and reference assemblies from `COMData1.dll` and `COMData2.dll` to produce `OfficeApp.exe`.

```
xsc -link:COMData1.dll,COMData2.dll -out:OfficeApp.exe
OfficeApp.prg
```

1.9.3.41 -linkresource

Creates a link to a .NET Framework resource in the output file. The resource file is not added to the output file. This differs from The -resource option which does embed a resource file in the output file.

Syntax

```
-linkresource:filename[,identifier[,accessibility-modifier]]
```

Arguments

filename	The .NET Framework resource file to which you want to link from the assembly.
identifier (optional)	The logical name for the resource; the name that is used to load the resource. The default is the name of the file.
accessibility-modifier (optional)	The accessibility of the resource: public or private. The default is public.

Remarks

By default, linked resources are public in the assembly when they are created with the X# compiler. To make the resources private, specify `private` as the accessibility modifier. No other modifier other than `public` or `private` is allowed.

-linkresource requires one of The -target options other than -target:module.

If `filename` is a .NET Framework resource file created, for example, by Resgen.exe or in the development environment, it can be accessed with members in the System.Resources namespace. For more information, see System.Resources.ResourceManager. For all other resources, use the `GetManifestResource*` methods in the Assembly class to access the resource at run time.

The file specified in `filename` can be any format. For example, you may want to make a native DLL part of the assembly, so that it can be installed into the global assembly cache and accessed from managed code in the assembly. The second of the following examples shows how to do this. You can do the same thing in the Assembly Linker. The third of the following examples shows how to do this. For more information, see Al.exe (Assembly Linker) and Working with Assemblies and the Global Assembly Cache
.
-linkres is the short form of -linkresource.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

Example

Compile `in.prg` and link to resource file `rf.resource`:

```
xsc -linkresource:rf.resource in.prg
```

Example

Compile `A.prg` into a DLL, link to a native DLL N.dll, and put the output in the Global Assembly Cache (GAC). In this example, both A.dll and N.dll will reside in the GAC.

```
xsc -linkresource:N.dll -t:library A.prg
gacutil -i A.dll
```

Example

This example does the same thing as the previous one, but by using Assembly Linker options.

```
xsc -t:module A.prg
al -out:A.dll A.netmodule -link:N.dll
gacutil -i A.dll
```

1.9.3.42 -main

This option specifies the class that contains the entry point to the program, if more than one class contains a Main method.

Syntax

```
-main:class
```

Arguments

class   The type that contains the Main method.

Remarks

If your compilation includes more than one type with a Main method, you can specify which type contains the Main method that you want to use as the entry point into the program.

This option is for use when compiling an .exe file.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Application property page.
3. Modify the Startup object property.
4. Click here to see the property page

To set this compiler option programmatically, see StartupObject.

Example

Compile `t2.prg` and `t3.prg`, specifying that the Main method will be found in `Test2`:

```
xsc t2.prg t3.prg -main:Test2
```

1.9.3.43 -memvar

The -memvar option tells the compiler to enable the support for 'memory variables'. This option is also needed in the FoxPro dialect if you want to make local variables visible to the macro compiler with the -fox2 compilation option.

Syntax

```
-memvar [+ | -]
```

Arguments

+ | -   Specifying +, or just -memvar, directs the compiler to enable support for memory variables.

Remarks

This will NOT work with the Core and Vulcan dialects.
Enabling this option will enable the following commands in the compiler:

MEMVAR <MemvarName,...>
PUBLIC <MemVarName,...>
PRIVATE <MemVarName,...>
PARAMETERS <ParameterName,,...>

Click here to see the property page

1.9.3.44 -modernsyntax

The -modernsyntax compiler option disables some legacy language features
These features are

- && for single line comments

  ? Today() *&& Show today's date*

- * for whole line comments

```
* The next line calls DoSomething()
? DoSomething()
```

- Bracketed strings

```
? ["'"] // use brackets to include quotes in a literal string
```

- Parenthesized expression lists

```
var x := IIF(Dow() == 1, (Open(), DoSomeWork(), Close()), 42)
```

These were used by some users to include more than one expression / function call on locations where a single expression was expected, such as inside an IIF() expression. If -modernsyntax is not enabled then the compiler will generate a local function with (in this example) 3 expression statements and insert a call to the local function in the IIF() expression.

1.9.3.45 -moduleassemblyname

Specifies an assembly whose non-public types a .netmodule can access.

Syntax

```
-moduleassemblyname:assembly_name
```

Arguments

assembly_name   The name of the assembly whose non-public types the .netmodule can access.

Remarks

-moduleassemblyname should be used when building a .netmodule, and where the following conditions are true:

- The .netmodule needs access to non-public types in an existing assembly.
- You know the name of the assembly into which the .netmodule will be built.
- The existing assembly has granted friend assembly access to the assembly into which the .netmodule will be built.

For more information on building a .netmodule, see -target:module.

For more information on friend assemblies, see Friend Assemblies.

This option is not available from within the development environment; it is only available when compiling from the command line.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

Example

This sample builds an assembly with a private type, and that gives friend assembly access to an assembly called csman_an_assembly.

```
-/ moduleassemblyname_1.prg
-/ compile with: -target:library
using System;
using System.Runtime.CompilerServices;

  [assembly:InternalsVisibleTo ("csman_an_assembly")]

  class An_Internal_Class

{

    public void Test()
    {

        Console.WriteLine("An_Internal_Class.Test called");
    }
}
```

Example

This sample builds a .netmodule that accesses a non-public type in the assembly moduleassemblyname_1.dll. By knowing that this .netmodule will be built into an assembly called csman_an_assembly, we can specify -moduleassemblyname, allowing the .netmodule to access non-public types in an assembly that has granted friend assembly access to csman_an_assembly.

```
-/ moduleassemblyname_2.prg
-/ compile with: -moduleassemblyname:csman_an_assembly -
target:module -reference:moduleassemblyname_1.dll
class B {
```

```
        public void Test() {
            An_Internal_Class x = new An_Internal_Class();
            x.Test();
        }
}
```

Example

This code sample builds the assembly csman_an_assembly, referencing the previously-built assembly and .netmodule.

```
-/ csman_an_assembly.prg
-/ compile with: -addmodule:moduleassemblyname_2.netmodule -
reference:moduleassemblyname_1.dll
class A {
    public static void Main() {
        B bb = new B();
        bb.Test();
    }
}
```

1.9.3.46 -modulename:<string>

Specify the name of the source module

1.9.3.47 -namedargs

Specifies whether to allow named arguments in the parser or not.

Syntax

```
-namedargs [+ | -]
```

Arguments

+ |-   Specifying +, or just -namedargs, directs the compiler to
                                  allow namedarguments. Specifying - directs the compiler to
                                  NOT use named arguments

Remarks

The default = + for the Core dialect and - for all other dialects. So -namedargs- only is useful in combination with the Core dialect.
Note If you enable this option then quite often existing code such as the code below will produce compiler errors>
That is why we have disabled named arguments for most dialects.

```
FUNCTION Start as VOID
LOCAL a AS ARRAY
-/ When named arguments are enabled then the compiler will
complain that there is no parameter named "a" for the Empty
function
IF !Empty( a := SomeFunctionThatReturnsAnArray())
 -/ do something
ENDIF
RETURN
```

[Click here to see the property page](#)

1.9.3.48 -noconfig

The -noconfig option tells the compiler not to compile with the xsc.rsp file, which is located in and loaded from the same directory as the xsc.exe file.

Syntax

```
-noconfig
```

Remarks

The xsc.rsp file references all the assemblies shipped with the .NET Framework. The actual references that the Visual Studio .NET development environment includes depend on the project type.

You can modify the xsc.rsp file and specify additional compiler options that should be included in every compilation from the command line with xsc.exe (except The -noconfig option).

The compiler processes the options passed to the xsc command last. Therefore, any option on the command line overrides the setting of the same option in the xsc.rsp file.

If you do not want the compiler to look for and use the settings in the xsc.rsp file, specify -noconfig.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

The current contents of xsc.rsp is:

```
# Copyright (c)  XSharp BV.  All Rights Reserved.

# This file contains command-line options that the X#
# command line compiler (XSC) will process as part
# of every compilation, unless the "/noconfig" option
# is specified.

# Reference the common Framework libraries
/r:Accessibility.dll
/r:Microsoft.CSharp.dll
/r:System.Configuration.dll
/r:System.Configuration.Install.dll
/r:System.Core.dll
/r:System.Data.dll
/r:System.Data.DataSetExtensions.dll
/r:System.Data.Linq.dll
/r:System.Data.OracleClient.dll
/r:System.Deployment.dll
/r:System.Design.dll
/r:System.DirectoryServices.dll
/r:System.dll
/r:System.Drawing.Design.dll
/r:System.Drawing.dll
/r:System.EnterpriseServices.dll
/r:System.Management.dll
/r:System.Messaging.dll
/r:System.Runtime.Remoting.dll
/r:System.Runtime.Serialization.dll
/r:System.Runtime.Serialization.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceModel.dll
/r:System.ServiceModel.Web.dll
/r:System.ServiceProcess.dll
/r:System.Transactions.dll
/r:System.Web.dll
/r:System.Web.Extensions.Design.dll
/r:System.Web.Extensions.dll
/r:System.Web.Mobile.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.dll
/r:System.Workflow.Activities.dll
/r:System.Workflow.ComponentModel.dll
/r:System.Workflow.Runtime.dll
/r:System.Xml.dll
/r:System.Xml.Linq.dll
```

1.9.3.49 -noinit

Suppress generation of empty $Init1() and $Exit() functions

Background info

The X# compiler generates special functions in each assembly that are used to call INIT procedures and EXIT procedures.
There are 3 levels of INIT procedures.
For each of these 3 levels a special function is created: $Init1(), $Init2() and $Init3(). For EXIT procedures the compiler creates a function $Exit().
The functions $Init2() and $Init3() are only created when needed.
The functions $Init1() and $Exit() are always created.

When you compile an EXE with X# then the compiler generates code that calls the $Init1(), $Init2() and $Init3() functions in all referenced assemblies at startup and the $Exit() functions at shutdown.

This mechanism also guarantees that classes in referenced assemblies are available at runtime, even when you have not explicitly referenced them in your code, so you can instantiate these classes with CreateInstance().

The compiler option -noinit suppresses the generation of empty $Init1() and $Exit() functions. As a result there will be no hard link to external assemblies if you do not reference code from these assemblies.
If you use these compiler option with an assembly that only contains DEFINES then the defines will be resolved at compile time and you will not need to include the assembly at runtime (unless these defines contain values that need to be resolved at runtime, such as symbols or date values).

1.9.3.50 -nologo

The -nologo option suppresses display of the sign-on banner when the compiler starts up and display of informational messages during compiling.

Syntax

```
-nologo
```

Remarks

This option is not available from within the development environment; it is only available when compiling from the command line.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

1.9.3.51   -nostddefs

The -nostddefs option prevents the compiler from using the preprocessor directives in XSharpDefs.xh.

Syntax

```
-nostddefs[+|-]
```

Arguments

+ |-   Specifying +, or just -nostddefs, prevents the compiler from using the preprocessor directives in XSharpDefs.xh.

Remarks

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Build tab.
3. In the Preprocessor section, modify the "Suppress standard header file" property.
4. Click here to see the property page

1.9.3.52   -nostdlib

-nostdlib prevents the import of mscorlib.dll, which defines the entire System namespace.

Syntax

```
-nostdlib[+ | -]
```

Remarks

Use this option if you want to define or create your own System namespace and objects.

If you do not specify -nostdlib, mscorlib.dll will be imported into your program (same as specifying -nostdlib-). Specifying -nostdlib is the same as specifying -nostdlib+.

To set this compiler option in the Visual Studio development environment

1. Open the Properties page for the project.
2. Click the Build properties page.
3. Add the option in the "User-Defined Command Line options" property

1.9.3.53 -nowarn

The -nowarn option lets you suppress the compiler from displaying one or more warnings. Separate multiple warning numbers with a comma.

Syntax

```
-nowarn:number1[,number2,...]
```

Arguments

number1, number2   Warning number(s) that you want the compiler to suppress.

Remarks

You should only specify the numeric part of the warning identifier. For example, if you want to suppress XS0028, you could specify -nowarn:28.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build property page.
3. Modify the "Suppress Specific Warnings" property.
4. Click here to see the property page

1.9.3.54 -nowin32manifest

Use The -nowin32manifest option to instruct the compiler not to embed any application manifest into the executable file.

Syntax

```
-nowin32manifest
```

Remarks

When this option is used, the application will be subject to virtualization on Windows Vista unless you provide an application manifest in a Win32 Resource file or during a later build step.

Click here to see the property page

1.9.3.55   -ns

The -ns option explicitly specifies the default namespace for all types that do not have an explicit namespace in their name.

Syntax

```
-ns[: ]namespaceName
```

Arguments

namespaceName   The name of the default namespace for all types declared in
                                  the application or class library.

Remarks

If The -ns option is not specified, then types that are not prefixed with a namespace and types that are not in a BEGIN NAMESPACE .. END NAMESPACE construct will be compiled as so-called global types.

The -ns option will work on the following types:

- Classes
- Interfaces
- Structures
- Vostructs
- Delegates

Namespace names must follow the same rules for program identifiers: they must begin with an uppercase or lowercase letter or an underscore, followed by zero or more uppercase or lowercase letters, numbers or underscores. All other characters are illegal, and will raise a compile-time error.

The default namespace is used for any declared types that do not have an explicit namespace in their name.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the General tab.
3. In the Application section, modify the Default Namespace property.
4. Click the Language tab
5. In the Namespaces section, modify the Prefix classes with Default Namespace property.
6. Click here to see the property page

Example

When the following code is compiled without -ns compiler option the following types will be produced:

- Customer
- Point
- MyProject.Customer
- MyProject.Data.Customer

```
CLASS Customer
.
.
END CLASS

STRUCT Point
.
.
END STRUCT

ENUM CustomerType
.
END ENUM

CLASS MyProject.Customer
.
.
END CLASS


BEGIN NAMESPACE MyProject.Data
CLASS Customer
.
.
END CLASS
END NAMESPACE
```

If you compile the same code with a -ns:MyNameSpace option the following types will be produced:

- MyNameSpace.Customer
- MyNameSpace.Point
- MyProject.Customer
- MyProject.Data.Customer

1.9.3.56 -optimize

The -optimize option enables or disables optimizations performed by the compiler to make your output file smaller, faster, and more efficient.

Syntax

```
-optimize[+ | -]
```

Remarks

-optimize also tells the common language runtime to optimize code at runtime.

By default, optimizations are disabled. Specify -optimize+ to enable optimizations.

When building a module to be used by an assembly, use the same -optimize settings as those of the assembly.

-o is the short form of -optimize.

It is possible to combine The -optimize and -debug options.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build property page.
3. Modify the "Optimize" property.
4. Click here to see the property page

Example

Compile `t2.prg` and enable compiler optimizations:

```
xsc t2.prg -optimize
```

1.9.3.57 -out

The -out option specifies the name of the output file.

Syntax

```
-out:filename
```

Arguments

filename   The name of the output file created by the compiler.

Remarks

On the command line, it is possible to specify multiple output files for your compilation. The compiler expects to find one or more source code files following The -out option. Then, all source code files will be compiled into the output file specified by that -out option.

Specify the full name and extension of the file you want to create.

If you do not specify the name of the output file:

- An .exe will take its name from the source code file that contains the Main method.
- A .dll or .netmodule will take its name from the first source code file.

A source code file used to compile one output file cannot be used in the same compilation for the compilation of another output file.

When producing multiple output files in a command-line compilation, keep in mind that only one of the output files can be an assembly and that only the first output file specified (implicitly or explicitly with -out) can be the assembly.

Any modules produced as part of a compilation become files associated with any assembly also produced in the compilation. Use ildasm.exe to view the assembly manifest to see the associated files.

The -out compiler option is required in order for an exe to be the target of a friend assembly.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Application property page.
3. Modify the Assembly name property.
4. Click here to see the property page

To set this compiler option programmatically: the OutputFileName is a read-only property, which is determined by a combination of the project type (exe, library, and so forth) and

the assembly name. Modifying one or both of these properties will be necessary to set the output file name.

Example

Compile t.prg and create output file `t.exe`, as well as build `t2.prg` and create module output file `mymodule.netmodule`:

```
xsc t.prg -out:mymodule.netmodule -target:module t2.prg
```

1.9.3.58 -ovf

The -ovf compiler option is an alias for the -checked command line option for compatibility.

Click here to see the property page

1.9.3.59 -parallel

Specifies whether to use concurrent build (+).

1.9.3.60 -parseonly

Performs a syntax check on the input files only. Does not produce the output exe, dll or pdb

Syntax

```
-parseonly[+|-]
```

Arguments

+ | - Specifying +, or just -parseonly, tells the compiler to only parse the source code and not to not to bind and generate an output file.
When combined with The -ppo option then there will be .ppo files written.

Remarks

This command line option allows you to check the code without generating a binary.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Build tab.
3. Add the option in the "Extra Command Line options" property

1.9.3.61 -pathmap

Specify a mapping for source path names output by the compiler.

Syntax

```
-pathmap:path1=sourcePath1,path2=sourcePath2
```

Arguments

path1	The full path to the source files in the current environment
sourcePath1	The source path substituted for path1 in any output files.

To specify multiple mapped source paths, separate each with a comma.

Remarks

The compiler writes the source path into its output for the following reasons:

1. The source path is substituted for an argument when the CallerFilePathAttribute is applied to an optional parameter.
2. The source path is embedded in a PDB file.
3. The path of the PDB file is embedded into a PE (portable executable) file.

This option maps each physical path on the machine where the compiler runs to a corresponding path that should be written in the output files.

1.9.3.62 -pdb

The -pdb compiler option specifies the name and location of the debug symbols file.

Syntax

```
-pdb:filename
```

Arguments

filename	The name and location of the debug symbols file.

Remarks

When you specify <u>-debug</u> , the compiler will create a .pdb file in the same directory where the compiler will create the output file (.exe or .dll) with a file name that is the same as the name of the output file.

-pdb allows you to specify a non-default file name and location for the .pdb file.

This compiler option cannot be set in the Visual Studio development environment, nor can it be changed programmatically.

Example

Compile `t.prg` and create a .pdb file called tt.pdb:

```
xsc -debug -pdb:tt t.prg
```

1.9.3.63 -platform

Specifies which version of the common language runtime (CLR) can run the assembly.

Syntax

```
-platform:string
```

Arguments

string   anycpu (default), anycpu32bitpreferred, ARM, x64, x86, or
                              Itanium.

Remarks

- anycpu (default) compiles your assembly to run on any platform. Your application runs as a 64-bit process whenever possible and falls back to 32-bit when only that mode is available.
- anycpu32bitpreferred compiles your assembly to run on any platform. Your application runs in 32-bit mode on systems that support both 64-bit and 32-bit applications. You can specify this option only for projects that target the .NET Framework 4.5.
- ARM compiles your assembly to run on a computer that has an Advanced RISC Machine (ARM) processor.

- x64 compiles your assembly to be run by the 64-bit common language runtime on a computer that supports the AMD64 or EM64T instruction set.
- x86 compiles your assembly to be run by the 32-bit, x86-compatible common language runtime.
- Itanium compiles your assembly to be run by the 64-bit common language runtime on a computer with an Itanium processor.

On a 64-bit Windows operating system:

- Assemblies compiled with -platform:x86 execute on the 32-bit CLR running under WOW64.
- A DLL compiled with The -platform:anycpu executes on the same CLR as the process into which it is loaded.
- Executables that are compiled with The -platform:anycpu execute on the 64-bit CLR.
- Executables compiled with -platform:anycpu32bitpreferred execute on the 32-bit CLR.

The anycpu32bitpreferred setting is valid only for executable (.EXE) files, and it requires the .NET Framework 4.5.

To set this compiler option in the Visual Studio development environment

1. Open the Properties page for the project.
2. Click the General property page.
3. Modify the Platform target property and, for projects that target the .NET Framework 4.5, select or clear the Prefer 32-bit check box.
4. [Click here to see the property page](#)

Example

The following example shows how to use The -platform option to specify that the application should be run by the 64-bit CLR on a 64-bit Windows operating system.

```
xsc -platform:anycpu filename.prg
```

1.9.3.64 -ppo

The -ppo option directs the compiler to write the output of the preprocessor to a file.

Syntax

```
-ppo [+ | -]
```

Arguments

+ | -   Specifying +, or just -ppo, directs the compiler to write the output of the preprocessor to a file.
                         When ppo is not enabled then the compiler will delete any existing ppo files that match input file names

Remarks

Each source file will be written to a file with the same base name and an extension of .ppo, in the same directory as the source file. If a .ppo file already exists, it will be overwritten without warning.

Tip:

.ppo files can be viewed within Visual Studio. After enabling The -ppo option and rebuilding the project, select any node in the project in Solution Explorer, click the Show All Files button in the Solution Explorer toolbar, then double click the desired .ppo file node.

To view a .prg and .ppo file side-by-side to compare them, right click the document tab in either the .prg or .ppo file and click New Vertical Tab Group.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page (see How to: Set Build Properties).
2. Click the Build tab.
3. In the Preprocessor section, modify the "Generate preprocessor output"property.
4. Click here to see the property page

1.9.3.65 -preferreduilang

By using The -preferreduilang compiler option, you can specify the language in which the X# compiler displays output, such as error messages.

Syntax

```
-preferreduilang: language
```

Arguments

language   The language name of the language to use for compiler output.

Remarks

The preferreduilang compiler option is recognized but ignored. The X# compiler only works in the english language (for now ?)

1.9.3.66 **-recurse**

The -recurse option enables you to compile source code files in all child directories of either the specified directory (dir) or of the project directory.

Syntax

```
-recurse:[dir\]file
```

Arguments

dir (optional)	The directory in which you want the search to begin. If this is not specified, the search begins in the project directory.
file	The file(s) to search for. Wildcard characters are allowed.

Remarks

The -recurse option lets you compile source code files in all child directories of either the specified directory (dir) or of the project directory.
You can use wildcards in a file name to compile all matching files in the project directory without using -recurse.
This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

Example

Compiles all X# files in the current directory:

```
xsc *.prg
```

Compiles all of the X# files in the dir1\dir2 directory and any directories below it and generates dir2.dll:

```
xsc -target:library -out:dir2.dll -recurse:dir1\dir2\*.prg
```

1.9.3.67   -reference

The -reference option causes the compiler to import public type information in the specified file into the current project, thus enabling you to reference metadata from the specified assembly files.

Syntax

```
-reference:[alias=]filename
-reference:filename
```

Arguments

`filename`	The name of a file that contains an assembly manifest. To import more than one file, include a separate -reference option for each file.
`alias`	A valid X# identifier that will represent a root namespace that will contain all namespaces in the assembly.

Remarks

To import from more than one file, include a -reference option for each file.

The files you import must contain a manifest; the output file must have been compiled with one of The -target options other than -target:module.

-r is the short form of -reference.

Use -addmodule to import metadata from an output file that does not contain an assembly manifest.

If you reference an assembly (Assembly A) that references another assembly (Assembly B), you will need to reference Assembly B if:

- A type you use from Assembly A inherits from a type or implements an interface from Assembly B.
- You invoke a field, property, event, or method that has a return type or parameter type from Assembly B.

Use -lib to specify the directory in which one or more of your assembly references is located. The -lib topic also discusses the directories in which the compiler searches for assemblies.

In order for the compiler to recognize a type in an assembly, and not in a module, it needs to be forced to resolve the type, which you can do by defining an instance of the type. There are other ways to resolve type names in an assembly for the compiler: for example,

if you inherit from a type in an assembly, the type name will then be recognized by the compiler.

Sometimes it is necessary to reference two different versions of the same component from within one assembly. To do this, use the alias suboption on The -reference switch for each file to distinguish between the two files. This alias will be used as a qualifier for the component name, and will resolve to the component in one of the files.

The xsc response (.rsp) file, which references commonly used .NET Framework assemblies, is used by default. Use -noconfig if you do not want the compiler to use xsc.rsp.

Note

In Visual Studio, use the Add Reference dialog box.

Example

This example shows how to use the extern alias feature.

You compile the source file and import metadata from `grid.dll` and `grid20.dll`,which have been compiled previously. The two DLLs contain separate versions of the same component, and you use two -reference with alias options to compile the source file. The options look like this:

-reference:GridV1=grid.dll and -reference:GridV2=grid20.dll

This sets up the external aliases "GridV1" and "GridV2," which you use in your program by means of an extern statement:

```
extern alias GridV1;
extern alias GridV2;
-/ Using statements go here.
```

Once this is done, you can refer to the grid control from grid.dll by prefixing the control name with GridV1, like this:

```
GridV1::Grid
```

In addition, you can refer to the grid control from grid20.dll by prefixing the control name with GridV2 like this:

```
GridV2::Grid
```

1.9.3.68 -refonly

The -refonly option indicates that a reference assembly should be output instead of an implementation assembly, as the primary output. The -refonly parameter silently disables outputting PDBs, as reference assemblies cannot be executed.

Syntax

```
-refonly
```

Metadata-only assemblies have their method bodies replaced with a single throw null body, but include all members except anonymous types. The reason for using throw null bodies (as opposed to no bodies) is so that PEVerify could run and pass (thus validating the completeness of the metadata).

Reference assemblies include an assembly-level ReferenceAssembly attribute. This attribute may be specified in source (then the compiler won't need to synthesize it). Because of this attribute, runtimes will refuse to load reference assemblies for execution (but they can still be loaded in reflection-only mode). Tools that reflect on assemblies need to ensure they load reference assemblies as reflection-only, otherwise they will receive a typeload error from the runtime.

Reference assemblies further remove metadata (private members) from metadata-only assemblies:

- A reference assembly only has references for what it needs in the API surface. The real assembly may have additional references related to specific implementations.
- Private function-members (methods, properties, and events) are removed in cases where their removal doesn't observably impact compilation. If there are no InternalsVisibleTo attributes, do the same for internal function-members.
- But all types (including private or nested types) are kept in reference assemblies. All attributes are kept (even internal ones).
- All virtual methods are kept. Explicit interface implementations are kept. Explicitly implemented properties and events are kept, as their accessors are virtual (and are therefore kept).
- All fields of a struct are kept. (This is a candidate for post-C#-7.1 refinement)

The -refonly and -refout options are mutually exclusive.

1.9.3.69 -refout

The -refout option specifies a file path where the reference assembly should be output. This translates to metadataPeStream in the Emit API.

Syntax

```
-refout:filepath
```

Arguments

filepath   The name and path of the output file created by the compiler.

Remarks

The filename should generally match that of the primary assembly. The recommended convention (used by MSBuild) is to place the reference assembly in a "ref/" sub-folder relative to the primary assembly.

Metadata-only assemblies have their method bodies replaced with a single throw null body, but include all members except anonymous types. The reason for using throw null bodies (as opposed to no bodies) is so that PEVerify could run and pass (thus validating the completeness of the metadata).
Reference assemblies include an assembly-level ReferenceAssembly attribute. This attribute may be specified in source (then the compiler won't need to synthesize it). Because of this attribute, runtimes will refuse to load reference assemblies for execution (but they can still be loaded in reflection-only mode). Tools that reflect on assemblies need to ensure they load reference assemblies as reflection-only, otherwise they will receive a typeload error from the runtime.
Reference assemblies further remove metadata (private members) from metadata-only assemblies:
- A reference assembly only has references for what it needs in the API surface. The real assembly may have additional references related to specific implementations.
- Private function-members (methods, properties, and events) are removed in cases where their removal doesn't observably impact compilation. If there are no InternalsVisibleTo attributes, do the same for internal function-members.
- But all types (including private or nested types) are kept in reference assemblies. All attributes are kept (even internal ones).
- All virtual methods are kept. Explicit interface implementations are kept. Explicitly implemented properties and events are kept, as their accessors are virtual (and are therefore kept).
- All fields of a struct are kept.

The -refout and [-refonly](#) options are mutually exclusive.

1.9.3.70 -resource

Embeds the specified resource into the output file.

Syntax

```
-resource:filename[,identifier[,accessibility-modifier]]
```

Arguments

`filename`   The .NET Framework resource file that you want to embed in the output file.

`identifier` (optional)   The logical name for the resource; the name that is used to load the resource. The default is the name of the file name.

`accessibility-modifier` (optional)   The accessibility of the resource: public or private. The default is public.

Remarks

Use -linkresource to link a resource to an assembly and not add the resource file to the output file.

By default, resources are public in the assembly when they are created by using the X# compiler. To make the resources private, specify `private` as the accessibility modifier. No other accessibility other than `public` or `private` is allowed.

If `filename` is a .NET Framework resource file created, for example, by Resgen.exe or in the development environment, it can be accessed with members in the System.Resources namespace. For more information, see System.Resources.ResourceManager. For all other resources, use the `GetManifestResource*` methods in the Assembly class to access the resource at run time.

-res is the short form of -resource.

The order of the resources in the output file is determined from the order specified on the command line.

To set this compiler option in the Visual Studio development environment

1. Add a resource file to your project.
2. Select the file that you want to embed in Solution Explorer.
3. Select Build Action for the file in the Properties window.
4. Set Build Action to Embedded Resource.

For information about how to set this compiler option programmatically, see BuildAction.

Example

Compile `in.prg` and attach resource file `rf.resource`:

```
xsc -resource:rf.resource in.prg
```

1.9.3.71 **-ruleset**

Specify a ruleset file that disables specific diagnostics.

Syntax

```
-ruleset:filename
```

1.9.3.72 **-shared**

Used the shared compiler process (XSCompiler.exe) to compile the project. This process caches type information so repeated compilations will be faster

Syntax

```
/shared[: pipeName]
```

Arguments

pipeName   This optional parameter specifies the pipename that the
                                      compiler uses to communicate between the foreground
                                      process (xsc.exe) and the background compiler
                                      (xscompiler.exe)

Remarks

This commandline option is useful to speed up compilation

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Build tab.
3. Check or uncheck the "Use Shared compiler" option.

1.9.3.73 -showdefs

Shows the #define tokens that the preprocessor sees

Syntax

```
/showdefs[+|-]
```

Arguments

+ | -   Specifying +, or just `/showdefs`, causes the compiler to
                                 output the names of #defines as they are processed.

Remarks

This commandline option is useful to debug the defines that are included in the
compilation

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Build tab.
3. Add the option in the "Extra Command Line options" property

1.9.3.74 -showincludes

The -showincludes option causes the compiler to output the names of #include files as
they are processed.

Syntax

```
-showincludes[+|-]
```

Arguments

+ | -   Specifying +, or just -showincludes, causes the compiler to
                                 output the names of #include files as they are processed.

Remarks

This option is useful in order to determine the files the compiler is processing as a result of an #include directive. Nested include files are also displayed.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Build tab.
3. Add the option in the "Extra Command Line options" property

1.9.3.75 -snk

The -snk compiler option is an alias for the -keyfile option, for compatibility .

1.9.3.76 -stddefs

Override the name of the standard defines file (default = XSharpDefs.xh)

Syntax

```
-stddefs:filepath
```

Arguments

filepath   The name and path of the header file to use in stead of XSharpDefs.Xh. When no path is included then the compiler will use the Includepath specified from the commandline (-i)

Click here to see the property page

1.9.3.77 -subsystemversion

Specifies the minimum version of the subsystem on which the generated executable file can run, thereby determining the versions of Windows on which the executable file can run. Most commonly, this option ensures that the executable file can leverage particular security features that aren't available with older versions of Windows.

Note
To specify the subsystem itself, use The -target compiler option.

Syntax

```
-subsystemversion:major.minor
```

Parameters

major.minor	The minimum required version of the subsystem, as expressed in a dot notation for major and minor versions. For example, you can specify that an application can't run on an operating system that's older than Windows 7 if you set the value of this option to 6.01, as the table later in this topic describes. You must specify the values for major and minor as integers.

Leading zeroes in the minor version don't change the version, but trailing zeroes do. For example, 6.1 and 6.01 refer to the same version, but 6.10 refers to a different version. We recommend expressing the minor version as two digits to avoid confusion.

Remarks

The following table lists common subsystem versions of Windows.

Windows version	Subsystem version
Windows 2000	5.00
Windows XP	5.01
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

Default values

The default value of The -subsystemversion compiler option depends on the conditions in the following list:

- The default value is 6.02 if any compiler option in the following list is set:
    - -target:appcontainerexe
    - -target:winmdobj
    - -platform:arm
- The default value is 6.00 if you're using MSBuild, you're targeting .NET Framework 4.5, and you haven't set any of the compiler options that were specified earlier in this list.
- The default value is 4.00 if none of the previous conditions is true.

Setting this option

To set The -subsystemversion compiler option in Visual Studio, you must open the .xsproj file and specify a value for the `SubsystemVersion` property in the MSBuild XML. You can't set this option in the Visual Studio IDE.

1.9.3.78 -target

The -target compiler option can be specified in one of four forms:

-target:appcontainerexe   To create an .exe file for Windows 8.x Store apps.
-target:exe   To create an .exe file.
-target:library   To create a code library.
-target:module   To create a module.
-target:winexe   To create a Windows program.
-target:winmdobj   To create an intermediate .winmdobj file.

Unless you specify -target:module, -target causes a .NET Framework assembly manifest to be placed in an output file. For more information, see Assemblies in the Common Language Runtime and Common Attributes.
The assembly manifest is placed in the first .exe output file in the compilation or in the first DLL, if there is no .exe output file. For example, in the following command line, the manifest will be placed in 1.exe:

```
xsc -out:1.exe t1.prg -out:2.netmodule t2.prg
```

The compiler creates only one assembly manifest per compilation. Information about all files in a compilation is placed in the assembly manifest. All output files except those created with -target:module can contain an assembly manifest. When producing multiple output files at the command line, only one assembly manifest can be created and it must go into the first output file specified on the command line. No matter what the first output file is (/target:exe, -target:winexe, -target:library or -target:module), any other output files produced in the same compilation must be modules (/target:module).

If you create an assembly, you can indicate that all or part of your code is CLS compliant with the CLSCompliantAttribute attribute.

```
// target_clscompliant.prg
[assembly:System.CLSCompliant(true)]  // specify assembly
compliance

[System.CLSCompliant(false)];  // specify compliance for an
element
CLASS TestClass

    PUBLIC STATIC METHOD Start AS VOID
        RETURN
END CLASS
```

For more information about setting this compiler option programmatically, see OutputType.Enter topic text here.

1.9.3.79   -touchedfiles

The -touchedfiles compiler option allows you to declare a file in which the compiler will list which files have been touched during the compilation process.

Syntax

```
-touchedfiles:filename
```

Remarks

The compiler will produce 2 output files:

filename.read	This lists all the files that were read by the compiler
filename.write	This lists all the files that were written by the compiler. That includes the exe, pdb, ppo etc.

The file names in both files are all capitalized and in alphabetical order

1.9.3.80   -undeclared

The -undeclared option tells the compiler to enable the support for 'undeclared variables'

Syntax

```
-undeclared [+ | -]
```

Arguments

+ | -	Specifying +, or just -undeclared, directs the compiler to enable the support for 'undeclared variables'

Remarks

This will NOT work with the Core and Vulcan dialects.
When the compiler detects an 'unknown identifier' it will assume that this is either a field in the current workarea or a memory variable.
The compiler option will only have effect when you also enabled -memvar
[Click here to see the property page](#)

1.9.3.81 -unsafe

The -unsafe compiler option allows code that uses the unsafe keyword to compile.

Syntax

```
-unsafe
```

Remarks

For more information about unsafe code, see Unsafe Code and Pointers.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Language property page.
3. Set the Allow Unsafe Code property in the General Section
4. [Click here to see the property page](#)

Example

Compile `in.prg` for unsafe mode:

```
xsc -unsafe in.prg
```

1.9.3.82 -usenativeversion

The -usenativeversion compiler option tells the compiler to use the native Win32 version resource supplied in a Win32 resource file and to not generate a resource from the Assembly attributes

Syntax

```
-usenativeversion
```

Remarks

Managed resources are usually generated from code such as the code below:

```
[assembly: AssemblyTitleAttribute( "Mycompany Custom Controls" )]
[assembly: AssemblyDescriptionAttribute( "This is a description of
```

```
the assembly" )]
#ifdef __DEBUG__
[assembly: AssemblyConfigurationAttribute( "Debug" )]
#else
[assembly: AssemblyConfigurationAttribute( "Release" )]
#endif
[assembly: AssemblyCompanyAttribute( "MyCompanyName" )]
[assembly: AssemblyProductAttribute( "MyProductName" )]
[assembly: AssemblyCopyrightAttribute( "Copyright © 2020
MyCompanyName" )]
[assembly: AssemblyTrademarkAttribute( "TM MyCompanyName" )]
[assembly: AssemblyCultureAttribute( "en-US" )]
// Version information for an assembly consists of the following
four values:
[assembly: AssemblyVersionAttribute( "2.3.1" )]
[assembly: AssemblyFileVersionAttribute( "2.3.1" )]
[assembly: AssemblyInformationalVersionAttribute( "2.3.1 Special
Build for customer Contoso" )]
```

We use this kind of attributes to generate the version info in our runtime assemblies:
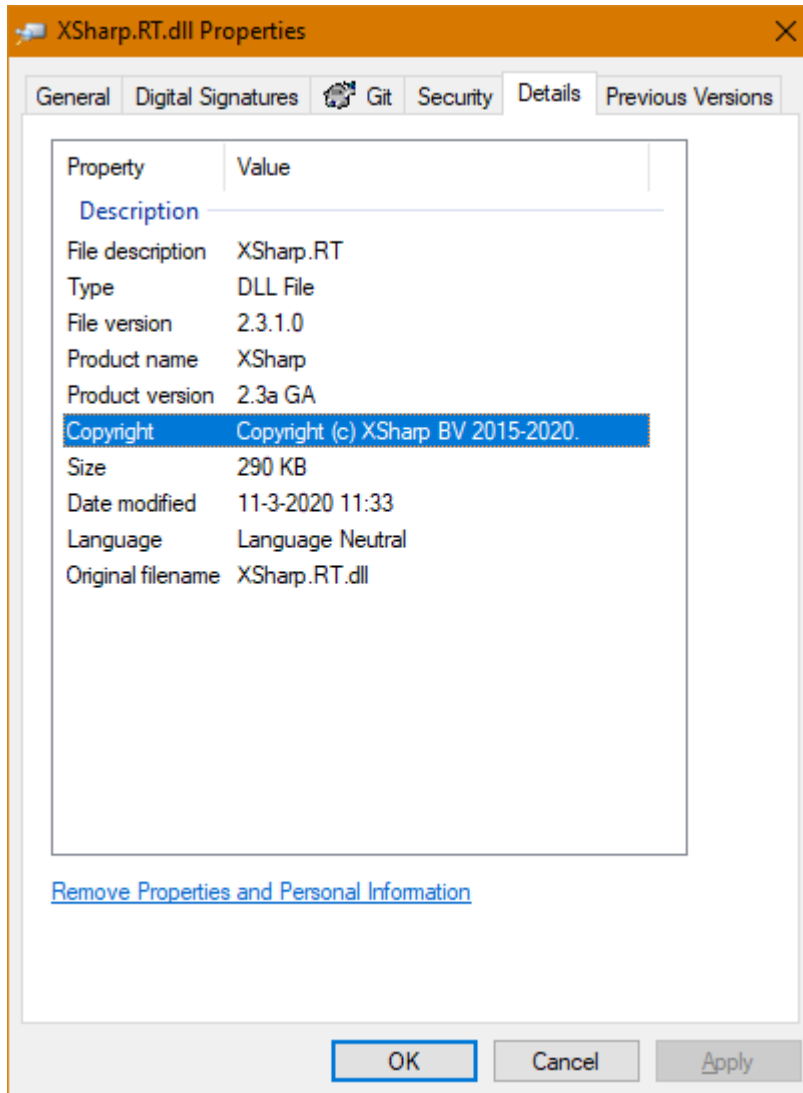
Click here to see the property page

1.9.3.83 -utf8output

The -utf8output option displays compiler output using UTF-8 encoding.

Syntax

```
-utf8output
```

Remarks

In some international configurations, compiler output cannot correctly be displayed in the console. In these configurations, use -utf8output and redirect compiler output to a file.

This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

1.9.3.84 -vo1

The -vo1 compiler option directs the compiler to treat Init() methods as constructors and Axit methods as Destructor.

Syntax

```
-vo1[+ | -]
```

Arguments

+ | -   Specifying +, or just -vo1, directs the compiler to convert Init() and Axit() methods to constructors and destructors.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. Click here to see the property page

1.9.3.85 -vo10

The -vo10 option enables the IIF operator to behave in a manner compatible with Visual Objects

Syntax

```
-vo10[+|-]
```

Arguments

+ | -   Specifying +, or just -vo10, causes the IIF operator to behave as it does in Visual Objects.

Remarks

If the true and false expressions in an IIF operator are not the same, or one cannot be implicitly converted to the type of the other, the compiler will raise an error.

Visual Objects allows this, and implicitly converts both expressions to USUAL, causing the IIF expression to also return USUAL. This is neither safe or efficient, but code originally written in Visual Objects may depend on this behavior, and if -vo10 is not used, errors may occur at runtime.

```
LOCAL x as LOGIC
x := TRUE
? IIF( x, 1, "Sunday")
```

When you use The -vo10 compiler option then the compiler will convert this to:

```
? IIF( x, (USUAL)1, (USUAL) "Sunday")     // for the VO and
Vulcan dialect
```

or

```
? IIF( x, (OBJECT)1, (OBJECT) "Sunday")     // for the Core
dialect
```

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. Click here to see the property page

1.9.3.86 **-vo11**

The -vo11 option enables Visual Objects compatible arithmetic conversions.

Syntax

```
-vo11[+|-]
```

Arguments

+ | -   Specifying +, or just -vo11, directs the compiler to emit code
                               that performs arithmetic conversions that are compatible
                               with Visual Objects.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. [Click here to see the property page](#)

Example

```
// using default or /vo11-
FUNCTION Start() AS VOID
LOCAL f AS FLOAT

f := 1.6
? (INT)f // 1 (rounding towards zero)

f := 1.5
? (INT)f // 1 (rounding towards zero)

f := 2.5 // 2 (rounding towards zero)
? (INT)f

f := 3.5 // 3 (rounding towards zero)
? (INT)f

RETURN


// using /vo11 or /vo11+
FUNCTION Start() AS VOID
LOCAL f AS FLOAT

f := 1.6
? (INT)f // 2 (rounding towards closest integer)

f := 1.5
? (INT)f // 2 (rounding towards closest even integer)

f := 2.5
? (INT)f // 2 (rounding towards closest even integer)

f := 3.5
? (INT)f // 4 (rounding towards closest even integer)

RETURN
```

1.9.3.87 -vo12

The -vo12 option enables Clipper compatible integer divisions.

Syntax

```
-vo12[+|-]
```

Arguments

+ | -   Specifying +, or just -vo12, directs the compiler to emit code
                                       that performs Clipper-compatible integer divisions.

Remarks

When The -vo12 option is enabled and both operands of The - (division) operator are integral values, both operands are converted to USUAL, and the return type is USUAL. The return value contained in the USUAL is:

- INT64 (UsualType() == 22), if one or both operands are greater than Int32.MaxValue and the remainder of the division is zero
- INT (UsualType() == 1), if one or both operands are INT or a smaller integral type and the remainder of the division is zero
- FLOAT (UsualType() == 3), if one or both operands are INT or a smaller integral type and the remainder of the division is not zero

The -vo12 option is enabled in projects created by the Transporter if the Compiler->Clipper Compatiblity->Integer Divisions option was enabled for the project in Visual Objects.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. [Click here to see the property page](#)

1.9.3.88 -vo13

The -vo13 option enables VO compatible string comparisons

Syntax

```
-vo13[+|-]
```

Arguments

+ | -

Specifying +, or just -vo13, tells the compiler that string comparisons are to be performed the same way as they are in VO, and in the RDDs, where the comparison depends on the SetCollation() setting. When - is specified, or -vo13 is not specified, string comparisons in code use String.Compare()

Remarks

When this option is enabled string comparisions are compatible with VO and the RDDs, and depend on SetCollation() as follows:

When collation=#WINDOWS, string comparisons make use of services provided by Windows that automatically handle foreign character sets. These string comparisons are implemented with a call to CompareStringA() function in kernel32.dll.

When collation = #CLIPPER, comparisons are performed byte by byte, using a weight table for each char. As in VO, a different table can be selected with the SetNatDll() function.
In .Net we are using the runtime DLL for the weight tables. In the runtime the nation DLLs are not physically implemented as separate files; the weighting tables are embedded as resources inside XSharp.Core.dll.

When collation = #Unicode then the comparisons will be done with the normal String.Compare() routine that uses the current culture.

When collation = #Ordinal then the comparisons will be done with the normal String.Compare() routine using an ordinal comparison (this is the fastest)

When -v013 is not enabled string comparisons are performed using String.Compare(), where the comparison makes use of culture-sensitive sort rules according to the current culture selected.

The setting for -vo13 is registered with the runtime by the main application at startup. If you library is compiled with -vo13+ but the main app isn't, then the string comparisons called from the library will follow the same rules as the main app because the main app registers -vo13 setting with the runtime and the stringcomparison routines in the runtime now detect that the main app does not want to do VO compatible string comparisons. We therefore recommend that 3rd party products always enable -vo13.

Compatibility Note:

When using -vo13 string comparisons involve converting unicode strings to ansi for compatibility and consequently are slower than with String.Compare().

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.

3. Change the value.
4. [Click here to see the property page](#)

1.9.3.89 -vo14

The -vo14 option enables VO compatible handling of floating point literal numbers

Syntax

```
-vo14[+|-]
```

Arguments

+ | -   Specifying +, or just -vo14, tells the compiler treat floating
                                     point literal numbers (for example: 123.456) as FLOAT data
                                     type, instead of as REAL8 (System.Double), which is the
                                     default.

Remarks

When this option is enabled, floating point literal numbers are treated by the compiler as
FLOAT values, for compatibility with existing Visual Objects code. This option is enabled
by default in transported projects.

For new code, it is strongly recommended not to enable this compiler option, as it
generates less efficient code. For example, for the following code:

```
LOCAL r := 3.0 AS REAL8
r := r * 4.5 + r + 5.5
```

if -vo14 is enabled, the compiler treats the "1.0", "1.5" and "2.5" values as numbers of type
FLOAT, causing the whole calculation to be made on FLOAT values and the result is at
the end converted to REAL8, before it is finally stored to the local variable. FLOAT is a
special data type defined in the runtime and is significantly slower than the REAL8
(System.Decimal) data type, which maps directly to a (mathematic) processor registry.
Disabling -vo14 option, would cause the above code to execute faster by a large factor.

Note that by using the "d" or "s" suffix, as in 123.456d and 123.456s, the REAL8 or REAL4
data type is being enforced on a literal number, no matter if -vo14 is enabled or disabled.
See Literals for more information.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. [Click here to see the property page](#)

1.9.3.90 -vo15

This compiler option will allow you to control what the compiler will do with local variables, parameters and return types without type specification.
The default is -vo15+ for the VO and Vulcan dialects. For the Core dialect the default is -vo15-

Syntax

```
-vo15[+|-]
```

Arguments

+ | -   Specifying +, or just -vo15, tells the compiler to treat untyped locals and return types as USUAL

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. [Click here to see the property page](#)

Example

```
FUNCTION LongSquare(nLong as LONG)      -/ Note that the return
type is missing

RETURN nLong * nLong
```

In VO/Vulcan mode this will (by default) generate a method with a USUAL return type. In Core mode this will not compile but produce a "Missing Type" error (XS1031)

When you compile with -vo15- this will also produce an error.

Similar code that will be influenced by this compiler option

```
FUNCTION MultiplyLong(nParam1 as LONG, nParam2) AS LONG -/ Note
that the type for nParam2 is missing
RETURN nParam1 * nParam2
```

And

```
FUNCTION Tomorrow() AS Date
LOCAL dToday := Today()          -/ Note that the AS DATE is
missing
RETURN dToday + 1
```

1.9.3.91 -vo16

Automatically generate Clipper calling convention constructors for classes without constructor

Syntax

```
-vo16[+|-]
```

Arguments

+ | -   Specifying +, or just -vo16, tells the compiler to automatically generate constructors with the same signature as the constructors of the parent class

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. [Click here to see the property page](#)

Example

```
CLASS Event
    EXPORT hWnd     AS PTR
    EXPORT uMsg     AS DWORD
    EXPORT wParam    AS DWORD
    EXPORT lParam    AS LONG
    EXPORT oWindow AS OBJECT
```

```
CONSTRUCTOR(_hWnd, _uMsg, _wParam, _lParam, _oWindow)
    SELF:hWnd   := _hWnd
    SELF:uMsg   := _uMsg
    SELF:wParam := _wParam
    SELF:lParam := _lParam
    SELF:oWindow := _oWindow
END CLASS


CLASS ControlEvent INHERIT Event
END CLASS
```

In the code above the compiler would generate a constructor for the ControlEvent class. This constructor will pass all the parameters to the constructor of the Event class.

The generated constructor would look like:

```
CONSTRUCTOR(_hWnd, _uMsg, _wParam, _lParam, _oWindow)
    SUPER(_hWnd, _uMsg, _wParam, _lParam, _oWindow)
```

1.9.3.92 -vo17

Generate code to fully implement the VO compatible BEGIN SEQUENCE .. END SEQUENCE.
The compiler generates calls to the runtime functions _SequenceError and _SequenceRecover that you may override in your own code.

```
-vo17[+|-]
```

Arguments

+|-   Specifying +, or just -vo17, tells the compiler to
                             automatically generate constructors with the same
                             signature as the constructors of the parent class

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. Click here to see the property page

The generated code by the compiler will call 2 runtime functions to allow you to adjust the behavior of the BEGIN SEQUENCE .. END SEQUENCE handling.

The default implementation of these functions is in the XSharp Runtime:

```
/// <summary>
/// This function is automatically inserted by the compiler in a
RECOVER USING block and gets called when the
/// RECOVER USING block is reached because of an exception.
/// </summary>
/// <param name="e">The exception that triggered the jump into the
RECOVER USING block</param>
/// <remarks>
/// The default implementation of this function (in the XSharp.RT
assembly) called the installed error handler
/// that is installed with ErrorBlock()
/// The function should then have the following prototype
/// <code language="X#">
/// FUNCTION _SequenceError(e as Exception) AS VOID
/// </code>
/// </remarks>
/// <returns>The result of the call to the error handler installed
in the ErrorBlock</returns>
/// <seelso cref='O:XSharp.RT.Functions._Break'>Break
Function</seelso>
/// <seelso cref='O:XSharp.RT.Functions.ErrorBlock'>Break
Function</seelso>
FUNCTION _SequenceError(e as Exception) AS USUAL
    LOCAL error as XSharp.Error
    IF e IS XSharp.Error VAR err
        error := err
    ELSE
        error := Error{e}
    ENDIF
    RETURN Eval(ErrorBlock(), error)
/// <summary>
/// This function is automatically inserted by the compiler in a
compiler generated
/// RECOVER USING block when you have a BEGIN SEQUENCE .. END
SEQUENCE in your code
/// without RECOVER USING clause
/// </summary>
/// <param name="u">The parameter that was passed in the BREAK
statement or the call to the _Break function</param>
/// <remarks>If a REAL exception occurs then this function is NOT
called. The function is only called when
/// the (generated) RECOVER USING block is called with a value
from a BREAK statement. <br />
```

```
/// The default implementation of this function (in the XSharp.RT
assembly) does nothing.
/// You can override this function in your own code if you want.
/// The function should then have the following prototype
/// <code language="X#">
/// FUNCTION _SequenceRecover(u as USUAL) AS VOID
/// </code>
/// </remarks>
/// <seelso cref='O:XSharp.RT.Functions._Break'>Break
Function</seealso>
FUNCTION _SequenceRecover(u as USUAL) AS VOID
    RETURN
```

1.9.3.93 -vo2

The -vo2 option directs the compiler to initialize all variables and fields of type STRING (System.String) and all elements of DIM ... AS STRING arrays to an empty string (/vo2[+|-]

Syntax

```
-vo2[+ | -]
```

Arguments

+ | -   Specifying +, or just -vo2, directs the compiler to initialize all variables and fields of type STRING and all elements of DIM ... AS STRING arrays to an empty string (String.Empty).

Remarks

This option is off by default, and all locals, globals, fields and array elements of type STRING have an initial value of NULL, which is the default initial value for any local, global, field or array element that contains a reference type.

Generally you will initialize a string variable to a specific value before it is used, and initializing it to an empty string would incur unnecessary overhead. In addition, it is inconsistent with the behavior of all other reference types, which have an initial value of NULL. However, this may break existing Visual Objects code that relies on Visual Object's behavior of initializing string variables to a string with zero characters.

When this option is not used, you may test for an empty string variable, field or array element by comparing it against NULL. When this option is enabled, you may test for an

empty string variable, field or array element by comparing it against "" or String.Empty, or testing that its length equals zero. System.String.IsNullOrEmpty() may also be used to test whether a string variable contains NULL or a valid string with zero characters.

Also note that the predefined constant NULL_STRING is normally equivalent to NULL, but when -vo2 is used, NULL_STRING is equivalent to "" (a zero-length string).

 Compatibility Note:
-vo2 does not initialize STRING fields in structures. Since structures do not have default constructors, structure fields cannot have initializer expressions. Although this is not a compatiblity issue since you cannot create structures (value types) in Visual Objects, it is something to keep in mind if you use structures in an application that uses -vo2.

Note that the use of the term "structure" here refers to STRUCTURE in X#, not STRUCTURE in Visual Objects, which has been renamed VOSTRUCT in X#. -vo2 has no effect on VOSTRUCT since you cannot declare fields that are reference types in a VOSTRUCT.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. Click here to see the property page

Example

```
// using default or /vo2-
FUNCTION Start() AS VOID
   LOCAL s AS STRING
   ? s == NULL              // true
   ? s == ""                // false
   ? s == String.Empty      // false
   ? Len(a)                 // runtime error -
NullReferenceException
   ? a:Length               // runtime error -
NullReferenceException
   ? String.IsNullOrEmpty(a) // true
   RETURN



// using /vo2 or /vo2+
FUNCTION Start() AS VOID
   LOCAL s AS STRING        // this compiles as 'LOCAL s := "" AS
STRING'
   ? s == NULL              // false
   ? s == ""                // true
   ? s == String.Empty      // true
```

```
? Len(a)                  // 0
? a:Length                // 0
? String.IsNullOrEmpty(a) // true
RETURN
```

1.9.3.94 -vo3

The -vo3 option directs the compiler to treat all methods (including ACCESS and ASSIGN methods) as virtual.

Syntax

```
-vo3[+|-]
```

Arguments

+ | -

Specifying +, or just -vo3, directs the compiler to treat all methods (including ACCESS and ASSIGN methods) as virtual, regardless of whether the VIRTUAL modifier is used or not. This provides compatibility with the Visual Objects inheritance model.

Remarks

A class method may always be explicitly declared as a virtual method by using the VIRTUAL keyword, regardless of whether -vo3 is used or not.

For performance reasons, this option is off by default. Virtual methods incur a slight performance penalty since the actual method implementation that is called cannot be determined until run-time, and depends on the run-time type of the instance on which the invocation takes place. In contrast, calls to non-virtual members can be fully resolved at compile time, and the call is always made to the compile-time type of the instance.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. Click here to see the property page

Example

```
CLASS BaseClass
METHOD WhoAmI() AS STRING
RETURN "BaseClass"
END CLASS

CLASS DerivedClass INHERIT BaseClass
METHOD WhoAmI() AS STRING
RETURN "DerivedClass"
END CLASS

FUNCTION Start() AS VOID
LOCAL c AS BaseClass
c := DerivedClass{}
? c:WhoAmI()
RETURN

// Output using default or /vo3-: BaseClass
// Output using /vo3 or /vo3+: DerivedClass
```

When The -vo3 switch is not used, the call to 'c:WhoAmI()' always resolves to the implementation in BaseClass, since the variable 'c' is typed as 'BaseClass' and 'BaseClass.WhoAmI' is a non-virtual method.

When The -vo3 switch is used, the call to 'c:WhoAmI()' resolves to the implementation in 'DerivedClass'. Even though the variable 'c' is typed as 'BaseClass', the actual type of the instance stored in 'c' at runtime determines what implementation of 'WhoAmI' to invoke since 'BaseClass.WhoAmI' is a virtual method.

The same behavior without using -vo3 could be obtained by doing:

```
VIRTUAL METHOD WhoAmI() AS STRING CLASS BaseClass
   ...
```

This is preferable over using -vo3 since you have explicit control over which methods are and are not virtual, and no unnecessary overhead is incurred where virtual inheritance is not required. However, existing Visual Objects code may not work properly without -vo3, and it may not be practical to modify existing code and add the VIRTUAL keyword to those methods that really need it.

1.9.3.95 -vo4

The -vo4 option directs the compiler to implicitly convert numeric types from larger types to smaller types but also from fractional types to integral types.

Syntax

```
-vo4[+|-]
```

Arguments

+ | -   Specifying +, or just -vo4, directs the compiler to implicitly convert signed integer values to/from unsigned values, and larger integer types to smaller integer types. This provides compatibility with Visual Objects, which permits such conversions without an explicit cast or conversion operator.

Remarks

For safety reasons, this option is off by default. Implicitly converting between signed and unsigned integer types or between larger to smaller integer types can cause numeric overflow errors at runtime or unintended values to be passed depending upon whether overflow checking is enabled or disabled. By default, you must explicitly cast a signed integer to its unsigned counterpart and from larger integer types to smaller integer types and by explicitly doing so, it is assumed that the conversion is known by the programmer to be safe.

When this option is enabled, the compiler will implicitly convert the data types listed in the table below:

From	To
SByte	BYTE, WORD, Char, DWORD
SHORT	BYTE, SByte, WORD, Char, DWORD
INT	BYTE, SByte, WORD, SHORT, Char, DWORD
INT64	BYTE, SByte, WORD, SHORT, Char, INT, DWORD, UINT64
BYTE	SByte
WORD	SByte, BYTE, SHORT, Char, INT
DWORD	SByte, BYTE, WORD, SHORT, INT
UINT64	SByte, BYTE, WORD, SHORT, Char, INT, DWORD, INT64
REAL8, REAL4, DECIMAL	All other numeric types

| FLOAT, CURRENCY | All other numeric types |

For each conversion, the compiler will raise an appropriate warning. You may disable the warning with The -wx switch, or insert an explicit cast in the source code to eliminate the warning.

It is strongly recommended that you do not use this compiler option in new code. All of the conversions listed in the table above have the ability to lose data or return incorrect values, since the range of values in the source data type cannot be represented in the target data type.

For example, an INT containing a negative number cannot be represented in a DWORD. Similarly, an INT greater than 65535 cannot be represeneted in a SHORT. If you must mix signed and unsigned types or pass a larger type to a smaller type, you should always supply an explicit cast rather than using -vo4. This will document the fact that the conversion is known to be safe, and it will not suppress compile time errors if incompatible integer types are unintentionally used.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. Click here to see the property page

Example

```
FUNCTION Start() AS VOID
LOCAL dw := 4294967296 AS DWORD
LOCAL i := -1 AS INT

DWORD_Function( i ) // no error if compiled with /vo4
INT_Function( dw ) // no error if compiled with /vo4
RETURN

FUNCTION DWORD_Function( x AS DWORD ) AS VOID
? x
RETURN

FUNCTION INT_Function( x AS INT ) AS VOID
? x
RETURN
```

1.9.3.96 -vo5

The -vo5 option directs the compiler to implicitly use the CLIPPER calling convention for functions declared with zero-arguments and no explicit calling convention.

Syntax

```
-vo5[+|-]
```

Arguments

+ | -   Specifying +, or just -vo5, directs the compiler to implicitly use the CLIPPER calling convention for functions, methods and constructors that are declared with zero arguments and no explicit calling convention.

Remarks

For compatibility with Clipper, Visual Objects uses the CLIPPER calling convention for all functions and methods that are declared with zero arguments and no explicit calling convention. The STRICT keyword may be used to override the default, and cause the function to use the STRICT calling convention.

However, in the vast majority of cases, parameters are never passed to functions and methods declared with zero arguments, and using the CLIPPER calling convention by default incurs unnecessary overhead not only in the function itself, but at every call site. In addition, the CLIPPER calling convention allows any number and type of arguments to be passed, preventing compile time error checking.

In X#, functions and methods declared with zero arguments are compiled with the STRICT calling convention by default, unless the CLIPPER keyword is explicitly specified. This behavior is the exact opposite of Visual Objects, but results in more efficient code as well as compile time error checking. Passing any arguments to a function declared to accept zero arguments will raise a compile-time error.

However, this can cause compatibility issues in code originally written in Visual Objects. The -vo5 compiler option reverses the default behavior of X# with regard to zero argument functions, so that the behavior is identical to Visual Objects.

Regardless of whether this option is enabled or not, the CLIPPER and STRICT keywords can always be used to explicitly specify the desired calling convention.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. Click here to see the property page

Example

```
FUNCTION foo() // CLIPPER
? pcount(), _getFParam( 1 )
RETURN

FUNCTION Start() AS VOID
foo( 1 )
RETURN
```

The above example will compile and run correctly if -vo5 is used, or if the CLIPPER keyword is added at the end of the FUNCTION foo() declaration. Otherwise, a compiler error will be generated on the call to foo(), as well as on the calls to pcount() and _getFParam() (which are illegal in a STRICT calling convention function).

1.9.3.97 -vo6

The -vo6 option directs the compiler to resolve typed function pointers to PTR.

Syntax

```
-vo6[+|-]
```

Arguments

+ |-   Specifying +, or just -vo6, directs the compiler to resolve pointers that would resolve to typed function pointers in Visual Objects to PTR.

Remarks

X# does not supported typed function pointers. Existing Visual Objects code that declares typed function pointers will not compile in X#, unless the type is changed to PTR or IntPtr.

If this option is enabled and a pointer type cannot be resolved, the compiler will attempt to locate a function with the same name as the pointer type (without "PTR"). If found, the compiler will resolve the type to PTR. This allows existing Visual Objects code to be compiled without modification, at least as far as the variable declaration is concerned.

The pointer type may be used as an argument to CCallNative(), PCallNative() or CallManaged(), depending on the type of function the pointer points to.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. [Click here to see the property page](#)

Example

The following code is valid in Visual Objects, but will not compile in X# unless -vo6 is used.

```
FUNCTION foo( x AS INT ) AS VOID
RETURN

GLOBAL pFoo AS foo PTR
```

Using **/vo6** has the same effect as if the **GLOBAL** declaration were rewritten as:

```
GLOBAL pFoo AS PTR
```

Using -vo6 has the same effect as if the GLOBAL declaration were rewritten as:

```
GLOBAL pFoo AS PTR
```

Note that PTR does not have the same semantics as typed function pointers in Visual Objects. However, typed function pointers are primarily used as arguments to CALL(), CCALL() and PCALL() in Visual Objects, which are not supported in X#. They have been replaced with CCallNative(), PCallNative() and CallManaged(), which accept IntPtr arguments. The same end result (invoking a function indirectly through a pointer) is therefore possible in X# without typed function pointers.

1.9.3.98 -vo7

The -vo7 option directs the compiler to allow implicit casts and conversions that are allowed in Visual Objects but which would normally be illegal or require an explicit cast in X#.

Syntax

```
-vo7[+|-]
```

Arguments

+ | -   Specifying +, or just -vo7, directs the compiler to allow certain implicit casts and conversions that are allowed in Visual Objects.

Remarks

Visual Objects allows implicit casts between types with different semantics, whereas X# normally requires explicit casts in such cases.

For example, Visual Objects allows implicit conversions between integer types and pointer types. While pointers are integers, they have different semantics. Integers are numerical values and pointers are addresses representing memory locations. In addition to the difference in semantics, the size of a pointer is dependent upon the underlying platform whereas the size of an integer does not change from platform to platform (with the exception of System.IntPtr).

While is it possible (and often necessary) to cast between types with different semantics, it should always be done via an explicit cast. This not only insures that the correct conversion code is generated (if necessary), it also self-documents the fact that you are casting one type to another type that has a different meaning.

X# supports most of the casts that Visual Objects supports, but in cases where the types have different semantics, an explicit cast is usually required. However, this can cause a large number of compiler errors in existing Visual Objects code.

Using -vo7 allows the following conversions to be performed implicitly, allowing existing Visual Objects code to compile:

From	To	Operation Performed
PTR	strongly typed PTR (e.g. INT PTR)	None, the types are binary compatible. However, the code may fail at runtime if the data the pointer points to is not the correct type.
INT or DWORD	strongly typed PTR (e.g. INT PTR)	None, the types are binary compatible. However, the code may fail at runtime if the data the pointer points to is not the correct type. Note that this conversion is only allowed when the target platform is set to x86.

INT64 or UINT64	strongly typed PTR (e.g. INT PTR)	None, the types are binary compatible. However, the code may fail at runtime if the data the pointer points to is not the correct type. Note that this conversion is only allowed when the target platform is set to x64 or Itanium.
OBJECT	any other reference type	Compiler inserts an explicit cast to the target type, which may fail at runtime.
type PTR	REF type	The compiler converts the pointer into a reference. Note that even with -vo7, not all pointers can be converted to references or else it would compromise the integrity of the garbage collector.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. <u>Click here to see the property page</u>

Example

The following code is valid in Visual Objects, but will not compile in X# unless -vo7 is used, because CreateObject() returns OBJECT and there is no implicit conversion from OBJECT to a more derived type (such a conversion cannot be guaranteed to be safe, and implicit conversions are always safe).

```
CLASS foo
...
END CLASS

FUNCTION Start() AS VOID
LOCAL f AS foo
LOCAL s AS SYMBOL
s := #foo
f := CreateObject( s ) // no implicit conversion from 'OBJECT' to
'foo'
RETURN
```

Using -vo7 has the same effect as if the assignment into f were rewritten as:

```
f := (foo) CreateObject( s )
```

In either case, the resulting code is exactly the same, and the cast to foo may fail at runtime. However, the explicit cast self-documents that you expect the return from CreateObject() to contain an instance of foo.

The following example is also valid in Visual Objects, but will not compile in X# unless -vo7 is used, because the @ operator returns the address of its operand (a typed pointer) and pointers are not the same as references in X#:

```
LOCAL x AS INT
ByRef( @x )

...

FUNCTION ByRef( i REF INT ) AS VOID
i := 5
RETURN
```

The -vo7 option will automatically convert @x, which resolves to type INT PTR, into REF INT which is compatible with the function parameter. However, it is recommended that you remove the @ operator rather than use -vo7 for this purpose.

1.9.3.99 -vo8

The -vo8 option enables Visual-Objects compatible preprocessor behavior.

Syntax

```
-vo8[+|-]
```

Arguments

+ | -   Specifying +, or just -vo8, changes certain aspects of the preprocessor to behave like Visual Objects.

Remarks

Unlike Visual Objects, X# uses a file-based preprocessor which has characteristics of traditional preprocessors in languages such as C, C++ and Clipper. The -vo8 option controls the following behaviors:

• Case Sensitivity

In a traditional preprocessor, #define foo 1 and #define FOO 2 declare two separate preprocessor symbols, because preprocessor symbols are case-sensitive.

However, in Visual Objects, DEFINE foo := 1 and DEFINE FOO := 2 declare the same entity (and would cause a compiler error because of the duplicate entity declaration).

In X#, by default, that is when -vo8 is disabled (not used or specified with "-vo8-"), preprocessor symbols are always case sensitive. When -vo8 is enabled, then the case sensitivity of symbols is decided by the state of the -cs option:
- when -cs is enabled, which makes the compiler treat all identifiers and type names as case-sensitive, then also preprocessor symbols are still case-sensitive
- when -cs is disabled, then preprocessor symbols are case-insensitive and behave in the same way as in VO

So, when -vo8 is enabled and -cs is disabled, #define foo 1 and #define FOO 2 declare the same preprocessor symbol (and would cause a compiler warning because of the redefinition).

The following code is valid in Visual Objects:

```
DEFINE foo := "bar"
? Foo // "bar"
```

but the following code would raise an unknown variable error on ? Foo because the X# preprocessor is case-sensitive by default:

```
#define foo "bar"
? Foo
```

Using The -vo8 (but not the -cs) option will allow the above example to compile. An alternative to using -vo8 is to modify the code so that the case of the text you want to replace matches the case used in #define.

• #ifdef

In a traditional preprocessor, code within a #ifdef ... #endif (or #else) block is compiled if the symbol after #ifdef is defined. It does not matter what the symbol resolves to, if it resolves to anything at all.

In Visual Objects, code within a #ifdef ... #endif (or #else) block is compiled only if the symbol after #ifdef is defined, and it resolves to an expression which resolves to a logical TRUE value. In the example below, The code will print "in #else":

```
DEFINE foo := FALSE

#ifdef foo
```

```
? "in #ifdef
#else
? "in #else" // <- this code is compiled in Visual Objects
#endif
```

whereas the equivalent code in X# would print "in #ifdef":

```
DEFINE foo := FALSE

#ifdef foo
? "in #ifdef // <- this code is compiled in Vulcan.NET
#else
? "in #else"
#endif
```

When -vo8 is used, the X# preprocessor examines the value of the preprocessor symbol to determine if the symbol resolves to a logical TRUE or FALSE value. However, the X# preprocessor does not evaluate preprocessor expressions, whereas Visual Objects does. Even with -vo8 enabled, the preprocessor symbol must resolve to a single expression containing TRUE or FALSE (case-insensitive) or a numerical value.
Numerical values of 0 resolve to FALSE and all non-zero numbers resolve to TRUE.
Preprocessor symbols that resolve to expressions are not evaluated and effectively resolve to FALSE.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. Click here to see the property page

Example

```
// For the purposes of #ifdef...

// these resolve to FALSE:
#define foo FALSE
#define foo 0

// these resolve to TRUE:
#define foo True
#define foo 1
#define foo -567
```

```
// these are not processed and effectively resolve to FALSE
// and therefore are incompatible with Visual Objects:
#define foo TRUE .AND. TRUE
#define foo TRUE .OR. TRUE
#define foo 1 * 2
```

Tip:
The -ppo option is useful for debugging the output generated by the preprocessor.

1.9.3.100 -vo9

The -vo9 option prevents the compiler from raising error XS0161 when a function or method does not have any RETURN statements. It also fixes problems with incorrect return values.

Syntax

```
-vo9[+|-]
```

Arguments

+ | -   Specifying +, or just -vo9, allows functions and methods that do not have any RETURN statement to compile without raising an error.

Remarks

Visual Objects allows functions and methods whose return type is not VOID to omit RETURN statements. The return value from any such functions or methods will always be the default value for the return type.

This is illegal in X#: all functions and methods must explicitly return a value unless the return type is VOID. However, this may prevent code that was originally written in Visual Objects from compiling in X#.

If -vo9 is enabled, any non-void functions or methods that do not have any RETURN statements will raise warning XS9025 instead of error XS0106. The warning may be disabled if desired, but it is strongly recommended that you fix the code in question. If the return value is never used, then type the function or method AS VOID. Otherwise, add a RETURN statement with an appropriate return value.
This compiler option also checks for methods/functions that have a return statement without value. In that case a warning XS9026 is shown.

The final check this compiler option does it for methods that have a return value but are not expected to return anything. If that is found then a warning XS9032 is shown.

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. Click here to see the property page

Example

```
FUNCTION x( y )
? y

METHOD x( y AS INT ) AS INT
? y
```

In the first example, the return type is not specified so it defaults to USUAL, and since there is no RETURN statement the function will always returns NIL (the default value for USUAL) in Visual Objects. In the second example, since there is no RETURN statement the method will always return zero (the default value for INT) in Visual Objects.

1.9.3.101 -w

The -w compiler option is an alias for the -nowarn command line option for compatibility.

1.9.3.102 -warn

The -warn option specifies the warning level for the compiler to display.

Syntax

```
-warn:option
```

Arguments

option	The warning level you want displayed for the compilation: Lower numbers show only high severity warnings; higher numbers show more warnings. Valid values are 0-4:

Warning level	Meaning
0	Turns off emission of all warning messages.
1	Displays severe warning messages

2	Displays level 1 warnings plus certain, less-severe warnings, such as warnings about hiding class members.
3	Displays level 2 warnings plus certain, less-severe warnings, such as warnings about expressions that always evaluate to `true` or `false`.
4 (the default)	Displays all level 3 warnings plus informational warnings.

Remarks

To get information about an error or warning, you can look up the error code in the Help Index. For other ways to get information about an error or warning, see X# Compiler Errors.

Use -warnaserror to treat all warnings as errors. Use -nowarn to disable certain warnings.

-w is the short form of -warn.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build property page.
3. Modify the Warning Level property.
4. Click here to see the property page

For information on how to set this compiler option programmatically, see WarningLevel.

Example

Compile `in.prg` and have the compiler only display level 1 warnings:

```
xsc -warn:1 in.prg
```

1.9.3.103 -warnaserror

The -warnaserror+ option treats all warnings as errors

Syntax

```
-warnaserror[+|-][:warning-list]
```

Remarks

Any messages that would ordinarily be reported as warnings are instead reported as errors, and the build process is halted (no output files are built).

By default, -warnaserror- is in effect, which causes warnings to not prevent the generation of an output file. -warnaserror, which is the same as -warnaserror+, causes warnings to be treated as errors.

Optionally, if you want only a few specific warnings to be treated as errors, you may specify a comma-separated list of warning numbers to treat as errors.

Use -warn to specify the level of warnings that you want the compiler to display. Use -nowarn to disable certain warnings.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties page.
2. Click the Build property page.
3. Modify the "Warnings As Errors" property.
4. Click here to see the property page

Example

Compile `in.prg` and have the compiler display no warnings:

```
xsc -warnaserror in.prg
xsc -warnaserror:642,649,652 in.prg
```

1.9.3.104 -win32icon

The -win32icon option inserts an .ico file in the output file, which gives the output file the desired appearance in the File Explorer.

Syntax

```
-win32icon:filename
```

Arguments

filename   The .ico file that you want to add to your output file.

Remarks

An .ico file can be created with the Resource Compiler. The Resource Compiler is invoked when you compile a Visual C++ program; an .ico file is created from the .rc file.

See -linkresource (to reference) or -resource (to attach) a .NET Framework resource file. See -win32res to import a .res file.

To set this compiler option in the Visual Studio development environment

1. Open the project's Properties pages.
2. Click the Application property page.
3. Modify the Application icon property.
4. Click here to see the property page

Example

Compile in.prg and attach an .ico file rf.ico to produce in.exe:

```
xsc -win32icon:rf.ico in.prg
```

1.9.3.105 `-win32manifest`

Use The -win32manifest option to specify a user-defined Win32 application manifest file to be embedded into a project's portable executable (PE) file.

Syntax

```
-win32manifest: filename
```

Arguments

`filename`   The name and location of the custom manifest file.

Remarks

By default, the X# compiler embeds an application manifest that specifies a requested execution level of "asInvoker." It creates the manifest in the same folder in which the executable is built, typically the bin\Debug or bin\Release folder when you use Visual Studio. If you want to supply a custom manifest, for example to specify a requested

execution level of "highestAvailable" or "requireAdministrator," use this option to specify the name of the file.

Note

This option and the -win32res option are mutually exclusive. If you try to use both options in the same command line you will get a build error.
An application that has no application manifest that specifies a requested execution level will be subject to file/registry virtualization under the User Account Control feature in Windows Vista.
Your application will be subject to virtualization if either of these conditions is true:

- You use the -nowin32manifest option and you do not provide a manifest in a later build step or as part of a Windows Resource (.res) file by using The -win32res option.
- You provide a custom manifest that does not specify a requested execution level.

Visual Studio creates a default .manifest file and stores it in the debug and release directories alongside the executable file. You can add a custom manifest by creating one in any text editor and then adding the file to the project. Alternatively, you can right-click the Project icon in Solution Explorer, click Add New Item, and then click Application Manifest File. After you have added your new or existing manifest file, it will appear in the Manifest drop down list.

You can provide the application manifest as a custom post-build step or as part of a Win32 resource file by using the -nowin32manifest option. Use that same option if you want your application to be subject to file or registry virtualization on Windows Vista. This will prevent the compiler from creating and embedding a default manifest in the portable executable (PE) file.

Example

The following example shows the default manifest that the X# compiler inserts into a PE.

Note

The compiler inserts a standard application name " MyApplication.app " into the xml. This is a workaround to enable applications to run on Windows Server 2003 Service Pack 3.

```xml
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-
com:asm.v3">
        <requestedExecutionLevel level="asInvoker"/>
```

X#

```
        </requestedPrivileges>
      </security>
    </trustInfo>
  </assembly>
```

1.9.3.106 `-win32res`

The -win32res option inserts a Win32 resource in the output file.

Syntax

```
-win32res:filename
```

Arguments

filename   The resource file that you want to add to your output file.

Remarks

A Win32 resource file can be created with the Resource Compiler. The Resource Compiler is invoked when include native resources (.RC files) in your X# Visual studio solution

A Win32 resource can contain version or bitmap (icon) information that would help identify your application in the File Explorer. If you do not specify -win32res, the compiler will generate version information based on the assembly version. You can also include Menu and Dialog definitions in a RC file.

See -linkresource (to reference) or -resource (to attach) a .NET Framework resource file.

To set this compiler option in the Visual Studio development environment

You cannot set this compiler option inside Visual Studio. If your application has native resource files, then the build system will automatically convert this to a nativeresources.res file and include this file with The -win32res command line option in your assembly.

Example

Compile `in.prg` and attach a Win32 resource file `rf.res` to produce `in.exe`:

```
xsc -win32res:rf.res in.prg
```

1.9.3.107 -wx

The -wx compiler option is an alias for the <u>-warnaserror</u> compiler option for compatibility

1.9.3.108 -xpp1

All XPP classes inherit from the Abstract class (default = OFF)

To set this compiler option in the Visual Studio development environment:

1. Open the project's Properties page.
2. Click the Dialect tab.
3. Change the value.
4. <u>Click here to see the property page</u>

1.10   X# Compiler Errors and Warnings

The complete list of X# Compiler errors and warnings is not included in this "book like" documentation but can be found in the CHM, Visual Studio Help and/or Webhelp
The compiler warnings with numbers < 9000 are "standard" Roslyn (C#) compiler errors.
The compiler warnings with numbers > 9000 are specific X# compiler errors.

1.11 X# Tips and Tricks

Enter topic text here.

1.11.1 Installer Command Line options

Our installer was created with the product Inno Setup. We are supporting the "normal" Inno Setup command line options and the following extra options.
Some options are not installed because these are not relevant for our installer, such as /lang and /password.

Custom Setup commandline options

Option	Description
/nouninstall	This option exists since X# 2.4. This suppresses uninstalling the previous version. When you choose this you can install 2 different versions of X# side by side, assuming you choose different installation folders. Please note that if both versions install into the same Visual Studio version then the latest installation "wins". If you want to switch to another version of the VS integration or MsBuild support files you should run the "deployvs**<num>**".cmd files in the Uninst folder of the installation that you want to activate.

Standard Inno Setup commandline options

Option	Description
/help, /?	Shows a summary of this information. Ignored if the UseSetupLdr [Setup] section directive was set to no.
/silent, /verysilent	Instructs Setup to be silent or very silent. When Setup is silent the wizard and the background window are not displayed but the installation progress window is. When a setup is very silent this installation progress window is not displayed. Everything else is normal so for example error messages during installation are displayed and the startup prompt is (if you haven't disabled it with DisableStartupPrompt or the '/SP-' command line option explained above). If a restart is necessary and the '/norestart command isn't used (see below) and Setup is silent, it will display a Reboot now? message box. If it's very silent it will reboot without asking.
/suppressmsgboxes	Instructs Setup to suppress message boxes. Only has an effect when combined with '/silent or '/verysilent. The default response in situations where there's a choice is: • Yes in a 'Keep newer file?' situation. • No in a 'File exists, confirm overwrite.' situation. • Abort in Abort/Retry situations.

	• Cancel in Retry/Cancel situations.
	• Yes (=continue) in a DiskSpaceWarning/DirExists/DirDoesntExist/NoUninstallWarning/ExitSetupMessage/ConfirmUninstall situation.
	• Yes (=restart) in a FinishedRestartMessage/UninstalledAndNeedsRestart situation.
	• The recommended choice in a PrivilegesRequiredOverridesAllowed=dialog situation.
	5 message boxes are not suppressible:
	  ○ The About Setup message box.
	  ○ The Exit Setup? message box.
	  ○ The FileNotInDir2 message box displayed when Setup requires a new disk to be inserted and the disk was not found.
	  ○ Any (error) message box displayed before Setup (or Uninstall) could read the command line parameters.
	  ○ Any task dialog or message box displayed by [Code] support functions TaskDialogMsgBox and MsgBox.

/log="filename"	This allows you to specify a fixed path/filename to use for the log file. If a file with the specified name already exists it will be overwritten. If the file cannot be created, Setup will abort with an error message. If you do not specify this command line option then the installer will log to a file with the name "Setup Log <date>#<number>.txt" in the current users TEMP folder.
/nocancel	Prevents the user from canceling during the installation process, by disabling the Cancel button and ignoring clicks on the close button. Useful along with '/silent or '/verysilent.
/norestart	Prevents Setup from restarting the system following a successful installation, or after a Preparing to Install failure that requests a restart. Typically used along with /silent or /verysilent.
/restartexitcode=exit code	Specifies a custom exit code that Setup is to return when the system needs to be restarted following a successful installation. (By default, 0 is returned in this case.) Typically used along with /norestart.
/closeapplications	Instructs Setup to close applications using files that need to be updated by Setup if possible.
/nocloseapplications	Prevents Setup from closing applications using files that need to be updated by Setup. If /closeapplications was also used, this command line parameter is ignored.
/forcecloseapplications	Instructs Setup to force close when closing applications.
/noforcecloseapplications	Prevents Setup from force closing when closing applications. If /forcecloseapplications was also used, this command line parameter is ignored.

/restartapplications	Instructs Setup to restart applications if possible. Only has an effect when combined with '/closeapplications.
/norestartapplications	Prevents Setup from restarting applications. If /restartapplications was also used, this command line parameter is ignored.
/loadinf="filename"	Instructs Setup to load the settings from the specified file after having checked the command line. This file can be prepared using the '/saveinf=' command as explained below.Don't forget to use quotes if the filename contains spaces.
/saveinf="filename"	Instructs Setup to save installation settings to the specified file. Don't forget to use quotes if the filename contains spaces.
/dir="x:\dirname"	Overrides the default directory name displayed on the Select Destination Location wizard page. A fully qualified pathname must be specified. May include an "expand:" prefix which instructs Setup to expand any constants in the name. For example: '/dir=expand:{autopf}\My Program'.
/group="folder name"	Overrides the default folder name displayed on the Select Start Menu Folder wizard page. May include an "expand:" prefix, see '/dir='. If the [Setup] section directive DisableProgramGroupPage was set to yes, this command line parameter is ignored.
/noicons	Instructs Setup to initially check the Don't create a Start Menu folder check box on the Select Start Menu Folder wizard page.
/type=type name	Overrides the default setup type. If the specified type exists and isn't a custom type, then any /components parameter will be ignored. ***The types in the X# installer are: full, compact, custom***
/components= "comma separated list of component names"	Overrides the default component settings. Using this command line parameter causes Setup to automatically select a custom type. If no custom type is defined, this parameter is ignored. Only the specified components will be selected; the rest will be deselected. If a component name is prefixed with a "*" character, any child components will be selected as well (except for those that include the dontinheritcheck flag). If a component name is prefixed with a "!" character, the component will be deselected. This parameter does not change the state of components that include the fixed flag.

Example:Deselect all components, then select the "help" and "plugins" components:
/components="help,plugins"
Example:Deselect all components, then select a parent component and all of its children with the exception of one:
/components="*parent,!parent\child"

The components in the X# installer are:

Component	Description
main	*The XSharp Compiler and Build System*
main\script	*Register .prgx as X# Script extension*
main\ngen	*Optimize performance by generating native images*
main\gac	*Register runtime DLLs in the GAC (recommended!)*
main\examples	*Install the Xsharp Examples*
main\debug	*Install debug versions of the runtime*
main\netcore	*Install the .Net Core version of the compiler*
vs	*Visual Studio Integration*
vs\1 thru vs\6	*The detected versions of Visual Studio*
xide	*Include the XIDE installer*

1.11.2 UnInstaller Command Line options

The Uninstaller (which can be found in the <Installdir>\Uninst folder) has the following command line options:

Option	Description
/silent, /verysilent	When specified, the uninstaller will not ask the user for startup confirmation or display a message stating that uninstall is complete. Shared files that are no longer in use are deleted automatically without prompting. Any critical error messages will still be shown on the screen. When '/verysilent is specified, the uninstallation progress window is not displayed. If a restart is necessary and the '/norestart command isn't used (see below) and '/verysilent is specified, the uninstaller will reboot without asking.
/suppressmsgboxes	Instructs the uninstaller to suppress message boxes. Only has an effect when combined with '/silent and '/verysilent.
/log	Causes Uninstall to create a log file in the user's TEMP directory detailing file uninstallation and actions taken during the uninstallation process. This can be a helpful debugging aid. The log file is created with a unique name based on the current date. (It will not overwrite or append to existing files.) The information contained in the log file is technical in nature and therefore not intended to be understandable by end users. Nor is it designed to be machine-parsable; the format of the file is subject to change without notice.
/log="filename"	Same as /log, except it allows you to specify a fixed path/filename to use for the log file. If a file with the specified name already exists it will be overwritten. If the file cannot be created, Uninstall will abort with an error message.
/norestart	Instructs the uninstaller not to reboot even if it's necessary.

1.11.3   Building XSharp apps with Visual Studio and/or MsBuild

When you build an application with MsBuild and/or Visual Studio you work with at least 2 types of files:

- The solution file (with the .sln extention)
- One or more project files. XSharp projects have the .xsproj extension. CSharp projects have the .csproj extension, Visual Basic projects the .vbproj extension.

The solution file (.sln)

The solution file is a text file with a list of project files and other information. Each project entry looks like this:

```
Project("<language guid>") = "<ProjectName>", "<Path and filename
of the project file>", "<project guid>"
EndProject
```

The <language guid> is always "{AA6C8D78-22FF-423A-9C7C-5F2393824E04}" for X# projects. This tells Visual Studio which project system to use to open the project file. The <project guid> is a generated and should match the project GUID that is defined inside the .xsproj file. These guids are also used in other sections of the .sln file.

Other language guids that you may see are {FAE04EC0-301F-11D3-BF4B-00C04F79EFBC} and {9A19103F-16F7-4668-BE54-9A1E7A4F7556} for C# and {2150E333-8FDC-42A3-9474-1A3956D46DE8} for subfolders in your solution. There are many more guids of course.

Solution files also contain sections that describe the various configurations that are available for the solution (such as "Debug" and "Release") and a section that maps solution configurations to project configurations and sometimes also a section that indicates how the source code control bindings for each of the projects are.

For the actual build process of your X# apps we can ignore the solution file for now. Solution files are "language agnostic". The building is done based on information in the project file.

The XSharp project file (.xsproj)

The project file contains all the instructions that are needed to build a X# project with MsBuild. The file is a Text file and contains XML contents in a specific format that MsBuild understands.
The file contains all the settings that you can set from the project properties dialogs in Visual Studio and also a list of the items (prg files, resx files, rc files etc) in the project.

The file uses some common information that is installed in a MsBuild subfolder inside the XSharp installation folder that belongs to the Visual Studio version that you are using.

The most important pieces in the file are for now:

Item	Description

`<Import Project="$(MSBuildExtensionsPath)\XSharp\XSharp.Default.props" />`	This imports default settings for XSharp from the XSharp folder inside the current MsBuild folder. This file contains several default values for XSharp and also imports default values from a common file delivered by Microsoft (Microsoft.Common.props)
Several `<PropertyGroup>` sections.	These sections contain values for the several options that you can find on the Project Properties dialog in Visual Studio. Some values are for all configurations, some values are configuration specific. These settings will be transformed to command line options for the X# compiler.
One or more `<ItemGroup>` sections with `<Reference>` items	The `<Reference>` items describe so called Assembly references that your project has. Usually you will find something like `<Reference Include="System" />` in there. Reference Items may also contain more information such as a version number etc. These references will be converted to [-reference](#) command line options for the compiler
One or more `<ItemGroup>` sections with `<ProjectReference>` items	The `<ProjectReference>` items describe so called Project References to other projects inside the same solution. MsBuild will determine the build order inside the solution based on the various project references and will try to build the referenced projects first before the projects referencing them. MSBuild will include a reference to the file produced by the project reference when building the command line for the compiler.
One or more `<ItemGroup>` sectins with `<COMReference>` items	The `<COMReference>` items describe references to COM components. This may be automation servers (such as Word or Excel for example) or ActiveX components (like the Shell Explorer that we use in the [email example](#)). Automation Servers will have a single COMReference with a `<wrappertool>` child node of type "tlbImp". ActiveX controls will have 2 COMReferences. One with the wrappertool set to "tlbimp" and another with the wrappertool set to "aximp". See the section below on how this is processed by MSBuild.
One or more `<ItemGroup>` sections with `<Compile>` items	The `<Compile>` items describe the source code items for the X# compiler. The template for the Console application has 2 of these items:

	`<Compile Include="Properties\AssemblyInfo.prg" />` `<Compile Include="Program.prg" />` The \<Compile\> items may have an optional \<SubType\>child node with the value "Code", "Form" or "UserControl". This subtype is ignored by the build process but used by Visual Studio to determine the icon that is shown before the item in the tree and to determine which editor to open when the item is double clicked. "Code" opens the source code editor by default. The other 2 types open the Windows Forms editor.
One or more \<ItemGroup\> sections with \<VOBinary\>, \<NativeResource\>,\<EmbeddedResource\> and other types of items	\<NativeResource\> items are handled specially by the X# build process. These are combined together in an unmanaged resource. See below \<EmbeddedResource\> files are managed resources. These are handled by MSBuild. How this works is one of the things that is described in a file that is included below
`<Import Project="$(MSBuildExtensionsPath)\XSharp\XSharp.targets" />`	This file tells MSBuild how to handle the \<Compile\> and \<NativeResource\> items in the project file and also (indirectly) imports a file Microsoft.Common.targets that tells MSBuild how to handle XAML files and how to compile \<EmbeddedResources\>.

How does MSBuild locate referenced assemblies

When locating the referenced assemblies needed for compiling your project it looks at the following:

1. When the reference node has a "hintpath" then it tries to locate the file through this path. That could look like `<HintPath>..\\SDK_Defines.dll</HintPath>`
2. When the reference node is a "normal" .Net framework assembly, it looks at the folder on your file system that matches the framework version. For example when the framework version of your project is 4.6 (there will be a node `<TargetFrameworkVersion>v4.6</TargetFrameworkVersion>` then it will look for System.DLL in the folder c:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.6
3. When the reference node is not a standard .Net framework assembly and the 3rd party vendor has registered a folder in a specific location then MsBuild will use that location. X# registers a folder in the `HKLM\Software\Microsoft\.NETFramework\v4.0.30319\AssemblyFoldersEx\XSharp` location. File registered in a location like this are also shown in the "Add References" dialogs in Visual Studio.
4. Finally (during building) MSBuild will look in the Global Assembly Cache (GAC).

Please note that there is a difference between Compiling and Running. During the compilation the files in the reference folders have preference over files in the GAC. When

X#

running the app the GAC is used and/or the local folder or path. Files in the reference folders are NEVER used at runtime. This would also not be possible since these files have no executable code in them.

The idea behind this division is that you can have a newer Framework version installed (for example 4.8) then you would like to compile against (for example 4.6). The reference assembly in the 4.6 folder only contains the subset of the api that was available for .Net Framework 4.6. So you will not be able to (accidentally) use methods or types that were added after framework 4.6, even though these methods and/or types are installed in the GAC.

How does MSBuild locate project references

When MSBuild delects a project reference then it tries to build that project first. When the project is successfully compiled then the output assembly from that project is included as "normal" reference to the X# compiler.

How are COM references handled

COM references require special processing. MSBuild uses 2 command line tools to extract the type libraries from these COM references and produces .Net assemblies (so called Interop assemblies) that describe the COM references. There are 2 tools involved:
tlbimp.exe for automation servers
aximp.exe for ActiveX controls
In our [email example](#) we are using the Shell.Explorer Active X. These 2 tools produce the files Interop.SHDocVw.dll and AxInterop.SHDocVw.dll . The AxInterop file describes the Windows Forms control and the Interop file the automation interface. In our [Excel example](#) we are referencing a "precompiled" assembly for Office and therefore we are not generating a new interop assembly but we are linking to a so called "Primary Interop Assembly (PIA)", with the name Microsoft.Office.Interop.Excel.dll .
If you include a COM component but you are not actually creating the COM objects but only consuming them then you can also set the "EmbedInteropTypes" option to true. When you do that then the X# compiler will copy the relevant information from the interop assembly and include that exe or dll, so you do not have to distribute the interop.dll with your application. In the Excel example that will not work because we are creating an excel application. The compiler will complain then "`error XS1752: Interop type 'Microsoft.Office.Interop.Excel.ApplicationClass' cannot be embedded. Use the applicable interface instead.`"
The resulting interop assemblies are produced before the compiler is called and are passed to the compiler as "normal" assembly references.

How does MSBuild call the Native Resource compiler

When your application contains Native resources then we must compile these native resources before the X# compiler can be used, since the result of the resource compilation must be included in the final exe/dll file. Of course MSBuild does not "know" about X#, so we have to tell it what to do. The instructions for this are stored in the XSharp.Targets file.
This file contains the following instructions:

```
<UsingTask TaskName="NativeResourceCompiler"
AssemblyFile="$(MSBuildThisFileDirectory)XSharp.Build.dll" />
<NativeResourceCompiler> .... </NativeResourceCompiler>
```

The first entry tells the compiler that there is a special DLL in the XSharp folder with the name XSharp.Build.dll. This DLL contains a type NativeResourceCompile, which is a subtype of Microsoft.Build.Utilities.ToolTask.
The second entry tells MSBuild how to pass information to this task to build the native resources.
This includes a list of all items from the project file with the itemtype <NativeResource>.
The task will then try to find the native resource compiler. To do that it looks in the registry in the following key:
- When running in 64 bit mode:
"HKEY_LOCAL_MACHINE\Software\WOW6432Node\XSharpBV\XSharp"
- When running in 32 bit mode: "HKEY_LOCAL_MACHINE\Software\XSharpBV\XSharp"
Inside this key it looks for the (string) value **XSharpPath** which is set by the installer at compile time.
When it can't find that path it defaults to "C:\Program Files (x86)\XSharp"
The task will then look for the **rc.exe** program in the Bin subfolder below that folder.
When the tool is found then this task checks for the date/time stamps of the various .rc files and compares these with the date/time stamp of the output file (NativeResources.res) in the "intermediate" folder. If the output file is older or does not exist then a command line for rc.exe is constructed and the compiler is called.
For this call we create a unique temporary rsp file in your temp folder. We are also saving the last version of this file in the file "LastXSharpNativeResourceResponseFile.Rsp".
If you want to see which information was passed to the native resource compiler you can look for this file in your temp folder.
The resulting NativeResources.res will be passed to the X# compiler later to be included in the exe/dll. For this we use the /win32res command line option of xsc.exe.

How are managed resources compiled

The compilation process for managed resources is mostly managed by MSBuild itself. It already knows how to handle these.
We do declare a task

```
<UsingTask TaskName="CreateXSharpManifestResourceName"
AssemblyFile="$(MSBuildThisFileDirectory)XSharp.Build.dll"/>
<CreateXSharpManifestResourceName> ...
</CreateXSharpManifestResourceName>
```

This task is also located in the same XSharp.Build.DLL and is used to help MSBuild to detect the right namespace for the generated resources.
The result of the managed resource compilation is that .resx files are compiled to one or more .resources file. These .resources files are then later passed to the compiler with the /resources command line option of xsc.exe.

Processing of XAML files

If you are creating a project that contains WPF windows or controls then an extra step is needed to produce the exe/dll.
In this step MSBuild produces so called `.baml` files and calls a code generator to generate source code for each XAML file.
For the WPF template 2 source files are produced:

- WPFWindow1.g.prg

- App.g.prg

These source files are automatically added to the command line for the X# compiler.
These source files contain a class declaration with a InitializeComponent() method that sets up the controls in your window. If you have named your controls then for each control with a name there will also be a field in the class and the generated Connect() method will set these fields to the control generated by the framework when the form is loaded.
App.g.prg also contains a class and a function Start() that is responsible for starting up your application.

Note: This source code is generated by a tool that we have registered in `c:\Windows\Microsoft.NET\Framework\v4.0.30319\Config\machine.config` .

```
  <system.codedom>
    <compilers>
      <compiler language="XSharp" extension=".prg"
type="XSharp.CodeDom.XSharpCodeDomProvider,XSharpCodeDomProvider,
Version=2.1.0.0, Culture=neutral,
                PublicKeyToken=ed555a0467764586,
ProcessorArchitecture=MSIL" />
    </compilers>
```

The XSharpCodeDomProvider.dll assembly is registered in the GAC and it contains a XSharpCodeGenerator type that is responsible for the code generation.
Note: this tool uses the keyword case setting that was specified in your Visual Studio options for the X# text editor.

How does MSBuild call the X# compiler

When MSBuild has successfully handled all external references and has created the "code behind" for XAML files compiled the native and managed resources then it calls the X# compiler.
Similar to how the native resource compiler is called the XSharp.Targets file also has instructions on how to call the compiler:

```
  <UsingTask TaskName="XSharp.Build.Xsc"
  AssemblyFile="$(MSBuildThisFileDirectory)XSharp.Build.dll"/>
  <Xsc> ..... </Xsc>
```

Again this describes a class in the XSharp.Build.DLL and the <Xsc> entry describes the properties of this type that need to be set.
The Xsc task looks for the xsc.exe just like how the native resource compiler does this:
- It looks for the installation location in the registry
- It defaults to the "C:\Program Files (x86)\XSharp" folder
There is one difference:

- It also looks for an environment variable "XSHARPDEV". When this environment variable exists it assumes that this is an alternate location where it can find the xsc.exe. We are using this internally so we can compiler with a newer version of the compiler than the one that is installed inside C:\Program Files (x86)\XSharp. You may use if you want to work with more than one version of the compiler on your machine.

When we can find the xsc.exe compiler then we construct the command line to the compiler. We are creating a unique temporaty RSP file in the temp folder, just like we do for the native resource compiler. We are also saving the last version of this file to the "LastXSharpResponseFile.Rsp" file in that folder.
If you have enabled the "Shared" compiler on the Build page in your project properties (this defaults to true) then we add the command line option /shared. This will tell xsc.exe to run XSCompiler.exe and pass the command line to that tool. XSCompiler.exe will continue to run in memory even after the compilation is finished and will cache type information from referenced assemblies. As a result a second compilation of the same project will usually be much faster, since all the relevant type information is already cached. Of course the compiler is smart enough to detect when a referenced DLL was changed (the reference could be generated from a referenced project) and will then reload the type information from that reference. Normally you will only see one copy of XSCompiler.exe running memory. You may see multiple copies of xsc.exe running in memory when MSBuild detects that 2 projects in the same solution are "independent" and can be compiled simultaneously.
The only situation where you might see 2 copies of XSCompiler.exe running in memory is when projects are compiled with difference settings for case sensitivity (the /cs command line option). One of the 2 copies will then have a case sensitive type cache and the other a case insensitive type cache.

Debugging MSBuild

If you want to see what MSBuild imports when compiling your xsproj file you can call MSBuild with a special commandline option. To do so open a visual studio developer command prompt and type the following:

```
msbuild -preprocess <yourproject.xsproj> > preprocessed.proj
```

The resulting preprocess.proj file will be an XML file that contains all imported instructions. You can open this inside Visual Studio. You may want to Format the document to make it a bit more readable.
You should see that all "<Import project" nodes are now converted to comments and the contents of these imported files is inserted into the preprocessed output.
Some imports had a condition that was not met and these are just in the file as comments.
The generated file is HUGE (the WPF template produces a file of over 8700 lines and some of these lines are thousands of characters wide. Almost all of the first 8600 lines of this preprocessed file are all imported.
Somewhere in this file you will see that MSBuild.
Please note that you are NOT able to build the output file. It just serves to see what MSBuild imports to create your project.

If you want to see how msbuild resolves the various references you should call msbuild from the command line and add the command line option to show detailed info.
The /target:rebuild on the next line makes sure that everything is rebuilt. If you are

compiling a project with native resources, managed resources or xaml files you should also see the logging of the tools that process this.

```
msbuild -verbosity:detailed <yourproject.xsproj> /target:rebuild
>buildlog.txt
```

1.11.4 Catching runtime errors at startup

Sometimes your program throws runtime errors at startup. These can be caused by missing assemblies and or by errors in initialization code.
These errors can be difficult to trap, since the errors occur inside code that is executed before the first line of code in your application.
Take the following example

```
GLOBAL x AS INT
GLOBAL y := 1 / x AS INT

FUNCTION Start AS VOID
 ? "Function Start"
    RETURN
```

This code will generate an exception at startup, which you can only see/read when you run the app from the commandline

```
Unhandled Exception: System.TypeInitializationException: The type
initializer for 'Application1.Exe.Functions' threw an exception.
---> System.DivideByZeroException: Attempted to divide by zero.
at Application1.Exe.Functions..cctor() in C:
\XIDE\Projects\Default\Applications\Application1\Prg\Start.prg:lin
e 4
--- End of inner exception stack trace ---
at Application1.Exe.Functions.Start()
```

The normal program flow in an X# application is this:
- Your application is started
- The DotNet framework is initialized
- The entrypoint is called. The normal entrypoint in an app is the Start function, which is converted by the compiler to a Start method in a compiler generated Functions class. The same class also has the globals and defines from your app. If one of these globals or defines contains an initialization expression that cannot be resolved at compile then these this code will be executed in the static constructor of the Functions class (in the error message above this is called the "type initializer").
- The code above clearly makes a mistake which causes a divide by zero error.

To intercept this we would like to run some other code at startup and add a try .. catch construct to make sure we can catch this kind of errors.

Add the following code:
```
CLASS MyStartupCode
 STATIC METHOD Start AS VOID STRICT
  TRY
   // Note that in the following line the name before .Exe must
   // match the file name of your EXE. In my case I am generating
Application1.exe
    Application1.Exe.Functions.Start()
  CATCH e AS Exception
```

```
   // We should probably log this to disk as well !
   Console.WriteLine("An unhandled exception has occurred")
   Console.WriteLine("==================================")
   DO WHILE e != NULL
     Console.WriteLine("Exception: "+e:Message)

     Console.WriteLine("Callstack:")
     Console.WriteLine(e:StackTrace)
     Console.WriteLine()
     e := e:InnerException
   ENDDO
   Console.WriteLine("==================================")
   Console.WriteLine("Press any to close the application")
   Console.ReadLine()
 END TRY
 RETURN
END CLASS
```

You may have to change the call to Application1.Exe.Functions.Start() into something that matches your EXE name.
Now goto the General page in the application properties in VS and at the entry "Startup Object" set the value MyStartupCode.

Application	
Default Namespace	Application
Dialect	Visual Objects
Output Type	Console Application
Startup Object	MyStartupCode
Target Framework Moniker	.NETFramework,Version=v4.7.2
AssemblyName	
Assembly Name	Application
Project	
Output File	Application.exe
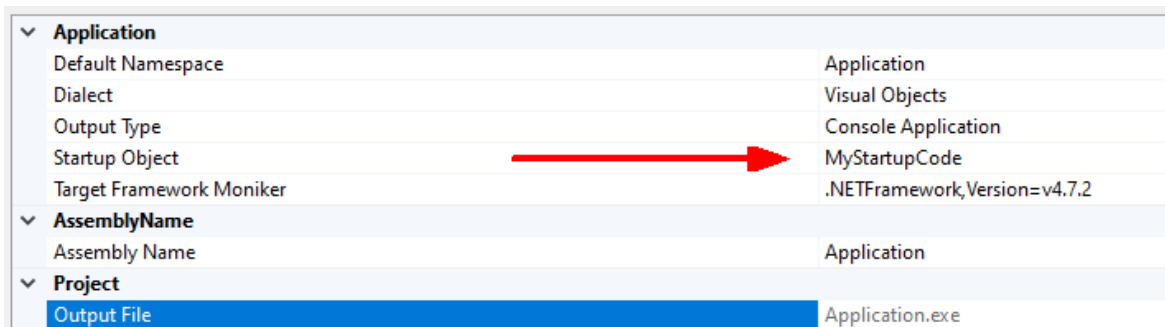
In XIDE add the command line option -main:MyStartupCode:

```
General options (all configurations)

-main:MyStartupCode
```

and run the code again.
The error is now trapped and shown.
If you app is not a Console app but a Windows app then the console output may not be visible.

Of course you can also register an UnHandledException handler in the AppDomain class inside the new startup code.
Change the code to:

```
CLASS MyStartupCode
 STATIC METHOD Start AS VOID STRICT
  TRY
    System.AppDomain.CurrentDomain:UnhandledException +=
ExceptionHandler
    Application1.Exe.Functions.Start()
  CATCH e AS Exception
    ExceptionHandler(NULL, UnhandledExceptionEventArgs{e, TRUE})
  END TRY
  RETURN

 STATIC METHOD ExceptionHandler( sender AS OBJECT, args AS
UnhandledExceptionEventArgs) AS VOID
    LOCAL e AS Exception
    LOCAL c AS STRINGe := (Exception) args:ExceptionObject
    c := "An unhandled exception has occurred"+crlf
    c += "=================================="+crlf
    DO WHILE e != NULL
       c += "Exception: "+e:Message+crlf
       c += "Callstack:"+crlf
       c += e:StackTrace+crlf
       e := e:InnerException
    ENDDO
    c += "=================================="+crlf
    MessageBox(IntPtr.Zero, c,"Error",0x60010) // MB_OK +
MB_ICONSTOP+ MB_DEFAULT_DESKTOP_ONLY + MB_TOPMOST

[DllImport("user32.dll", CharSet := CharSet.Ansi)];
STATIC METHOD MessageBox(hwnd AS IntPtr, lpText AS STRING,
lpCaption AS STRING, uType AS DWORD) AS INT PASCAL
END CLASS
```

One remark:
Do NOT use or call any Xbase types and or functions in the exception handler, since you can't be sure that the runtime was initialized properly.
If you use classes written by yourself make sure that everything is strongly typed and uses native types only. So no USUAL, FLOAT, SYMBOL etc.
And do not call code inside functions in the same app or DLLs, because again the type initializers for the classes in which these functions are located can also throw exceptions.

1.11.5 Compiler magic in the startup code

The X# compiler also does some extra "tricks" in the startup code.

XBase code may have so called INIT procedures, which contain code that will be executed at startup. For example inside the VO GUI Classes there is a procedure

```
PROCEDURE __WCInitCriticalSections() _INIT1
```

To ensure that these procedures are called at start the compiler generates one to three special methods in the functions class with the names $Init1, $Init2, $Init3 and $Exit. The VOGuiclasses assembly has two of these ($Init1 and $Init3). When you look at the contents of this method in ILSpy you see the following (using C# decompilation)

```
// VOGUIClasses.Functions
using System.Runtime.CompilerServices;

[CompilerGenerated]
public static void $Init1()
{
    __WCInitCriticalSections();
}
```

In some assemblies you will see that the $Init1() method is there but it is empty. For example in the VOSystemClasses:

```
// VOSystemClasses.Functions
using System.Runtime.CompilerServices;

[CompilerGenerated]
public static void $Init1()
{
}
```

The reason why we are creating these empty initializers is the following:
many of our VO customers are instantiating classes indirectly, by calling CreateInstance().
To be able to do so the classes have to be available at runtime.
The Roslyn compiler that we use is very smart. It does not include references to external assemblies into the exe when the exe does not reference any types or methods in that assembly.
As a result if you were calling CreateInstance(#DbServer) in your app but you were never declaring variables of type DbServer (but only of type DataServer for example) then even when you include a reference to the VORDDClasses assembly at compile time you would not have a reference to the RDDClasses in your main app.
That is why we are generating the empty $Init() methods.

When building the main application, the X# compiler checks all referenced assemblies and looks for all $Init1, $Init2, $Init3 and $Exit methods and build some code to call all these methods. As a result all referenced assemblies will be loaded at startup. You will

find this startup code in a special compiler generated method in the compiler generated <module> class:

```
// <Module>
using Application1.Exe;
using System;
using System.Runtime.CompilerServices;
using VOWin32APILibrary;
using XSharp;
using XSharp.RT;

[CompilerGenerated]
internal static void $AppInit()
{
    try
    {
        RuntimeState.AppModule =
typeof(Application1.Exe.Functions).Module;
        RuntimeState.CompilerOptionOVF = false;
        RuntimeState.CompilerOptionVO11 = false;
        RuntimeState.CompilerOptionVO13 = false;
        RuntimeState.Dialect = XSharpDialect.VO;
        VOWin32APILibrary.Functions.$Init1();
        XSharp.RT.Functions.$Init1();
        Application1.Exe.Functions.$Init1();
    }
    catch (Exception innerException)
    {
        throw new Exception("Error when executing code in INIT
procedure(s)", innerException);
    }
}
```

As you can see this code not only calls several $Init1() methods, but it also sets some properties in the X# runtime. You can also see that the code above was called in the VO Dialect.
And if you look at the generated code for the Start function this looks like this:

```
// Application1.Exe.Functions
using System;
using XSharp.RT;

public static void Start()
{
    try
    {
        <Module>.$AppInit();
```

```
        XSharp.RT.Functions.QOut("Function Start");
    }
    finally
    {
        <Module>.$AppExit();
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

The compiler has created a try finally block and calls $AppInit() to initialize the runtime
state and call the init procedures in every referenced assembly.
You also see that in the finally clause there is a call to $AppExit() from which EXIT
procedures are called (we have added these in X#) and where the garbage collector
cleans all references and waits for all finalizers to finish before your app finished.

Unfortunately this does not mean that for example all open servers will be closed. If you
have opened a DbServer and assigned the object to a global variable then this does not
automatically close the server.
There is some code in the runtime that takes care of this (but that is the subject of another
topic)..

1.11.6 Special classes and code generated by the compiler

The compiler may generate some special classes for optimization. Some of these classes are generated by Roslyn (such as the classes for Lambda expressions or the state machines for asynchronous code). Others are generated by the X# compiler. Below are some examples of these classes (that you can see if you open a X# compiler assembly with a tool such as IlSpy)

Class	Purpose

Xs$PSZLiteralsTable	This class is generated by the compiler if you have code in your application that looks like this:
	
	```LoadLibrary(PSZ(_CAST, "RICHED20.DLL")) // inside GUI classes```
	
	Since we cannot "know" at compile time what the lifetime of the PSZ must be we create a static field in this class and assign the generated PSZ value (a value type) to this field. As a result this PSZ will be "alive" during the whole lifetime of your application.
	If you know that the PSZ will not be needed after the call to the WIN32 api then you are better off replacing the PSZ(_CAST with a String2Psz(). This will ensure that the PSZ value is destroyed when the function that creates it finishes.
	The following code:
	
	```FUNCTION TestMe() AS VOID LoadLibrary(PSZ(_CAST, "RICHED20.DLL")) RETURN```
	
	generates the following
	
	```public unsafe static void TestMe() { VOWin32APILibrary.Functions.LoadLibrary((IntPtr)(void*)Xs$PSZLiteralsTable._$psz_$0); }```
	
	and the following PSZ table:
	
	```internal static class Xs$PSZLiteralsTable { internal static readonly __Psz _$psz_$0 = new __Psz("RICHED20.DLL"); }```
	
	As you can see the PSZ value is stored in the table. Please note that every PSZ variable contains a pointer to static memory allocated with the String2Mem function in the runtime. So these

static memory blocks are allocated for the whole lifetime of your application.

If you change the code to use String2Psz() instead

```
FUNCTION TestMe() AS VOID
    LoadLibrary(String2Psz("RICHED20.DLL"))
    RETURN
```

then the result will be:

```
public unsafe static void TestMe()
{
    List<IntPtr> pszList = new List<IntPtr>();
    try
    {
        VOWin32APILibrary.Functions.LoadLibrary(
(IntPtr)(void*)new
__Psz(CompilerServices.String2Psz("RICHED20.DLL"
, pszList)));
    }
    finally
    {
        CompilerServices.String2PszRelease(pszLi
st);
    }
}
```

as you can see the compiler has now generated a local variable (a list of IntPtr) which is passed to a runtime function at the end that takes care of deleting the allocated memory when the function finishes. To ensure that a try .. finally was added.

Xs$SymbolTable | This class is generated by the compiler if you are using literal symbols in your code. For each symbol in your app there will be a field in the class. Inside the System classes there are 21 literal symbols as you can see when you decompile its code:
internal static class Xs$SymbolTable

```
{
    internal static readonly __Symbol _init =
new __Symbol("INIT");
    internal static readonly __Symbol
_concurrencycontrol = new
__Symbol("CONCURRENCYCONTROL");
    internal static readonly __Symbol _notify =
new __Symbol("NOTIFY");
```

```
            .
            .
    internal static readonly __Symbol _unknown =
 new __Symbol("UNKNOWN");
    internal static readonly __Symbol
 _resourcestring = new
 __Symbol("RESOURCESTRING");
}
```

This is very similar to the way how symbols are handed in Visual Objects.
When the first symbol in an assembly is used then all the symbols are created and after that using literal symbols is very fast.
Symbols are stored in a static table in the runtime and the symbol value contains only the offset in this table. Comparing 2 symbols is like comparing 2 numbers and therefore very fast.

<AssemblyName>.Functions	Dotnet does not know the concept of functions or global variables. The X# compiler is therefore creating a static class in each assembly that contains static methods for each of the functions or procedures in your code. The name of this class is derived from the name of your output assembly: MyFile.DLL will contain a class MyFile.Functions MyFile.EXE will contain a class MyFile.Exe.Functions If your output assembly name contains embedded dots then these dots will be replaced with underscore characters in the functions class name: MyApp.Main.EXE will contain a classname MyApp_Main.EXE.Functions
Functions$<ModuleName>$	Whenever your code uses STATIC FUNCTION, STATIC DEFINE, STATIC GLOBAL (whose visibility is within the same file only) then the compiler generates a separate class for each modulde (PRG file) where the name of the PRG file is used for the <Modulename>, so the file Start.Prg in Application1.exe will result in a class name Application1.Exe.Functions$Start$
$PCall$<FunctionName>$<suffix>	If your code contains PCALL() constructs then the compiler will generate a special delegate with a name based on the method/function name and will make this delegate a nested object inside the type where the PCALL() is used. So a PCALL() inside a function will result in a nested delegate inside the Functions class and a PCALL() in a method of the Window class will result in a nested delegate inside the Windows class. The return type and parameter names and types of the delegates are derived from the function declaration for the typed pointer that you are passing to PCall(). For example The VOGUIClasses assembly contain a $PCall$DeleteTrayIcon$430 inside the Window class and a

$PCall$__InitFunctionPointer$28 inside the Functions class.
If you look at the original code in the __InitFunctionPointer procedure then it looks like this:

```
IF !PCALL(gpfnInitCommonControlsEx, @icex)
```

The resulting code looks like this:

```
if
(!$PCallGetDelegate<$PCall$__InitFunctionPointer$28>(g
pfnInitCommonControlsEx)(&icex))
```

The $PCallGetDelegate function is a special compiler generated function that looks like this:

```
[CompilerGenerated]
internal static T $PCallGetDelegate<T>(IntPtr p)
{
    return (T)(object)
Marshal.GetDelegateForFunctionPointer(p,
typeof(T));
}
```

In short: it takes a function pointer (p) and Gets a delegate of type T. This delegate is then used to call the API function.
Please don't worry if you don't get this. It took us a while to create this ourselves too !

$PCallNative$<FunctionName>$<suffix>	This is a delegate generated for PCallNative constructs. The return type is the type of the generic argument and the parameter types are derived from the types of the arguments. The parameter names are $param1, $param2 etc. So the following code inside a Test function ``` LOCAL p AS IntPtr P := GetProcAddress(hDLL, "MyFunc") PCallNative<INT> (p,1,2,3) ``` Will generate a delegate like this: ``` [CompilerGenerated] internal delegate int $PCallNative$Test$0(int $param1, int $param2, int $param3); ```
Functions.$Init1 Functions.$Init2 Functions.$Init3	These special methods inside the function class are generated to call Init and Exit procedures. See the topic about startup code for more information about this

Functions.$Exit <Module>.$AppIn it() <Module>.$AppE xit()	
<Module>.RunIni tProcs()	This special method inside the <Module> class is generated by the compiler and will be called at runtime when you are dynamically loading assemblies using the XSharpLoadLibrary() function. This takes care of calling all init procedures when a DLL is loaded dynamically.
<>ClassName	Special classes that start with a <> prefix are generated by the Roslyn compiler for lambda expressions and codeblocks. If you look at the VORDDClasses assembly you will find many examples of these. You may have to set ILSpy to show IL instead of C# or XSharp code, because otherwise these classes will be hidden by the tool. If you look at the RDD classes in C# mode it will looks like this: If you switch ILSpy to IL mode it looks like this:
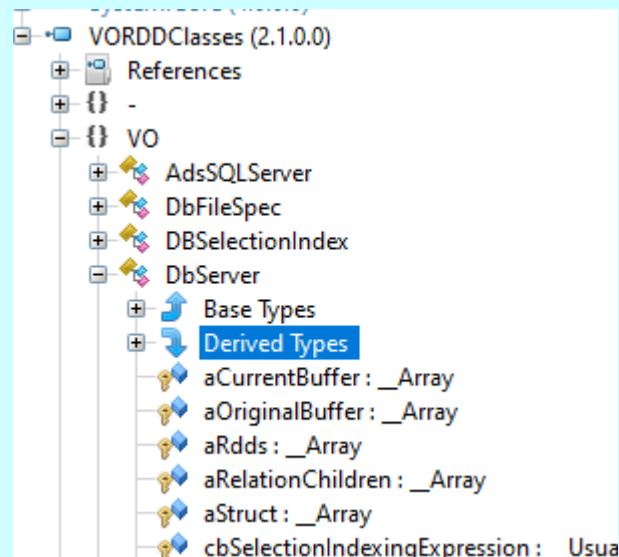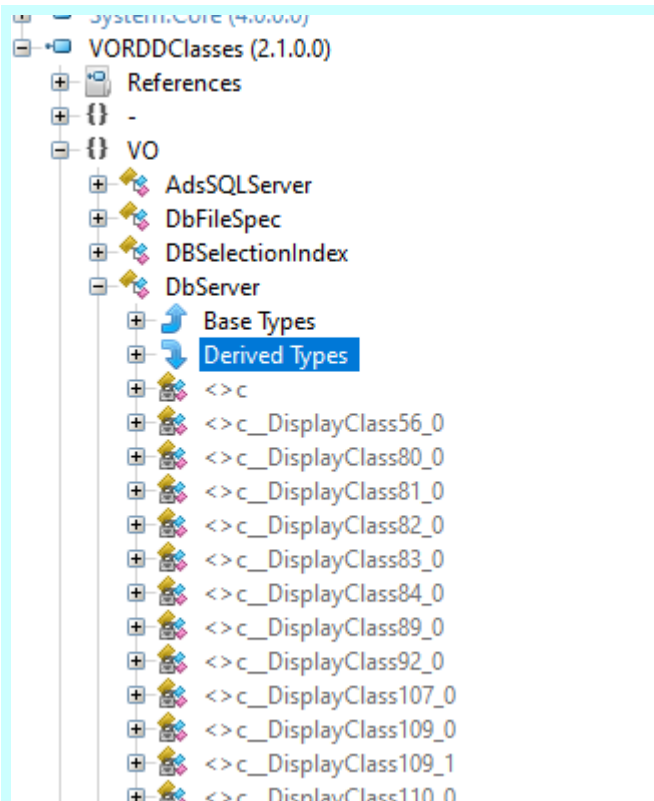
As you can see there are now quite some nested classes inside the DbServer class. The <>c class contains codeblocks that do not need to access local variables from functions or methods. In the DbServer class this class has some 25 methods, each of which is a codeblock.

The classes with the name <>c_DisplayClass<nn> contain codeblocks that need access to local variables from the function or method where they are defined.

The compiler has detected this and has moved the local variables out of the function/method and made them fields in a compiler generated class, so the codeblocks can access them. In Clipper and VO these were called "detached locals".

For example DisplayClass56_0 has the variables for the Average function:

```
.class nested private auto ansi sealed
beforefieldinit '<>c__DisplayClass56_0'
    extends [mscorlib]System.Object
{
    .custom instance void [mscorlib]
System.Runtime.CompilerServices.CompilerGenerate
dAttribute::.ctor() = (
        01 00 00 00
    )
    // Fields
```

```
        .field public int32 iCount
        .field public class [XSharp.RT]
XSharp.__Array acbExpr
        .field public class [XSharp.RT]
XSharp.__Array aResults
        .field public valuetype [XSharp.RT]
XSharp.__Usual cbKey
        .field public valuetype [XSharp.RT]
XSharp.__Usual uValue

    // Methods
    .method public hidebysig specialname
rtspecialname
        instance void .ctor () cil managed
    {

} // end of class <>c__DisplayClass56_0
```

The codeblocks inside the Average method apparently access 5 locals variables (iCount, acbExpr, aResults, cbKey and uValue).

If you look inside the Average() method of DbServer you will see a codeblock such as

```
SELF:__DBServerEval( { || iCount += 1,
__IterateForSum( acbExpr, aResults ) }.......)
```

If you look in the decompiled code for Average() (in C# mode) you will see something like this:

```
    __DbServerEval(new
                    {
                        Cb$Eval$ =
    (<>F<__Usual>)delegate
                        {
                            iCount++;
                            VORDDClasses.Functio
ns.__IterateForSum(acbExpr, aResults);
                            return
default(__Usual);
                        },
                        Cb$Src$ = "{ || iCount
+= 1, __IterateForSum( acbExpr, aResults ) }"
                    },........................
    .)
```

The whole {} after the **new** is an anonymous codeblock expression
The Cb$Eval$ field in this expression is a delegate that contains the code for the codeblock.
The CB$Src field in this expression includes the source for the codeblock so at runtime you will be able to see the source of the compiler time codeblock (this was introduced in build 2.3.0)

The actual body of the codeblock (the part from iCount++ until return default(__Usual)) is in reality stored as a method of <>c__DisplayClass56_0. And all the variables that are needed inside this codeblock are not really stored as variables inside Average() but they are stored as fields of <>c__DisplayClass56_0.

Please don't worry if you don't get this. It took us a while to understand and create this ourselves too !

Xs$Args

Whenever your code contains functions or methods with the so called CLIPPER calling convention, then X# compiler will create code that handles the parameters in a special way: For example the function Str() in the runtime. This is declared with the following parameters:

```
FUNCTION Str(nNumber,nLength,nDecimals) AS STRING
```

The compiler sees this as CLIPPER calling convention because all 3 of the parameters are optional.

The C# version of the IL code generated for this function is:

```
[ClipperCallingConvention(new string[]
{ "nNumber", "nLength", "nDecimals" })]
public static string Str([CompilerGenerated]
params __Usual[] Xs$Args)
{
    int num = (Xs$Args != null) ? Xs$Args.Length
 : 0;
    __Usual nNumber = (num >= 1) ? Xs$Args[0] :
__Usual._NIL;
    __Usual nLength = (num >= 2) ? Xs$Args[1] :
__Usual._NIL;
    __Usual nDecimals = (num >= 3) ?
Xs$Args[2] : __Usual._NIL;
```

As you can see the function now has a single argument, an array of usuals.
The parameter names are stored in an attribute of type ClipperCallingConvention. This attribute is used by the intellisense inside Visual Studio and XIDE so the parameter names can be shown.

Inside the body of the generated method the compiler now declares a variable that has the length of the array (the number of arguments passed, which you can also request at runtime with PCount()). The compiler also generates a local variable with the same name as the parameter and initializes each variable with either the value passed (0 based array elements) or with NIL.

In the body of the method you will see a try finally. In the finally clause there is the following code:

```
finally
    {
        if (num >= 2)
        {
            Xs$Args[1] = nLength;
        }
    }
```

The reason for this code is that somewhere inside Str() nLength has been assigned. Str() does not know if the variable was passed by value or by reference. If the value was passed by reference then the array element inside Xs$Args must be updated, which is exactly what happens here.

The code that calls Str() is now responsible for assigning back the value from the array to its local, when that value is passed by reference

Part II
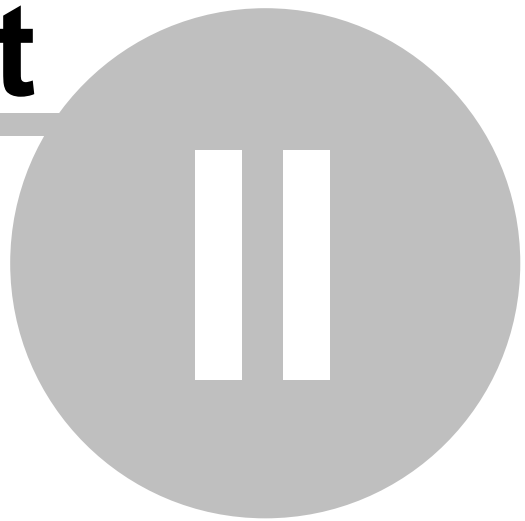
2 X# Examples

The following X# Examples will also be installed as an example solution on your machine in the Users\Public\XSharp\Examples folder.

2.1 Anonymous Method Expressions

An example of an Anonymous Method Expression (AME) (note the DELEGATE keyword):
Note that the body of the that you can have :
1. A single Expression
2. An Expression List
3. A statement List
The first 2 require the expression(s) to be on the same line as the opening Curly **{** . Of course you can use the statement continuation character **;** to tell the compiler that you have spread the statement over more than one line.
The last one requires the statements in the list to be on separate lines and the closing Curly **}** must also be on a separate line. This is shown in the example below.

```
USING System.Windows.Forms
FUNCTION Start() AS VOID
    TestAnonymous()
    RETURN

FUNCTION TestAnonymous() AS VOID
    LOCAL oForm AS Form
    oForm := Form{}
    oForm:Text := "Click me to activate the anonymous method"
    oForm:Click += DELEGATE(o AS System.Object, e AS
System.EventArgs ) {
                    System.Windows.Forms.MessageBox.Show("Click 1!")

                    System.Windows.Forms.MessageBox.Show("Click 2!")

            }
    oForm:ShowDialog()
    RETURN
```

2.2 Anonymous Types

Anonymous types are used a lot in relation to LINQ. See the LINQ example below. In the first LINQ statement an anonymous class is created

The following example shows a couple of LINQ Queries in X#

```
//references:
//System.dll
//System.Core.dll
//System.Linq.dll
USING System.Collections.Generic
USING System.Linq
USING STATIC System.Console

FUNCTION Start AS VOID
   VAR oDev    := GetDevelopers()
   VAR oC      := GetCountries()
   VAR oAll    := FROM D        IN oDev  ;
                  JOIN C        IN oC ON D:Country EQUALS C:Name
;
                  ORDERBY D:LastName  ;
                  SELECT CLASS {D:Name, D:Country,
C:Region}        // Anonymous class !
   // The type of oAll is
IOrderedEnumerable<<>f__AnonymousType0<Developer,Country>>
   // We prefer the VAR keyword!

   VAR oGreek := FROM Developer IN oDev ;
           WHERE Developer:Country == "Greece" ;
           ORDERBY Developer:LastName DESCENDING ;
           SELECT Developer
   // The type of oGreek is IOrderedEnumerable<Developer>
   // We prefer the VAR keyword!

   VAR oCount :=    FROM Developer IN oDev ;
           GROUP Developer BY Developer:Country INTO NewGroup ;
           ORDERBY NewGroup:Key SELECT NewGroup
   // The type of oCount is
IOrderedEnumerable<<IGrouping<string,Developer>>
   // We prefer the VAR keyword!

   WriteLine(e"X# does LINQ!\n")
   WriteLine(e"All X# developers (country+lastname order)\n")
   FOREACH VAR oDeveloper IN oAll
      WriteLine("{0} in {1}, {2}",oDeveloper:Name,
oDeveloper:Country, oDeveloper:Region)
   NEXT
```

```
    WriteLine(e"\nGreek X# Developers (descending lastname)\n")
    FOREACH oDeveloper AS Developer IN oGreek
        WriteLine(oDeveloper:Name)
    NEXT

    WriteLine(e"\nDevelopers grouped per country\n")

    FOREACH VAR country IN oCount
        WriteLine(i"{country.Key}, {country.Count()} developer(s)")
        FOREACH VAR oDeveloper IN country
            WriteLine("  " + oDeveloper:Name)
        NEXT
    NEXT
    WriteLine("Enter to continue")
    ReadLine()
    RETURN


FUNCTION GetDevelopers AS IList<Developer>
    // This function uses a collection initializer for the List of
Developers
    // and Object initializers for the Developer Objects
    VAR oList := List<Developer>{}    { ;
                            Developer{}{ FirstName  := "Chris",
LastName := "Pyrgas", Country := "Greece"},;
                            Developer{}{ FirstName  := "Robert",
LastName := "van der Hulst", Country := "The Netherlands"},;
                            Developer{}{ FirstName  := "Fabrice",
LastName := "Foray", Country := "France"},;
                            Developer{}{ FirstName  := "Nikos",
LastName := "Kokkalis", Country := "Greece"} ;
                            }
    RETURN oList

FUNCTION GetCountries AS IList<Country>
    // This function uses a collection initializer for the List of
Counties
    // and Object initializers for the Country Objects
    VAR oList := List<Country>{}{ ;
                            Country{} {Name := "Greece",        Region
:= "South East Europe"},;
                            Country{} {Name := "France",        Region
:= "West Europe"},;
                            Country{} {Name := "The Netherlands",
Region := "North West Europe"} ;
                            }
    RETURN oList
```

```
CLASS Developer
    PROPERTY Name        AS STRING GET FirstName + " " + LastName
    PROPERTY FirstName    AS STRING AUTO
    PROPERTY LastName     AS STRING AUTO
    PROPERTY Country      AS STRING AUTO
END CLASS

CLASS Country
    PROPERTY Name         AS STRING AUTO
    PROPERTY Region      AS STRING AUTO
END CLASS
```

2.3 ASYNC Example

```
//
// This example shows that you can call an async task and wait for
it to finish
// The result of the async task (in this case the size of the file
that has been downloaded)
// will be come available when the task has finished
// The calling code (The Start()) function will not have to wait
until the async task has
// finished. That is why the line "2....." will be printed before
the results from TestClass.DoTest()
// The sample also shows an event and displays the thread id's.
You can see that the DownloadFileTaskAsync() method
// starts multiple threads to download the web document in
multiple pieces.

USING System
USING System.Threading.Tasks

FUNCTION Start() AS VOID
    ? "1. calling long process"
    TestClass.DoTest()
    ? "2. this should be printed while processing"
    Console.ReadKey()

CLASS TestClass
    STATIC PROTECT oLock AS OBJECT      // To make sure we
synchronize the writing to the screen
    STATIC CONSTRUCTOR
       oLock := OBJECT{}

    ASYNC STATIC METHOD DoTest() AS VOID
        LOCAL Size AS INT64
        Size := AWAIT LoooongProcess()
        ? "3. returned from long process"
        ? Size, " Bytes downloaded"

    ASYNC STATIC METHOD LoooongProcess() AS Task<INT64>
        VAR WebClient := System.Net.WebClient{}
        VAR FileName := System.IO.Path.GetTempPath()+"temp.txt"
        webClient:DownloadProgressChanged += OnDownloadProgress
        webClient:Credentials :=
System.Net.CredentialCache.DefaultNetworkCredentials
        AWAIT
webClient:DownloadFileTaskAsync("http://www.xsharp.info/index.php"
, FileName)
        VAR dirInfo       :=
```

```
System.IO.DirectoryInfo{System.IO.Path.GetTempPath()}
        VAR Files         := dirInfo:GetFiles("temp.txt")
        IF Files:Length > 0
            System.IO.File.Delete(FileName)
            RETURN Files[1]:Length
        ENDIF
        RETURN 0


    STATIC METHOD OnDownloadProgress (sender AS OBJECT, e AS
System.Net.DownloadProgressChangedEventArgs) AS VOID
        BEGIN LOCK oLock
          ? String.Format("{0,3} % Size: {1,8:N0} Thread {2}",
100*e:BytesReceived / e:TotalBytesToReceive , e:BytesReceived, ;
            System.Threading.Thread.CurrentThread:ManagedThreadId)
        END LOCK
        RETURN

END CLASS
```

2.4 BEGIN UNSAFE Example

Enter topic text here.

2.5 BEGIN USING Example

```
//
// XSharp allows you to not only use the using statement to link
to namespaces
// You can also link to a static class and call the methods in
this class as if they are functions.
// The functions WriteLine and ReadKey() in the following code are
actually resolved as System.Console.WriteLine()
// and System.Console.ReadKey()
// Finally there is also the BEGIN USING .. END USING construct
which controls the lifetime of a variable
// At the end of the block the Variable will be automatically
disposed.
USING System
USING STATIC System.Console

FUNCTION Start() AS VOID
      WriteLine("Before Using Block")
   WriteLine("------------------")
   BEGIN USING VAR oTest := Test{}
      oTest:DoSomething()
   END USING
   WriteLine("------------------")
   WriteLine("After Using Block")
   ReadKey()


CLASS Test IMPLEMENTS IDisposable
   CONSTRUCTOR()
      Console.WriteLine("Test:Constructor()")

   METHOD DoSomething() AS VOID
      Console.WriteLine("Test:DoSomething()")

   METHOD Dispose() AS VOID
      Console.WriteLine("Test:Dispose()")

END CLASS
```

2.6   CHECKED Example

```
FUNCTION Start() AS VOID
   LOCAL d AS DWORD
   LOCAL n AS INT

   d := UInt32.MaxValue
   ? "Initial value of d:", d

   BEGIN UNCHECKED
      // arithmetic operations inside an UNCHECKED block will not
produce
      // overflow exceptions on arithmetic conversions and
operations,
      // no matter if overflow checking is enabled application-
wide or not
      n := (INT)d
      ? "Value of n after conversion:", n
      d ++
      ? "Value of d after increasing it:", d
   END UNCHECKED

   d := UInt32.MaxValue
   BEGIN CHECKED
      // arithmetic operations inside a CHECKED block always do
      // overflow checking and throw exceptions if overflow is
detected
      TRY
         n := (INT)d
         d ++
      CATCH e AS Exception
         ? "Exception thrown in CHECKED operation:", e:Message
      END TRY
   END CHECKED
   Console.ReadLine()
RETURN
```

2.7   EVENT Example

```
USING System.Collections.Generic
FUNCTION Start AS VOID
    LOCAL e AS EventsExample
    e := EventsExample{}
    e:Event1 += TestClass.DelegateMethod
    e:Event1 += TestClass.DelegateMethod
    e:Event1 -= TestClass.DelegateMethod   // added 2, removed 1,
should be called once
    e:Event2 += TestClass.DelegateMethod
    e:Event2 += TestClass.DelegateMethod
    e:Event2 -= TestClass.DelegateMethod // added 2, removed 1,
should be called once
    e:Event3 += TestClass.DelegateMethod
    e:RaiseEvent1("This is a test through a multi line event
definition")
    e:RaiseEvent2("This is a test through a single line event
definition")
    e:RaiseEvent3("This is a test through an old style event
definition")
    Console.WriteLine("Press a Key")
    Console.ReadLine()


DELEGATE EventHandler (s AS STRING) AS VOID

CLASS TestClass
    STATIC METHOD DelegateMethod(s AS STRING ) AS VOID
        Console.WriteLine( s)
END CLASS


CLASS EventsExample
    PRIVATE eventsTable AS Dictionary<STRING, System.Delegate>
    PRIVATE CONST sEvent1 := "Event1" AS STRING
    PRIVATE CONST sEvent2 := "Event2" AS STRING
    CONSTRUCTOR()
        eventsTable := Dictionary<STRING, System.Delegate>{}
        eventsTable:Add(sEvent1,NULL_OBJECT)
        eventsTable:Add(sEvent2,NULL_OBJECT)

    // Multiline definition
    EVENT Event1 AS EventHandler
        ADD
            BEGIN LOCK eventsTable
                eventsTable[sEvent1] := ((EventHandler)
eventsTable[sEvent1]) + value
```

```
            END LOCK
            Console.WriteLine(__ENTITY__ + " "+value:ToString())
        END
        REMOVE
            BEGIN LOCK eventsTable
                eventsTable[sEvent1] := ((EventHandler)
eventsTable[sEvent1]) - value
            END LOCK
            Console.WriteLine(__ENTITY__+ " "+value:ToString())
        END
    END EVENT


    // Single Line defintion on multilpe lines with semi colons,
for better reading !
    EVENT Event2 AS EventHandler  ;
        ADD     eventsTable[sEvent2] := ((EventHandler)
eventsTable[sEvent2]) + value ;
        REMOVE  eventsTable[sEvent2] := ((EventHandler)
eventsTable[sEvent2]) - value

    // Old style definition
    EVENT Event3 AS EventHandler

    METHOD RaiseEvent1(s AS STRING) AS VOID
        VAR handler := (EventHandler) eventsTable[sEvent1]
        IF handler != NULL
            handler(s)
        ENDIF

    METHOD RaiseEvent2(s AS STRING) AS VOID
        VAR handler := (EventHandler) eventsTable[sEvent2]
        IF handler != NULL
            handler(s)
        ENDIF

    METHOD RaiseEvent3(s AS STRING) AS VOID
        IF SELF:Event3 != NULL
            Event3(s)
        ENDIF
END CLASS
```

2.8   Expression Examples

```
//
// This example shows various new expression formats
//
using System.Collections.Generic

Function Start() as void
   VAR oNone    := Person{"No", "Parent"}

   FOREACH VAR oValue in GetList()
      if oValue IS STRING                 // Value IS Type
         ? (String) oValue
      ELSEIF oValue IS INT
         ? (Int) oValue
      ELSEIF oValue IS DateTime
         ? (DateTime) oValue
      ELSEIF oValue IS Person
         LOCAL oPerson as Person
         oPerson := (Person) oValue
         ? oPerson:FirstName, oPerson:LastName
      // Value DEFAULT Value2 . When Value IS NULL then Value2
will be used
         oPerson := oPerson:Parent DEFAULT oNone
         ? "Parent: ", oPerson:FirstName, oPerson:LastName
      ENDIF
   NEXT
   LOCAL oEmptyPerson as Person
   LOCAL sName as STRING
   oEmptyPerson := GetAPerson()
   sName    := oEmptyPerson?:FirstName        // Conditional
Access: This will not crash, even when Person is a NULL_OBJECT
   ? sName DEFAULT "None"
   Console.ReadLine()
   RETURN

FUNCTION GetList() AS List<OBJECT>
   VAR aList := List<OBJECT>{}
   aList:Add(DateTime.Now)
   aList:Add("abcdefg")
   aList:Add(123456)
   VAR oPerson := Person{"John", "Doe"}
   aList:Add(oPerson)
   VAR oChild := Person{"Jane", "Doe"}
   oChild:Parent := oPerson
   aList:Add(oChild)
   RETURN aList
```

```
CLASS Person
   EXPORT FirstName   AS STRING
   EXPORT LastName      as STRING
   EXPORT Parent       as Person
   CONSTRUCTOR(First as STRING, Last as STRING)
      FirstName  := First
      LastName   := Last

END CLASS

FUNCTION GetAPerson() as Person
   RETURN NULL_OBJECT
```

2.9   FIXED Example

The new FIXED modifier and BEGIN FIXED .. END FIXED keywords allow you to tell the
.Net runtime that you do not want a variable to be moved by the Garbage collector.

```
UNSAFE FUNCTION Start AS VOID
    VAR s := "SDRS"
    BEGIN FIXED LOCAL p := s AS CHAR PTR
        VAR i := 0
        WHILE p[i] != 0
            p[i++]++
        END
    END FIXED
    Console.WriteLine(s)
    Console.Read()
    RETURN
```

As you can see the BEGIN FIXED statement requires a local variable declaration. The
contents of this local (in the example above a CHAR PTR) will be excluded from garbage
collection inside the block.

Please note:
The FIXED keyword and the example above should be used with extreme care. Strings in
.Net are immutable. You normally should not manipulate strings this way !

2.10   GENERICs Example

Stack Example
This example shows that we can now create generic classes with X# !
In the Stack class the T parameter will be replaced with a type at compile time.

```
/*
Stack Example - Written by Robert van der Hulst
This example shows that we can now create generic classes with X#
!
Note: Compile with the /AZ option
*/

USING System.Collections.Generic
USING STATIC System.Console

FUNCTION Start AS VOID
    LOCAL oStack AS Stack<INT>
    LOCAL i AS LONG
    TRY
    oStack := Stack<INT>{25}
        WriteLine("Created a stack with {0} items",oStack:Capacity)
        WriteLine("Pushing 10 items")
        FOR I := 1 TO 10
            oStack:Push(i)
        NEXT
        WriteLine("Popping the stack until it is empty")
    i := 0
    WHILE oStack:Size > 0
        i += 1
        WriteLine(oStack:Pop())
    END
    WriteLine("{0} Items popped from the stack",i)
    WriteLine("Press Enter")
    ReadLine()
    WriteLine("The next line pops from an empty stack and throws an
exception")
    ReadLine()
    WriteLine(oStack:Pop())
    CATCH e AS Exception
        WriteLine("An exception was catched: {0}", e:Message)
    END TRY
    WriteLine("Press Enter to Exit")
    ReadLine()
    RETURN

CLASS Stack<T>  WHERE T IS STRUCT, NEW()
    PROTECT _Items        AS T[]
```

```
PROTECT _Size        AS INT
PROTECT _Capacity    AS INT
PROPERTY Size        AS INT GET _Size
PROPERTY Capacity    AS INT GET _Capacity

CONSTRUCTOR()
    SELF(100)

CONSTRUCTOR(nCapacity AS INT)
    _Capacity := nCapacity
    _Items := T[]{nCapacity}
    _Size  := 0
    RETURN

PUBLIC METHOD Push( item AS T) AS VOID
    IF _Size >= _Capacity
        THROW StackOverFlowException{}
    ENDIF
    _Items[_Size] := item
    _Size++
    RETURN

PUBLIC METHOD Pop( ) AS T
     _Size--
    IF _Size >= 0
        RETURN _Items[_Size]
    ELSE
        _Size := 0
        THROW Exception{"Cannot pop from an empty stack"}
    ENDIF
END CLASS
```

2.11   Lamda Expressions

Lamda Expressions are very much like CodeBlocks, but the difference is that it has optional typed parameters and **return** values.
You can also specify the parameter type **in** the parameter list **as** the 3rd example shows

```
DELEGATE Multiply(x AS REAL8) AS REAL8
FUNCTION Start AS VOID
   LOCAL del AS Multiply
   del := {e => e * e}
   ? del(1)
   ? del(2)
   ? del(3)
   ? del(4)
   Console.ReadLine()
LOCAL dfunc AS System.Func<Double,Double>
   dfunc :=   {x =>
            ? "square of", x
            RETURN x^2
         }
   ? dfunc(5)
LOCAL typed AS Multiply
   typed :=   {x  AS REAL8 =>
            ? "square of", x
            RETURN x^2
         }
   ? typed(6)
RETURN
```

2.12 LINQ Example

The following example shows a couple of LINQ Queries in X#

```
//references:
//System.dll
//System.Core.dll
//System.Linq.dll
USING System.Collections.Generic
USING System.Linq
USING STATIC System.Console

FUNCTION Start AS VOID
   VAR oDev   := GetDevelopers()
   VAR oC     := GetCountries()
   VAR oAll   := FROM D        IN oDev  ;
                 JOIN C        IN oC ON D:Country EQUALS C:Name
;
                 ORDERBY D:LastName  ;
                 SELECT CLASS {D:Name, D:Country,
C:Region}        // Anonymous class !
   // The type of oAll is
IOrderedEnumerable<<>f__AnonymousType0<Developer,Country>>
   // We prefer the VAR keyword!

   VAR oGreek := FROM Developer IN oDev ;
          WHERE Developer:Country == "Greece" ;
          ORDERBY Developer:LastName DESCENDING ;
          SELECT Developer
   // The type of oGreek is IOrderedEnumerable<Developer>
   // We prefer the VAR keyword!

   VAR oCount :=   FROM Developer IN oDev ;
          GROUP Developer BY Developer:Country INTO NewGroup ;
          ORDERBY NewGroup:Key SELECT NewGroup
   // The type of oCount is
IOrderedEnumerable<<IGrouping<string,Developer>>
   // We prefer the VAR keyword!

   WriteLine(e"X# does LINQ!\n")
   WriteLine(e"All X# developers (country+lastname order)\n")
   FOREACH VAR oDeveloper IN oAll
      WriteLine("{0} in {1}, {2}",oDeveloper:Name,
oDeveloper:Country, oDeveloper:Region)
   NEXT

   WriteLine(e"\nGreek X# Developers (descending lastname)\n")
   FOREACH oDeveloper AS Developer IN oGreek
      WriteLine(oDeveloper:Name)
```

```
            NEXT

        WriteLine(e"\nDevelopers grouped per country\n")

        FOREACH VAR country IN oCount
            WriteLine(i"{country.Key}, {country.Count()} developer(s)")
            FOREACH VAR oDeveloper IN country
                WriteLine("  " + oDeveloper:Name)
            NEXT
        NEXT
        WriteLine("Enter to continue")
        ReadLine()
        RETURN


FUNCTION GetDevelopers AS IList<Developer>
    // This function uses a collection initializer for the List of
Developers
    // and Object initializers for the Developer Objects
    VAR oList := List<Developer>{}    { ;
                            Developer{}{ FirstName  := "Chris",
LastName := "Pyrgas", Country := "Greece"},;
                            Developer{}{ FirstName  := "Robert",
LastName := "van der Hulst", Country := "The Netherlands"},;
                            Developer{}{ FirstName  := "Fabrice",
LastName := "Foray", Country := "France"},;
                            Developer{}{ FirstName  := "Nikos",
LastName := "Kokkalis", Country := "Greece"} ;
                        }
    RETURN oList

FUNCTION GetCountries AS IList<Country>
    // This function uses a collection initializer for the List of
Counties
    // and Object initializers for the Country Objects
    VAR oList := List<Country>{}{ ;
                        Country{} {Name := "Greece",      Region
:= "South East Europe"},;
                        Country{} {Name := "France",      Region
:= "West Europe"},;
                        Country{} {Name := "The Netherlands",
Region := "North West Europe"} ;
                    }
    RETURN oList

CLASS Developer
    PROPERTY Name       AS STRING GET FirstName + " " + LastName
    PROPERTY FirstName    AS STRING AUTO
```

```
    PROPERTY LastName    AS STRING AUTO
    PROPERTY Country     AS STRING AUTO
END CLASS

CLASS Country
    PROPERTY Name        AS STRING AUTO
    PROPERTY Region      AS STRING AUTO
END CLASS
```

2.13 NOP Example

```
// The NOP keyword is an empty statement.
// This tells the compiler that there is no code missing !
FUNCTION Start() AS VOID
LOCAL i as LONG
FOR i := 1 to 10
   IF I % 2 == 0
      Console.WriteLine(i)
   ELSE
      NOP      // Nothing happens here. This tells the compiler
that there is no code missing !
   ENDIF
NEXT
RETURN
```

2.14 SWITCH Example

```
//
// The SWITCH statement is a replacement for the DO CASE statement
// The biggest difference is that the expression (in this case
sDeveloper) is only evaluated once.
// which will have a performance benefit over the DO CASE
statement
// Empty statement lists for a CASE are allowed. In that case the
labels share the code (see CHRIS and NIKOS below)
//
// Please note that EXIT statements inside a switch are not
allowed, however RETURN, LOOP and THROW are allowed.
using System.Collections.Generic

Function Start() as void
    FOREACH VAR sDeveloper in GetDevelopers()
      SWITCH sDeveloper:ToUpper()
      CASE "FABRICE"
         ? sDeveloper, "France"
      CASE "CHRIS"
      CASE "NIKOS"
         ? sDeveloper, "Greece"
      CASE "ROBERT"
         ? sDeveloper, "The Netherlands"
      OTHERWISE
         ? sDeveloper, "Earth"
      END SWITCH
    NEXT
      Console.ReadKey()
    RETURN


FUNCTION GetDevelopers as List<String>
VAR aList := List<String>{}
aList:AddRange(<string>{ "Chris", "Fabrice", "Nikos", "Robert",
"YourName" } )
RETURN aList
```

2.15   Typed Enums

Enums may now be typed in X#. The type indicates the base class for the ENUM. By specifying a AS clause you can determine the size in bytes that the ENUM uses.

```
ENUM Foo AS BYTE
    MEMBER One
    MEMBER Two
END ENUM
```

2.16 USING Example

```
//
// XSharp allows you to not only use the using statement to link
to namespaces
// You can also link to a static class and call the methods in
this class as if they are functions.
// The functions WriteLine and ReadKey() in the following code are
actually resolved as System.Console.WriteLine()
// and System.Console.ReadKey()
// Finally there is also the BEGIN USING .. END USING construct
which controls the lifetime of a variable
// At the end of the block the Variable will be automatically
disposed.
USING System
USING STATIC System.Console

FUNCTION Start() AS VOID
   WriteLine("Before Using Block")
   WriteLine("------------------")
   BEGIN USING VAR oTest := Test{}
      oTest:DoSomething()
   END USING
   WriteLine("------------------")
   WriteLine("After Using Block")
   ReadKey()


CLASS Test IMPLEMENTS IDisposable
   CONSTRUCTOR()
      Console.WriteLine("Test:Constructor()")

   METHOD DoSomething() AS VOID
      Console.WriteLine("Test:DoSomething()")

   METHOD Dispose() AS VOID
      Console.WriteLine("Test:Dispose()")

END CLASS
```

2.17   VAR Example

```
//
// The VAR keyword has been added to the language because in many
situations
// the result of an expression will be directly assigned to a
local, and the expression
// will already describe the type of the variable
// VAR is a synonym for LOCAL IMPLIED
using System.Collections.Generic

FUNCTION Start AS VOID
// In the next line the compiler "knows" that today is a DateTime
VAR today := System.DateTime.Now
? today

// In the next line the compiler "knows" that text is a String
VAR text  := Convert.ToString(123)
? text

// In the next line the compiler "knows" that s is a string
FOREACH VAR s in GetList()
    ? s
NEXT

Console.ReadLine()

RETURN

FUNCTION GetList AS List<String>
VAR aList := List<String>{}
aList:Add("abc")
aList:Add("def")
aList:Add("ghi")
return aList
```

2.18   Vulcan Runtime (BYOR)

This "Bring Your Own Runtime" (BYOR) example shows how you can use Vulcan Datatypes
and functions **in** X#.

The example does NOT come with the Vulcan Runtime DLLs. You need **to** have these installed
on your machine and you may need **to** update the references **in** the example and point them
to the Vulcan Runtime DLLs on your machine.

```
// Please read the comments in Readme.txt !
using System
using System.Collections.Generic
using System.Linq
using System.Text

Function Start() as void

    LOCAL startingColor AS ConsoleColor
    startingColor := Console.ForegroundColor

    ConsoleHeading("Bring Your Own Runtime (BYOR)
Sample",ConsoleColor.Magenta)
    Console.ForegroundColor := ConsoleColor.Gray

    DateSamples()
    Conversion()
    Strings()
    Numeric()
    Wait()
    Arrays()
    Symbols()
    LateBinding()
    Wait()

    Workarea()
    Wait()
    Macros()
    Wait()
    CodeBlocks()
    Wait()

    Console.ForegroundColor := startingColor

FUNCTION ConsoleHeading(s AS STRING, c := ConsoleColor.Yellow AS
ConsoleColor) AS VOID
```

X#

```
        LOCAL originalColor AS ConsoleColor
        originalColor := Console.ForegroundColor

        Console.ForegroundColor := c
        Console.WriteLine(s)
        Console.ForegroundColor := originalColor

FUNCTION Wait() AS VOID
    ConsoleHeading("Press any key to
continue...",ConsoleColor.Green)
    Console.ReadKey()


FUNCTION DateSamples() AS VOID
    ConsoleHeading("Dates")
    LOCAL dToday := Today() AS DATE
    Console.WriteLine( i"Today is {dToday:d}"  )  // Interpolated
string, short Date Format
    Console.WriteLine( i"Today is {dToday:G}"  )  // Interpolated
string, General Date Format

    LOCAL dTomorrow AS __VODATE
    dTomorrow := dToday + 1  // can perform VO date arithmetic
    Console.WriteLine( i"Tomorrow is {dTomorrow:D}") // Long date
format

    LOCAL dNewYear AS DATE
     dNewYear := 2016.01.01

     Console.WriteLine("New year was " +AsString(dNewYear)+ " on a
"+CDOW(dNewyear))

FUNCTION Conversion() AS VOID
    ConsoleHeading("Conversion")
    LOCAL sToday AS STRING
    sToday := DTOS(Today()) // Convert date using VO function
    Console.WriteLine(i"Today is {sToday}")

FUNCTION Strings() AS VOID
    ConsoleHeading("Strings")
    LOCAL s AS STRING
    LOCAL sToday := DTOS(Today())
    s := SubStr(sToday,1,4) // string manipulation
    Console.WriteLine(String.Format("SubStr is {0}",s))

    LOCAL n AS DWORD
    n := At("0",s)
    Console.WriteLine(String.Format("At is {0}",n))
```

```
    LOCAL c := "" AS STRING
    Console.WriteLine(String.Format("c is
{0}",IIF(Empty(c),"Empty","Not empty")))
    c := "x"
    Console.WriteLine(String.Format("c is
{0}",IIF(Empty(c),"Empty","Not empty")))
    RETURN

FUNCTION Numeric() as VOID
    ConsoleHeading("Numeric")
    LOCAL r AS REAL8
    r := Pow(2,3)
    Console.WriteLine(String.Format("Pow is {0}",r))
    LOCAL f AS FLOAT
    f := Sqrt(10)
    Console.WriteLine(String.Format("The root of 10 is {0}",f))
    RETURN

FUNCTION Arrays() AS VOID
    ConsoleHeading("Array")
    LOCAL a AS ARRAY
    a := {1,2,3}
    Console.WriteLine(String.Format("Original Array length is
{0}",ALen(a)))
    AAdd(a,4)
    AAdd(a,5)
    AAdd(a,6)
    Console.WriteLine(String.Format("Array length after adding 3
elements is {0}",ALen(a)))
    FOR VAR i := 1 TO ALen(a)
       Console.Writeline(String.Format("{0} {1}", i, a[i]))
    NEXT
    ADel(a, 1)
    Asize(a, Alen(a) -1)
    Console.WriteLine(String.Format("Array length after deleting 1
elements is {0}",ALen(a)))
    RETURN

FUNCTION Symbols() AS VOID
    LOCAL s as Symbol
    LOCAL c as STRING
    ConsoleHeading("Symbol")
    s := #LastName
    Console.WriteLine("Symbol s: "+s:ToString())
    c := Symbol2String(s)
    Console.WriteLine("Symbol converted to string : "+c)
```

```
        RETURN

FUNCTION LateBinding() AS VOID
    LOCAL e as Object
    ConsoleHeading("Late binding")
    e := Error{0, "Message"}
    Console.WriteLine("Reading property from a untype variable")
    Console.WriteLine( (String) e:Message)

    RETURN

Function Workarea() AS VOID
    LOCAL cWorkDir as STRING
    FIELD LASTNAME, FIRSTNAME in CUSTOMER
    ConsoleHeading("DBF Access")
    cWorkDir := "..\..\"
    SetPath(cWorkDir)
    DbUseArea(TRUE, "DBFNTX","Customer")
    DbSetIndex("CustNum.NTX")
    DbSetIndex("CustName.NTX")
    DbGoTop()
    DO WHILE ! EOF()
        ? Str(Recno(),3), LASTNAME, FIRSTNAME, _FIELD->CITY
        DbSkip(1)
    ENDDO
    DbCloseArea()

FUNCTION Macros() AS VOID
    LOCAL cMacro as STRING
    LOCAL cbMacro as CodeBlock
    ConsoleHeading("Macros")
    cMacro := "{||Today()}"
    cbMacro := &(cMacro)
    ? cMacro, Eval(cbMacro)
    cMacro := "1+2+3"
    ? cMacro, &cMacro
    cMacro := "System.Int32.MaxValue"
    cbMacro := MCompile(cMacro)
    ? cMacro, MExec(cbMacro)
    cMacro := "{|a,b,c| a*b*c}"
    cbMacro := &(cMacro)
    ? cMacro, Eval(cbMacro, 2,3,4)

    ?
    RETURN

FUNCTION CodeBlocks() AS VOID
    LOCAL oBlock as CodeBlock
```

```
ConsoleHeading("Codeblocks")
oBlock := {||Today()}
? oBlock, eval(oBlock)
oBlock := {||System.Math.Pow(2,3)}
? oBlock, eval(oBlock)
oBlock := {|a,b,c|a*b*c}
? oBlock, eval(oBlock,2,3,4)
?
RETURN
```

2.19   YIELD Example

```
using System.Collections.Generic

// The Yield return statement allows you to create code that
returns a
// collection of values without having to create the collection in
memory first.
// The compiler will create code that "remembers" where you were
inside the
// loop and returns to that spot.
FUNCTION Start AS VOID
    FOREACH nYear AS INT IN GetAllLeapYears(1896, 2040)
        ? "Year", nYear, "is a leap year."
    NEXT
    Console.ReadLine()
RETURN

FUNCTION GetAllLeapYears(nMin AS INT, nMax AS INT) AS
IEnumerable<INT>
    FOR LOCAL nYear := nMin AS INT UPTO nMax
        IF nYear % 4 == 0 .and. (nYear % 100 != 0 .or. nYear % 400
== 0)
            YIELD RETURN nYear
        END IF
        IF nYear == 2012
            YIELD EXIT      // Exit the loop
        ENDIF
    NEXT
```

Index

- - -

- ! -

- # -

- $ -

- % -

- & -

- * -

- . -

- / -

X#

- T -

- U -

- V -

- W -

- X -

- Y -

- Z -

Back Cover