Visual FoxPro Programming Standards and Guidelines

Version 4.0

May 16, 2004

Visual FoxPro Development Guidelines

Table of Contents

Visual FoxPro Development Guidelines

Purpose of Standards and Guidelines

The definition of the word standard has many meanings, but in the context of developer standards we like the following definition:

stan·dard (stan'dard)

n. Abbr. **std**.

1. An acknowledged measure of comparison for quantitative or qualitative value; a criterion.
   An object that under specified conditions defines, represents, or records the magnitude of a unit.
2. The set proportion by weight of gold or silver to alloy metal prescribed for use in coinage.
3. The commodity or commodities used to back a monetary system.
4. A degree or level of requirement, excellence, or attainment.

We also grabbed this definition of Guideline while at Dictionary.com:

guide·line (gīd'līn')
n.

1. A statement or other indication of policy or procedure by which to determine a course of action.

The shining standard definition "A degree or level of requirement, excellence, or attainment" is something we pride ourselves in at White Light Computing, Inc. The standards and guidelines are written for two reasons, to write the best darn code that is ultimately supportable and extendible, and to make it easier to read and understand the functionality of the code developed.

Our code is written with the expectation that it performs optimally and that it meets or exceeds the requirements of our customers. We use these standards and guidelines to enhance the communication of the requirements and the implementation between our developers and the developers we collaborate with on projects. You might have questioned the context of "The set proportion by weight of gold or silver to alloy metal prescribed for use in coinage" with respect to developer standards. We consider these standards and guidelines the "gold standard" at our company. In this business (like so many others), time is money, and a deviation from standards costs us developer time, which translates to lost revenues. Nothing worse than a developer chasing his tail because he assumed the other developer adhered to the standards. Deviating from standards will guarantee that some day one of your co-workers will go down a path thinking one thing, when the reality is something else. Following the standards does not guarantee that your co-workers will not make a mistake or follow his or her assumptions, just to make sure we do not to push them into a large sinkhole.

Purpose of Document

The purpose of this document is to consolidate and clearly define the standards and guidelines that have been approved and are currently being practiced by the development staff at our company. These guidelines are a living list of items. All applications created during this timeframe are expected to follow these guidelines unless technically prohibited, until such time that a revision of these guidelines is released. It is not the intention of the development teams to grandfather these practices back to previously developed applications, nor projects assumed from clients that were developed by another company. Henceforth it will be necessary to indicate the version of standards followed in the documentation and code headers of the application being created.

It may initially seem odd that the standards and guidelines outlined in this publication seem very familiar. It is the intention of White Light Computing, Inc. to conform to as many industry standards as practical in meeting the needs of our customers and our developers. We have developed these standards from everyday practice in our past development and from reading the writings of some of the finest developers in our development community.

Who is Responsible to Follow the Standards and Guidelines?

You! Each and every developer at White Light Computing, Inc is responsible to follow the standards and guidelines detail in this documentation as it is published. This is not optional; it is a requirement for employment in this fine organization.

Who is Responsible for Change?

You! Each and every developer at White Light Computing, Inc (and other companies that adopt these standards) is responsible for enhancing, improving, adjusting, adding to, deleting from, and in general changing this living document. We not only encourage change, but demand that our developers help improve the development standards and guidelines. We know that the code solutions we develop and the communications between developers will be better each time this document is improved.

Process of Implementing Change

The process to improve the standards and guidelines is straightforward:

1. Develop the idea.
2. Put it in writing. Publish the idea, include any supporting documentation like the old standard if improving one, reason for the change or new idea, and make sure it is available for all other developers to see. This publication will be centrally located and accessible for all developers to be able to read. The repository will keep all the proposed changes until the regular standards and guideline meeting is held.
3. Open the debate dialog during a regular standards and guidelines approval meeting.
4. Motion is made and seconded to submit change for a vote, vote is taken, majority rules.
5. If accepted, the standards and guidelines librarian will update this documentation.

We want the process to be simple and flexible. It should be noted that the corporate management determine the participants in the regular standards and guidelines approval meeting. The findings of the participants at the meeting will be final for that cycle. Any new ideas or changes can be submitted again at a later meeting if they were not accepted at the meeting they were proposed.

Naming Conventions

Database Naming Conventions

Variable names, table fields, and other database objects should be in Hungarian notation for readability. The first character of the field name represents the type of variable it is. These identifying characters will be in lower case. Then the descriptive variable name using both upper and lower case letters will follow.

Data Type (VFP tables)

This standard will be used for each new project and is consistent for the entire application. The data type indicator is the first character position and is set up as follows:

Indicator	Data Type	Example
c	Character	curCustomer.**c**LastName
d	Date	curCustomer.**d**BirthDay
t	Datetime	curCustomer.**t**LastMod
b	Double	curCustomer.**b**Rate
f	Float	curCustomer.**f**Value
g	General	curCustomer.**g**Picture
l	Logical	curCustomer.**l**SentMail
m	Memo	curCustomer.**m**Comments
y	Currency	curCustomer.**y**YearToDate
n	Numeric	curCustomer.**n**Items
i	Integer	curCustomer.**i**CustID

Data Type (SQL Server tables)

This standard will be used for each new project and is consistent for the entire application. The data type indicator is the first character position and is set up as follows:

Indicator	Data Type	Example
chr	Char	@**c**FirstName
chv	Varchar	@**c**Activity
chrn	Nchar	@**c**LastName
chvn	Nvarchar	@**c**LastName
txt	Text	@**m**Note
txtn	Ntext	@**m**Comment
dtm	Datetime	@**t**TargetDate
dts	Smalldatetime	@**t**CompletionDate
iny	Tinyint	@**n**ActivityId
ins	Smallint	@**n**EquipmentTypeId
int	Integer	@**n**Asset
dec	Numeric/Decimal	@**n**Profit
rea	Real	@**n**Velocity
flt	Float	@**f**Length
mns	Smallmoney	@**y**Cost
mny	Money	@**y**Price
bin	Binary	@**l**Path

Indicator	Data Type	Example
biv	Varbinary	@**l**Contract
img	Image	@**g**Logo
bit	Bit	@**l**Operational
tsp	Timestamp	@**s**Current
guid	Uniqueidentifier	@OrderId **_PK/FK**
cur	Cursor	@**cur**Inventory

Views

All local and remote view names are to use "v_" as their initial characters.

The view name should include the name of the principle-underlying table used to populate the view cursor. The number of records expected in the result set should be indicated. Using the underlying table in the singular indicates that the view retrieves a single record. This will usually be used for a parameterized view using a primary surrogate or candidate natural key.

Multiple-record views can be indicated by using the principle underlying table in the plural (assuming that you follow the company standard of naming tables in the singular), or by including the word "List" in the view name, e.g. v_Customers for a Customer table, v_VendorList for a Vendor table.

Clause	Standard	Example
Filter conditions can be indicated by including "For" followed by the parameterized field	For	v_Customers**For**Region v_Orders**For**Customer v_Employees**For**Department
If necessary to indicate a sort order, include "By" in the view name	By	v_Customers**By**Sales v_Customers**By**Zip v_Customers**By**State v_Employees**By**Tenure
Views that are created in code will have "ND" (for No Designer) as a suffix to the view name	ND	v_CustomersSalesWithJoins**ND**
Views created initially for populating combo boxes or list boxes should have "v_cbo" or "v_lst" as their initial characters	cbo lst	v**l**_CustomerList v**l**_EmployeesForDept
Views created initially for reports should have "v_rpt" as their initial characters	rpt	v**r**_SalesYTDForCustomer v**r**_EmployeesByName

An optional standard is to include the letters "RO" as the last two characters in the view name (or just preceding the "ND") for views created as Read Only (non-updateable) views. In many cases, the fact that the view is not updateable can be inferred from the table name (seeing "List" in the name for example). Likewise, views that have a name identical to the underlying table (as in v_Customer for Customer) are very likely to be updateable.

If a view *must* be read-only, if the name is such that it could be inferred to be an updatable view, or if it's necessary to distinguish between two identically named views that differ only in whether or not their *SendUpdates* property is **.T.**, the programmer should include "RO" in the view name.

Note that one of the advantages of using a non-updateable view for some processes is that the view contents can be modified, unlike a cursor created from an SQL-Select command without the **READWRITE** clause, and that an updateable view can do double-duty. In cases where the view is opened without any intention of making modifications to the underlying tables, you can use the **NOUPDATE** keyword if opening the view in code, use the *ReadOnly* property of the cursor object if opening the view in the DataEnvironment, or issue a **CURSORSETPROP("SendUpdates",.F.,<Alias>)** in code to prevent unintentional modification of the underlying table. Be aware that frameworks like Visual MaxFrame and Visual FoxExpress will see any updateable view as being in need of a **TABLEUPDATE()**. If it will be necessary to **REQUERY()** the view at some point, do not use the *ReadOnly* property or **NOUPDATE** keyword, as the **REQUERY()** will trigger a "Can't update the cursor" error – use the **CURSORSETPROP("SendUpdates")** instead.

While you might find it convenient to use an updateable view in a context needing only a ReadOnly version of the view, you should never change a non-updateable view to an updateable view – this has the potential of introducing bugs in other parts of the system using the same view. Instead, open the view in the ViewEditor and save it under a different name, then change the *SendUpdates* property, or copy-and-paste the code in CREATEVW.PRG (see next section) to a new function, and add the necessary **DBSETPROP()** commands to change the updateable properties.

Views Created in Code

All views are to be maintained either using the ViewEditor, or for more complex views, in code to be contained in a PRG file included in each project, CREATEVW.PRG. Each **CREATE SQL VIEW** command is to be wrapped in a function, with the naming convention "vw" + view name. This allows easy location of views in a CREATEVW.PRG that contains hundreds of views by using the Document View tool included in Visual FoxPro. If undecided whether to create a view using the ViewEditor or in code, preference should be given to maintenance in code, even for updateable views. Also note that the ViewEditor will save your view definition to a PRG.

Views maintained in code instead of the ViewEditor must use the "ND" suffix to indicate "No Designer", to indicate that the view is maintained in CREATEVW.PRG, e.g:

```
FUNCTION vwv_customerListND
	CREATE SQL VIEW v_CustomerListND as ;
		SELECT customer.cCustomerNo, ;
				customer.cName, ;
				customer.cCity, ;
				region.cDescription, ;
				upper(customer.cName) as cOrderBy ;
			FROM customer join codes ;
					ON customer.Region_FK = region.Region_PK ;
			WHERE customer.lActive ;
			ORDER BY cOrderBy
ENDFUNC
```

Database Objects

Names of database objects should also consist of two parts:

- The prefix, which describes the type of database object.
- The base part, which describes the content of the object.

The following table shows object abbreviations that should be used as prefixes:

Object:	Abbreviation:	Example:
Table		Activity (singular)
Stored procedure	usp	**usp**CompleteOrder
Trigger	tr	**tr**Order_IUD
Default	dft	**dft**Today
Rule	rul	**rul**CheckZIP
Index	idx	idx_LastName
Primary key	_pk	ContactId_**pk**
Foreign key	_fk	Order_OrderType_**fk**
User-defined datatype	udt	**udt**Phone

Tables should not have prefixes to describe the object type.

Triggers

The base part of the trigger name should be the name of the table to which the trigger is attached. A suffix which shows modification statements (insert, update and delete) should also be part of its name.

Example:

- trOrder_IU (trigger on the Order table for Insert and Update)

If there is more than one trigger per modification statement attached to the table (not possible prior to version 7.0), the base part should contain the name of the table and a reference to a business rule implemented by a trigger:

Examples:

- trOrderCascadingDelete_D (delete trigger on the Order table which implements cascading deletes of order items)
- trOrderItemTotal_D (delete trigger on the Order table which maintains a total of order item prices)

Stored Procedures

The base name of a stored procedure should usually be created from a verb followed by a noun to describe the process the stored procedure performs on an object.

Examples:

- uspGetEquipment
- uspCloseLease

If the procedure performs several tasks, all of those tasks should become part of the procedure name. It's okay to make procedure names longer then variable names. You should be able to pick a name between 20 to 40 characters.

Some developers use the 'sp_' prefix in front of the base name of a stored procedure. This prefix should be reserved for system stored procedures which reside in the master database and which are accessible from all databases.

You should also avoid computer-oriented or fuzzy names like:

- uspProcessData
- uspDoAction

Variable Naming Conventions

Variable names must be in Hungarian notation for readability with the first character defining scope and the second character representing the type of variable it is. These identifying characters will be in lower case. Then the descriptive variable name using both upper and lower case letters would follow. Array data types can have an additional (optional) third character. This third character defines the data type that is contained within the different columns in the array. If there are different data types in different columns, use the “x” data type for unknown.

Scope

The scope is not determined by the name of the variable, but rather how it is declared and scoped by the development language. This guideline assists the developer when reading code, but it in no way guarantees that the variable is scoped accordingly (although they should be confident in their co-workers following the standard). This is the responsibility of a good developer. The following are the guidelines of the scope character.

Indicator	Scope	Example
l	Local	**l**nCounter
p	Private (default)	**p**nStatus
g	Public (global)	**g**nOldSafety
t	Parameter	**t**nRecNo
c	Constant	**c**nARRAY_ROWS
r	Report	**r**nHours
vp_	View Parameter	**vp_**ComPk

Constants are defined with **`#DEFINE`** and are used only for compiles. They are scoped for the program or method that they reside in. The use of the underscore breaks up word syllables and is optional. The industry standard is to uppercase all but the scope and data type characters. Some third-party products may use include files (used via the **`#INCLUDE`**) that have constants defined that do not use the scope and data type characters. These constants will be used as they are defined by the third-party product.

All view parameters are to use "vp_" as their initial characters. The remainder of the parameter variable name should mimic the table field involved in the comparison, e.g. "**`WHERE customer.cCustID = ?vp_cCustID`**" or "**`WHERE company.region_fk = ?vp_Region_Fk`**". Where this is not practical for some reason, the view parameter should give some indication as to what table elements are involved in the comparison.

Data Type

Like the scope character, the data type character allows the developer to read the code easier. It does not guarantee that the variable is the indicated variable type. This is the responsibility of a good developer to name the variables accordingly. The following are the guidelines of the data type character.

Indicator	Data Type	Example
a	Array	l**a**Months
c	Character, Varchar	l**c**LastName
q	Blob, Varbinary	l**q**Document
y	Currency	l**y**CurrentValue
d	Date	l**d**Birthday
t	Datetime	l**t**LastModified
b	Double	l**b**Value
f	Float	l**f**Interest
g	General (not used)	l**g**Logo
l	Logical	l**l**Flag
n	Numeric	l**n**Counter
o	Object	l**o**Employee
u	Unknown or mixed	l**u**ReturnValue

Field Memvars vs. Tables vs. Declared Memvars

The use of m. with memvars is necessary to differentiate between variables declared with a **`SCATTER MEMVAR MEMO`**, and the field name in the table. With the naming convention of defining the scope and datatype of variables you will

not confuse variables defined local and those defined via a **SCATTER MEMVAR MEMO**. Table/Cursor alias names should always be used when referencing cursor field names just to further differentiate that you are not referencing a memvar. Our recommendation is to use SCATTER NAME and use the standard object reference to the data to avoid any problems/confusion with column names and memory variables.

Object Naming Conventions

Objects are utilized throughout application development. This guideline assists developers in reading method code. It does not guarantee that the objects instantiated are going to be as they are named. This is the responsibility of a good developer. The following are guidelines for naming different base objects provided by the development language.

Indicator	Object Type	Example
acd	ActiveDoc	**acd**Customer
chk	CheckBox	chkReadOnly
cbo	ComboBox	**cbo**English
cmd	CommandButton	**cmd**Cancel
cmg	CommandGroup	**cmg**Choices
ctr	Container	**ctr**MoverList
ctl	Control	**ctl**FileList
cad	CursorAdapter	**cad**Employee
cus	Custom	**cus**App
dte	DataEnvironment	**dte**EmployeeEntry
edt	EditBox	**edt**TextArea
frm	Form	**frm**FileOpen
frs	FormSet (not used)	**frs**DataEntry
grd	Grid	**grd**Prices
grc	Column	**grc**CurrentPrice
grh	Header	**grh**TotalInventory
hpl	HyperLink	**hpl**Company
img	Image	**img**Icon
lbl	Label	**lbl**HelpMessage
lin	Line	**lin**Vertical
lst	ListBox	**lst**PolicyCodes
olb	OLEBoundControl	**olb**GraphicLogo
ole	OLEControl	**ole**ProgressBar
opt	OptionButton	**opt**French
opg	OptionGroup	**opg**Type
pag	Page	**pag**DataUpdate
pgf	PageFrame	**pgf**Left
phk	ProjectHook	**phk**Base
rel	Relation	**rel**InvoiceToItems
sep	Separator	**sep**ToolSection
shp	Shape	**shp**Circle
spn	Spinner	**spn**Values
txt	TextBox	**txt**GetText
tmr	Timer	**tmr**Alarm
tbr	ToolBar	**tbr**EditReport
xad	XMLAdapter	**xad**WSResult

Indicator	Object Type	Example
xfd	XMLField	**xfd**StockPrice
xtb	XMLTable	**xtb**Checks

Source Code Naming Conventions

Indicator	File Type	Example
pck	Picklist/Lookup form	**pck**People
rpt	Report criteria form	**rpt**CustomerList

Header Information

Program/Class Header

Program and Class Header Information for all non-generated programs and class libraries is to be created using an outlined box. Also, the header must include all pieces listed below unless specifically stated as optional, even if there may be no information to provide for that topic. There is no upper limit to the amount of topics a developer may wish to include in the program header. Additional information is always received well by other developers. Single spacing is required between topics and topic headings are in upper case.

Note: Only programs developed within the company need to be documented in this fashion. The respective development staff documents any third-party programs or libraries.

Section	Description
Program Name or Class Name	Program name Class Name
Author	Original programmer's name.
Copyright	Year program was created or modified.
System	This is the name of the application the program or class was created for or used by. If it was created for a common library or other generic routine place “Common Program” or “Common Library” as appropriate. The directory where the application is located can be substituted for the system name.
Program Description	Description of the program purpose and any other documentation that is appropriate. It is always better to fault on the side of too much description <g>.
Parameters:	Parameter section consists of four subsections. All subsections are optional unless parameters exist in the code. If there are no parameters, then “None” is the description for the section and none of the subsections are included.
Calling Syntax	Example of a program line with all input parameters, and actual syntax. All parameters must be in brackets <> and will correspond to the parameters in the Input Parameters section.
Input Parameters	Description, data type, whether optional, whether passed by value, indicate default value if set.
Output Parameters	Description, data type, whether it is a returned value, and indicate any input parameters which are altered intentionally for use in the calling routine.
Sample Call	Real calling example. Use as much code as necessary to give other developers a good idea of how this routine can be executed (i.e. lines of setup and follow up code needed)

Section	Description
Databases Tables Accessed	List of all tables accessed in this routine that are contained in a database. Also include the database container that the table belongs to in parentheses. *This is optional.*
Free Tables Accessed	List of all tables accessed in this routine that are not contained in a database. *This is optional.*
Development Standards	Version of standards used, and any non-conformance.
Special Requirements/Devices	Special requirements or devices needed for the program to execute correctly. *This is optional.*
Test Information	Note any test information within the program. Must include name of Test Driver if the program is a library module. *This is optional.*
Future Enhancements	Any thoughts on suggested enhancements for a future release. *This is optional.*
Language/Version	Development tool and version used to create routine, or version it can be run under.
Change Log	Date, developer initials, work order number assigned, and description of change. This must be included any time the routine is modified. Include any hint to the work that was completed for the changes implemented.

Program/Class Example:

```
*******************************************************************************
*  PROGRAM NAME: FrameWorkWizard.prg
*
*  AUTHOR: Richard A. Schummer
*
*  COPYRIGHT © 2001-2004   All Rights Reserved.
*     White Light Computing, Inc.
*     42759 Flis Dr.
*     Sterling Heights,  MI  48314
*
*  SYSTEM: Internal FrameWork Wizard
*
*  PROGRAM DESCRIPTION:
*     This program starts the application wizard, sets up the initial environment,
*     instantiates the application object, and kicks off the READ EVENT.
*
*  PARAMETERS:
*     CALLING SYNTAX:
*        DO FrameWorkWizard.prg WITH <tcParam1>
*
*     INPUT PARAMETERS:
*        tcParam1 = Optional parameter, this is a way to access special
*                   development features when it is "Development",
*                   otherwise nothing special happens
*
*     OUTPUT PARAMETERS:
*        None
*
*     SAMPLE CALL:
*        DO FrameWorkWizard.prg WITH "Development"
*
*  DATABASE TABLES ACCESSED:
*     None
*
*  FREE TABLES ACCESSED:
*     AppInfo = Application initialization file
*
*  DEVELOPMENT STANDARDS:
*     Version 3.0 compliant
*
*  TEST INFORMATION:
*     None
*
*  SPECIAL REQUIREMENTS/DEVICES:
*     None
*
*  LANGUAGE/VERSION:
*     Visual FoxPro 7.0 or higher
*******************************************************************************
*                          C H A N G E    L O G
*
*   Date      Dev  System  Description
* ----------  ---  ------  -------------------------------------------------
* 01/20/1999  RAS  PT      Created program
* -----------------------------------------------------------------------------
*******************************************************************************
```

Method/Procedure/Function Header

Procedure and method header information for all non-generated procedures and class libraries is to be created using an outlined box. Creating the right side of the box is optional, but should remain consistent through out the application. Also, the header must include all pieces listed below unless specifically stated as optional, even if there may be no information to provide for that topic. There is no upper limit to the amount of topics a developer may wish to include in the header. Additional information is always received well by other developers. Single spacing is required between topics and topic headings are in upper case.

Note: Only custom methods (those not provided by Visual FoxPro) need this documentation.

Note: Only programs developed within the development team need to be documented in this fashion. Any third-party programs or libraries are documented by their respective development staff.

Section	Description
Procedure Name or Function Name or Class Name	Procedure, function, or class name. *This is optional.*
Author	Original programmer's name. *This is optional*
Procedure Description or Function Description or Class Description	Description of the routine's purpose and any needed documentation that is appropriate. If this is a class method, the class method description that is entered in the New Method or Edit Property/Method dialogs (and shows in the Property Sheet) is accepted in place of this in the code. If there is not enough room in the 256 character limit that the description has, then add more in this header.
Parameters: Input Parameters Output Parameters	Parameter section consists of two subsections. All subsections are optional unless parameters exist in the code. If there are no parameters, then "None" is the description for the section and none of the subsections are included. VFP intrinsic parameters do not need to be documented. Name, description of parameter, data type, if optional, if passed by value, indicate default value if set. Name, description of parameter, data type, if it is a returned value, and indicate any input parameters if they are altered intentionally for use in the calling routine.

Procedure/Method Example:

```
************************************************************************
*  PROCEDURE NAME: SelfTest
*
*  PROCEDURE DESCRIPTION:
*     This method contains code to completely test this object standalone.
*     The tests are performed by passing it a number indicating the test
*     to be performed.
*
*  PARAMETERS:
*     INPUT PARAMETERS:
*        tnTest    = Required, numeric, a number between 1 and nth test.
*
*     OUTPUT PARAMETERS:
*        llSuccess = Logical, .T. if all tables open, else .F.
************************************************************************
```

Note: The header for a class that resides in a Visual Class Library should have the header place in the custom zReadMe Method (standard in Visual MaxFrame Professional) or the zzAbout added to the ILibs in Visual FoxExpress.

Comments

Full Line and In-Line

Comments within the code are done in two different manners, full line and in-line. Full line comments are used when a detailed explanation is required. Full line comments are to be written as complete sentences and void of abbreviations unless defined prior to use within the specific program being commented. Full line comments are indented to line up with the segment of code being commented.

In-line comments are used to provide a short statement about a specific line of code. In-line comments are not required to be full sentences, and can be abbreviated if the abbreviation is easily understood. In-line comments are included on the line with the program code and are required to line up with each other through out a segment of code.

An accepted guideline is to begin in-line comments in column 40, but this is optional. The key is to line all comments up and remains consistent throughout the application.

All comments should reflect what the actual code is doing. A liberal use of comments is better than a sparse use of commenting. If you change the code, change the comments to reflect the changes made.

Date the comments when changing code that is already in production. Initial the comments with the developer initials so the others know who made the change. These are both optional.

Special Commenting

Starting a line with an asterisk (*) usually indicates normal commenting through out an application. In addition to normal comments we have categorized various special comments. Using special commenting practices are not required, but could be very helpful for developers doing joint development of an application. All special commenting should include initials of developer and date.

Comment Characters	Description
*!	Enhancements and future features
*?	Open Items for Release; marker to indicate a correction or change required prior to release
*+	Walkthrough questions and issues
*^ OR *!*	Blocking out blocks of test code
*<	Changes commented out instead of removed, must include comment.
*$	Workaround a known Visual FoxPro or framework bug

Example:

```
*? Make sure release version updated
*+ Ckeck out new process for standard library

*< RAS 07-Apr-2001  Different process implemented
*< DO SkipItPR
DO SkipIt2PR
```

Custom Properties and Methods

All custom properties and methods must have comments that show up in the property sheet.

Indentation and Spacing

Indenting nested conditions and loops is expected. Three spaces or a tab is required for indentation. Tabs settings are set to 3 spaces when used to give the code a consistent look. Do not nest over six levels deep. **IF…ENDIF** statements and looping constructs need to be separated with one blank line before and after. The only exception to this is if there is an IF statement or looping construct directly before a line of code and it is indented. The indention will be equivalent to the blank line and no additional blank line is required.

EXAMPLE:

```
IF llTester
   DO TestProcedureForDevelopment
ELSE
   IF llActual
      DO OtherPR

      DO WHILE !EOF()
         DO SkipItPR
      ENDDO

      FOR lnCounter = 1 TO 10
         IF llNoWay
            DO Backup
         ENDIF
      ENDFOR
   ELSE
      FOR lnCounter = 1 TO 10
         DO SkipIt
      ENDFOR

      MESSAGEBOX(“Nice document!”,16)

      IF llSetThis
         gnValue = 1
      ENDIF
   ENDIF
ENDIF
```

Line spacing is required between procedures, functions, class definitions, methods, and/or subroutines incorporated in the same program. The accepted guideline is 2 blank lines. The goal is to achieve readable code. It is also important to remain consistent throughout your code.

Variables, conditions, and expressions found in a block, procedure, or function are required to be in alignment with others of its kind. It is a suggested practice that variables and similar expressions being defined be aligned on the "=" sign within a block of code (optional).

EXAMPLE:

```
lnCount       = 1
lcMaxNum      = "1000"
ldYearEnd     = {^1997/01/01}
```

Aliasing

All field names are to be aliased all the time. The only exception is in a **REPLACE** command, in which the aliasing of the field names is optional (but encouraged), but the **IN** clause is required.

All functions that accept an alias argument, (e.g. **RECNO(), EOF(), RECCOUNT(), CURSORGETPROP()**) should never be used without the alias argument. A rule of thumb for the developer/programmer is that any time they issue a **SELECT <alias>** command, a warning bell should go off. If it is being done to support a command that does not

accept an alias argument (such as **SCAN…ENDSCAN** or **LOCATE**), then ok, but if it is being done to support a function or command that accepts an alias argument, the **SELECT <alias>** should be removed, and the alias argument should be added to the command or function.

Best Practices Regarding Updateable Views

An updateable view must act only on a single table, and *in general*, should only include columns from that single table.

It may be useful on rare occasions to include fields from another related table in an updateable view. Sometimes doing so can be a very convenient way of simplifying some implementation detail. There is nothing inherently wrong with including non-updateable fields from a related table in an updateable view; it is simply a practice that should be used sparingly. Whenever you find yourself wanting to follow this practice, ask yourself how you would implement the solution *without* including related, non-updateable fields in the updateable view. Examine the design of the data structures you're working with; a need to include fields from related tables in an updateable can indicate a flaw in the basic data model.

If implementation without inclusion of the non-updateable related fields is not significantly more complex, or significantly less intuitive or maintainable, then keep the updateable views "pure". If there is a flaw in the database that is being circumvented by inclusion of non-updateable related fields, fix the database schema if possible. Otherwise, if the database design is good, and if inclusion of non-updateable related fields in the view results in a greatly simplified or maintainable implementation, have at it.

Note that a variation on the inclusion of non-updateable related fields into a view is to include joins on tables whose fields are not included in the field list, and (and as a result are not updateable), but do participate in WHERE clauses. The principle for use of these views is the same as that for updateable view that include non-updateable fields from related tables.

A view should never, ever update fields from more than one table. While such a view is supported in Visual FoxPro, it is not supported in any database server such as SQL/Server or Oracle, thus this practice (in addition to being very poor design) makes your application VFP/LAN specific, and cannot be upsized to a client/server application at a later date. Even in the case of a one-to-0-or-one relationship, it's just as easy to create two updateable views

Development Language Reserved Words

All words defined and/or reserved by the development language be in the same case and spelled out. Our company has chosen to put all reserved words in upper case. Again, the developer is required to remain consistent throughout the application. All words and/or variables created by the developer are to be in Hungarian notation (mixed case, upper on word syllables). Properties follow variable naming guidelines; methods follow Procedure naming guidelines. The advent of IntelliSense in Visual FoxPro 7 will enhance the conformance of this standard provided that you configure the IntelliSense Manager with the correct settings.

Methods, Procedures, and Functions

Method, Procedure, Function Naming

Method, procedure, and function names are not required to be a specific length, but Hungarian notation (mixed case, upper on word syllables) is required and must be descriptive to the development staff.

Method names will follow the Verb-Object except where several methods should be grouped on the property sheet. This could be where several methods are all hook methods that should start with the method from which they are

called, or some other situation in which the methods should logically be grouped by some attribute such as the object on which they operate, or the process they support.

Form delegation

Developers spend too much time trying to track down bugs in a call to a program somewhere in framework classes. These problems usually result from a call in a control method in a composite class. Had these calls been in a form method, it would have been found much quicker. To reduce the amount of debugging, it is a programming standard to delegate most form functionality to form methods. This eases maintenance, encourages code re-use and good object design.

Exceptions to this rule can include a single control **`Refresh(),`** code to change state, enable/disable, or combobox Requery() code.

Checking Number of Parameters

The suggested practice for all methods, functions, and procedures with parameter(s) is that they must check for the number of parameters and assign default settings for optional parameters if not passed. If parameters are required and not passed, then the procedure should display an error condition and halt execution. Assertion code (see **`ASSERT`** and **`SET ASSERT`**) should be used when developer messages are required. These will never reach the end user.

Passing Parameters

All parameters are passed by value to functions and by reference to procedures (Visual FoxPro default) unless otherwise specified.

File Naming Conventions

Application files have been separated into four categories: Data Files, Executable Files, Application Source Files, and Class Library Files. Each of these files has a disk presence. All files can use the long file name supported by all current PC Operating Systems and Network Operation Systems. The only exception will be files that are implemented on a OS or NOS that does not support long file names in the implementation. It is recommended that the names be short and sweet, yet descriptive for others to understand. There will be no embedded spaces in any filenames or directories.

Data Files

Data Files include databases, tables and flat ASCII files.

Databases

Database names are descriptive to the context of the application(s) that use it. Some frameworks like Visual MaxFrame Professional framework forces database names to consist of two characters.

Tables

The database table name should reflect the type of entity of information retained in the rows of the table. Long filenames and aliases are acceptable for implementations on platforms that allow long filenames. All table names will be singular in nature. For instance, we will use Customer, not Customers.

Project Files

Project files are considered source code. Projects will be stored in the root folder for the project. Optionally, we prefix project names with a three-letter acronym to indicate the company the project is created for.

Executable Files

All executable files are compiled with the same name as the application project file. This helps developers determine the project to maintain when enhancements need to be made or bug fixes need to be applied. Optionally, executable files can have the company three-letter prefix removed.

Application Source Files

All application source files (including programs, forms, menus, and queries) should be named to reflect the process or feature they represent. The key is to place them in the appropriate directory under the project directory. The company uses the following directory structure:

J:\WLCProject\NiceCustomer\Common\

Libs	(Visual Class Libraries)
ILibs	(VFE Framework Class Libraries)
Docs\	(Documentation)
Images	(Graphic files, including BMP, ICO, GIF, JPG)
Sounds	(Sound clips)

J:\WLCProject\NiceCustomer\CoolProject\

Libs	(Visual Class Libraries)
Data	(Databases and tables)
Deploy	(Installation project files, like InstallShield, Wise for Windows Installer, INNO)
DataModel	(xCase data models)
Docs\	(Documentation)
FunctionalSpec	(Functional Specification Documentation)
Support	(Technical Support information)
Forms	(Forms)
Images	(Graphic files, including BMP, ICO, GIF, JPG)
Html	(HyperLink Markup Language files)
Includes	(Header include files)
Labels	(Label forms)
Menus	(Menus)
MetaData	(Metadata for the application)
PDFs	(Acrobat Reader documents)
Prgs	(Program files)
Queries	(Queries)
Reports	(Report forms)
Sounds	(Sound clips)
Text	(Text files)
Test	(Any test files used in Quality Assurance)
Help	(Help source code)
Conversion	(Data conversion source code - Optional)
Save	(AKA: CYA, files that need to be saved for later, possible subdirectories)

Development Directory Structure

Grouping	Directory
Visual FoxExpress Libraries	J:\WLCProject\Framework\VFE\VFEFrame\Libs
Visual FoxPro Fox Foundation Classes	J:\WLCProject\Framework\FFC

Protocol to Change Framework Class Libraries

No one changes the framework class libraries without expressed written permission from the chief architect or project manager. Only company class librarians change the code in the libraries above the project level. No classes will be subclassed from a class that is lower in the hierarchy. For instance, a report object form in the company level will not be subclassed from the project level report object form.

Temporary Cursors

Temporary cursors are generated via SQL-Selects or create cursor code. These cursors need to be named uniquely and the name should reflect the data in the cursor. Naming convention is to start out the cursor name with a "cur" so other developers know that the cursor was generated and not the alias of a database table or view.

Always clean up temporary cursors when the process they are used by is completed.

Message Handling

The company standard is to utilize the public domain MsgSvc functionality to allow all our applications to be enabled for internationalization. All commercial frameworks provide a mechanism to access the strings translation table provided by Steven Black.

All other standard information windows will leverage the Windows (and Visual FoxPro) Message Box. All messages will use the appropriate icon and button set. A default button is expected. All **`MESSAGEBOX()`** function calls will use the standard constants defined in the FoxPro.h include file.

EXAMPLE:

```
=MESSAGEBOX("Need to pass name of Database Container and it's path", ;
            MB_OK + MB_ICONEXCLAMATION, ;
            _SCREEN.Caption)
```

The company standard is to include the **`_SCREEN.Caption`** as the caption of any message.

Access Keys

Access keys (or Hotkeys as they are sometimes called) are useful for various reasons. Some of the reasons for assigning access keys include the following. If the mouse becomes inoperative the user may still use the application. Access keys allow users who are fast typists and prefer the keyboard to access the controls via the keyboard. Users with certain disabilities may not be able to use a mouse. Access keys make the application more user-friendly.

The rule of thumb for assigning an access key is to use the first letter, or meaningful letter in the caption. A letter should only be assigned as an access key once. If the first letter is already assigned to another control, then look for an unassigned consonant. Lastly, use an unassigned vowel. If there still are no unassigned letters, consider changing the control's caption, reassigning letters on other controls or if assigning an access key falls with the discretionary use guidelines, not using an access key.

Some controls in a project should always have an access key, while others are left to the discretion of the client or project leader. Destructive controls may or may not use an access key; however, destructive controls that use an access key should always prompt the user before proceeding. Controls that are inaccessible without a mouse always must use an access key. All controls should be keyboard accessible in some manner, whether it's via a tab or hotkey. Controls without a caption can use an access key by placing the access key on the label preceding it. Just set the tab number of the label with the access key to one less than the control you wish to access.

Controls that Require an Access Key

- Menu pads
- Menu items
- Pages on a page frame
- Command buttons (with Captions)
- Controls that are only accessible with a mouse

Discretionary Access Key Use

- Any control as required by design or customer

Help

Help is to be invoked by pressing the F1 Key or off the menu following the Windows User Interface Standards and is presented in the Windows Help System (HTML Help). The practice of using Context Sensitive Help is highly recommended.

Reports

The following section describes guidelines for the development of reports using the VFP report designer. Each project will have a template for the reports to follow. The project leader to meet the expectations of the customer defines the standard template. Some basic considerations to be included in report guidelines include:

- Right-justify numeric fields, align on the decimals.
- Left justify character fields
- Align titles the same as the data. Left justify for character, right for numerics.
- Include the date and time.
- Always show the data filtering criteria on the report so the user knows the scope of the data included. This filtering criteria is in natural language, not the VFP filter statement.
- Always design for 8.5 x 11 paper, unless otherwise dictate by special requirements.
- Be consistent in the design and layouts of reports for a client. For example, always use the same heading styles, logo placement, date placement, fonts, font size, etc. for all reports for the customer/project.

Miscellaneous Guidelines

This section is a catchall category to define guidelines for various developments within Visual FoxPro.

- Arrays will use the square brackets ([and]) to delimit the array indexes as well as the array dimensioning.
- **DO CASE** will contain an **OTHERWISE** even if blank (with documentation defining why blank) so other developers know that the original developer considered the alternatives.
- Leave the **SET** environment the way it was when you make a change to a **SET** before exiting the method, procedure or function. This is especially critical when developing in a black box object.
- Declare all variables **LOCAL** unless otherwise needed **PRIVATE**. Document the reason they were declared in the first place, on the line they are declared or on the line above the declaration.
- All variables are defined at the top of a method, procedure, function, or program.
- Always use "**FOR !DELETED()**" on appends (unless deleted records are required by customer)
- No dots (.) around the **AND, OR,** or the **NOT** logical operators
- The not equals operator will be indicated by the **<>** OR **#**.
- SQL-Selects should only be performed with previously opened cursors (avoids potential pathing problem).
- All table joins in SQL-Selects will use the **JOIN** syntax instead of the **WHERE** clauses.
- Captions will be filled in for tables and views, fields, and indexes (via DBCX)

- Use the **SELECT()** to save the current workarea before selecting another workarea. The workarea can always be reselected; cursor aliases go away when they are closed.
- All temporary cursors created in code are closed manually.
- **PRIVATE** variables must initialized after declaration.
- Others...

Versioning

- Version numbering system is as follows:
- The initial release of a product will be v1.0.
- Visual FoxPro 5.0 introduced the internal EXE version number. This number is in the format 1.2.3. The first number is the major release number, the second number is the minor release number, the last number is the build number. The build number is a counter incremented each time the EXE is built for the current release. This number can be used to keep track of issues with the release as interim betas are created.
- Major enhancements will increase by one.
- Minor enhancements will increment by tenths.
- Releases for correction of anomalies or betas will use the "build" number.

Version Release Example

A release of V2.3.165 means there have been two major functional releases of the application. It is on the third minor release, and this is the 165th time the application EXE was created for this minor release. Note that when the version goes to 3, all subsequent revision levels will go back to zero.

Testing

This section contains the issues related to testing applications.

No piece of code shall be considered ready for testing until after it has been run by the developer and has no syntax errors. If you cannot run it, you cannot test it! The Quality Assurance Team will test all application functionality before it is released to a customer for acceptance testing and production use.

No EXE is ready for customer use until after it has been run standalone by our staff.

All business objects will have a SelfTest() method. This method will contain one or more test scenarios to completely test the business object functionality. This test method will be developed when the object is written and will be considered the foundation of the test plan. The developer will write this method as a communication to other developers in our company so they understand how this object is used and what is needed to utilize this object within the context of the application.

Project Level Standards

All the standards and guidelines included in this document are generic enough to be used across all applications developed by our company. They also leave room for flexibility when it comes to a specific project. Items that should be considered at the project level include:

- Standard Labels
- Standard Industry or Application-wide Abbreviations
- Standard Fonts (Ex. Tahoma, size 8)
- Hot Key Standards
- Standard form layouts
- Report Layout standards

- Standard behavior (Ex. When I enter a zip it should fill in the City in every form)
- What is the minimum screen resolution this system should handle?
- What hardware does this system have to be compatible with?

Appendix A: Document History

Date	Updater	Description
May 16, 2004	Richard A. Schummer	Adopted previous documented (prior life at EDS, Kirtland Associates, Polaris Solutions, and Geeks and Gurus) as basis to start the new standards implemented at, White Light Computing, Inc.