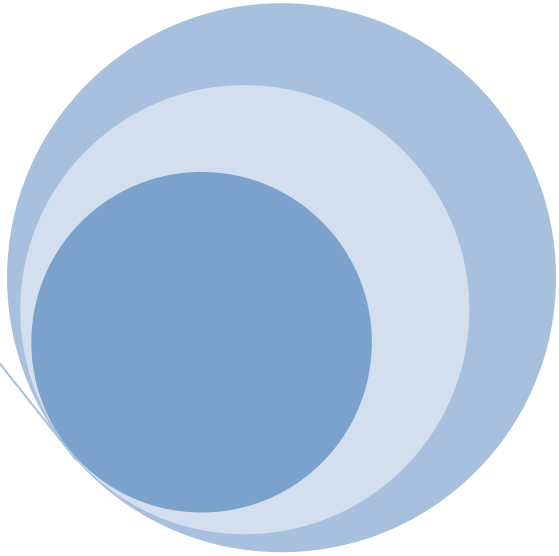
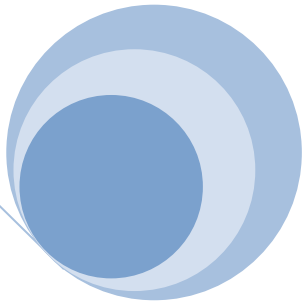




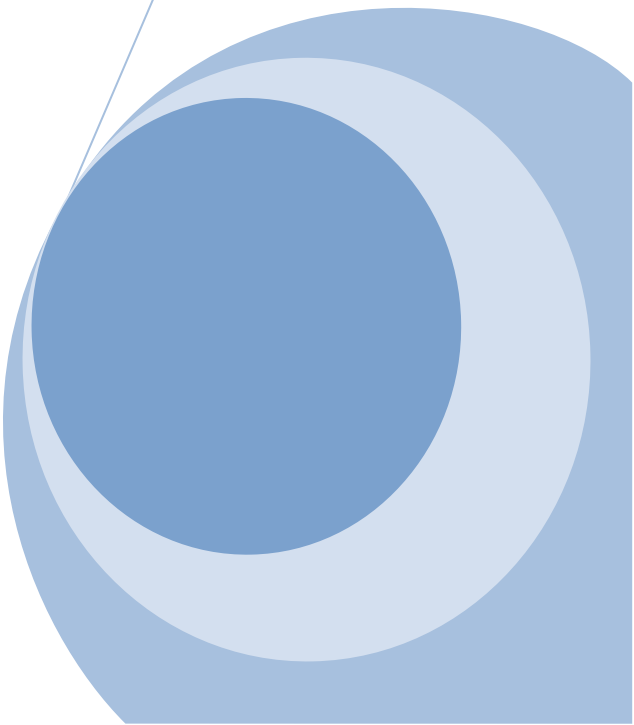
OpenInsight Coding Standards

Best-practice programming notes for Application Development

A short guide containing essential programming practices for developing OpenInsight applications.

VERSION 1.0.7

Carl Pates - Sprezzatura Ltd
12 November 2013



Contents

Introduction

Utilising Sprezzatura's considerable OpenInsight application development experience and BASIC+ program writing knowledge, this document has been written to outline those practices that Sprezzatura consider to lend themselves to maximum maintainability and performance for OpenInsight applications.

Over the next few pages we will discuss adopting a general coding style and how to best structure your programs. This will enable your programs to be better understood by other application developers and make updating those programs much easier for you in the future.

We will also take a look at using comments within your code and how to handle errors correctly. Again, the good use of comments will enable your code to be better maintained in the future and following a standard practice for error handling will result in a better experience for your users and also your support colleagues.

Towards the end of the document we will take a look at adopting best practice standards whilst working within the OpenInsight Form Designer, including the use of commuter modules.

Finally, this document will look at techniques that will help to optimise your code and thereby maximise the performance across your OpenInsight based applications. In closing this document will promulgate the use of binary manipulation functions.

This document is not intended to be cast in stone but rather as Sprezzatura finesse the techniques outlined herein whilst all the time acquiring new ones it will be updated to reflect this. Perfection is the goal not the actuality.

General Coding Style

Use of case in function and variable names

Basic+ is a case-insensitive language so theoretically you have plenty of freedom to style your code as you see fit – a style is not imposed on you by the language designers. Generally speaking you will see four types of case-usage when programming Basic+:

- Pascal-Case: Get_Property, SendMessage, Xlate etc.
- Camel-Case: get_Property, sendMessage, xlate etc.
- Lower-Case: get_property, sendmessage, xlate etc.
- Upper-Case: GET_PROPERTY, SENDMESSAGE, XLATE etc.

By default the new System Editor++ enforces Pascal-Case so most new code will probably use that unless you change the keyword configuration record. Either of the first two options are preferred – use of mixed-case characters makes the code easier to read - the worst option to choose is the all upper-case one – avoid this if at all possible – it will make your eyes bleed.

A suggested approach to using case within your programs follows here (these conventions are based on the recommendations in Code Complete and the .Net coding guidelines on the MSDN website):

Code Entity	Preferred Case	Example
Function Name	Pascal	SendMessage Get_Property Xlate
Function Parameter	Camel	objectID param1
Internal Subroutine	Pascal	ParseTheSentence OnCreate
Local Variable	Camel	patientRow pestName cntr
Global Variable	Camel + "@"	userID@ envRec@
Equated Constant	Upper + "$"	TRUE$ MAX_SIZE$ BOOKS$TITLE$
System Variable	'@' + Camel	@sysTables @file_Error @lower_Case
Compiler directives	"#" + Upper "$" + Upper	#DEFINE #PRAGMA $INSERT $USES

Ultimately it's down to personal preference and to any standards enforced by your organization but whichever you choose ensure that the use is consistent across your application codebase.

Indenting - tabs or spaces?

The question of using tabs or spaces for indenting is one of the original holy wars waged amongst the members of the programming community.

We prefer using 3 spaces for indenting for the following reasons:

- It makes code easy to share between Arev and OpenInsight (The Arev editor doesn't deal well with Tab characters)
- The code can be copied or viewed in other tools (such as email) and it always looks the same – there is no need to worry about tab settings.
- We don't have a problem using the space bar (It's not like it's really heavy to lift or anything is it?)
- One day the System Editor++ will be clever enough to have a "Convert Tabs to Spaces" option and then there will be no excuse for not using spaces.

Both methods have their advantages but regardless of which you decide on make sure usage is consistent across your code.

Using spaces in statements

Use spaces in your statements to increase the clarity of your code. Separate identifiers from operators and other identifiers with a space.

Don't do this (becauseitmakesitreallyhardtoread):

```
someStr=Get_Property(ctrlEntID,"TEXT"):"-":xStr
```

Do this instead:

```
someStr = Get_Property( ctrlEntID, "TEXT" ) : "-" : xStr
```

Avoid period characters in variable names

Basic+ supports using "." characters in your variable names. This should be avoided when creating variables as underscores are the preferred method of breaking up words within variable and function names (using "." characters in variable names tends to confuse programmers familiar with other languages that use "." to denote object/property notation like VB and JavaScript).

Many system variables use the "." character in their names, but the compiler also supports a "_" variant which should be used in preference.

For example, don't do this:

```
If @file.Error< FSCODE$ > = FS_REC_DNE$ Then
```

Do this instead:

```
If @file_Error< FSCODE$ > = FS_REC_DNE$ Then
```

Use namespaces when creating application components

When creating new components in your application care should be taken when naming them. Avoid simple names such as Editor, Utility, PlaceDialog and so on because you run the risk of colliding with items already defined in the core system, or items that may be installed in a future Revelation upgrade. Instead prefix your component names with a specific string to differentiate them from those supplied by Revelation or another company.

e.g.

- zzx_Utility() instead of utility()
- zzx_PlaceDialog() instead of placeDialog()
- zzx_TCL instead of tcl()

and so on.

At Sprezzatura we have adopted the following namespaces:

- "ZZX_" for internal core support routines.
- "ZZ_" for individual products and code we intend to release to other developers and clients.
- "WINAPI_" for code relating to Windows API functions.

Revelation themselves use:

- "RTI_"
- "REV_"

Avoid modifying passed parameters in your programs

All parameters passed to stored procedures are passed *by reference*, so if you change a parameter that has been passed to you in your program the change is reflected in the caller.

For this reason do not modify any parameter variable *unless* the purpose of the parameter is to explicitly return information to a caller *and* it is documented as doing so.

Working with constants in your programs

Avoid "Magic Numbers"

Magic numbers are literal numbers such as 100 or 23457 that appear in the middle of a program without any explanation. You should never put these in your code; rather, use the EQU statement to create a named constant in the header of a program and then use the constant name in the code. You should also make yourself aware of insert records supplied with the system that contain useful equates already such as LOGICAL and the FSERROR_xxx series.

Code that looks like this, with magic numbers, is almost impossible to figure out:

```
@file_Error< 1 > = 441
done = 1
if @record< 47 > = 3 then
```

With the numbers replaced by named constants the meaning becomes clear:

```
@file_Error< FSCODE$ > = FS_SYS_SORT_EXTRACT_ERR$
done = TRUE$
if @record< PATIENT_STATUS$ > = PATIENT_IS_DEAD$ then
```

Equate literal strings as well

Literal character strings in code should be treated like magic numbers; if the meaning of the string is not obvious then EQU a named constant for it. Make sure you are aware of any predefined insert records available to you as well.

Don't do this:

```
Call Msg( @window, "", "B234-5" )
```

Do this instead:

```
Call Msg( @window, "", M$CLIPBOARD_EMPTY$ )
```

Literal string limits

The source code of an event handler or stored procedure is limited in size only by the operating system (theoretically 2GB on a 32-bit OS, but in reality this is somewhat less), but there are much more strict limitations imposed by the format of the object code produced by the compiler.

Object code is basically broken down into three sections:

1. A 12-byte header segment
2. A code segment (max 64KB)
3. A data segment (max 64KB)

All of your embedded strings will be placed in the data segment – they are each prefixed by a single byte which contains their length, so no individual embedded string can be larger than 255 bytes in

length, and the total length of all these embedded strings in your program can be no greater than 64KB.

If the total length of the data segment is greater than 64KB the compiler will not raise an error but strings stored beyond the 64KB segment will appear to be corrupt at runtime.

If you have a large number of strings you wish to use in your program then store them in a record in the database and load them dynamically at runtime. This is a common technique used by OpenInsight and Windows itself (aka. resource strings).

Use consistent EQUATE naming conventions

Suffix all equated constants with a "$" character:

```
Equ MAX_PATH$     To 256
Equ SIZE_OF_RECT$ To 32
```

As well as making it obvious in your code that the value is a constant it also helps the compiler to detect mistyped equates and report them at compile time.

I suggest making equated constants all upper-case for two reasons:

1. This is a style familiar to C/C++, Java, and Windows programmers.
2. Some parts of OpenInsight include equates that do not follow the "$" convention (e.g. the OIPRINT_EQUATES insert record), so making the constant names uppercase helps to indicate that they are indeed constant values and not actual variables.

When creating equates for table column positions include the name of the table as well as the column. This ensures that identical column names with different column positions in disparate tables do not clash.

For example, when creating equates for use with the BOOKS table don't do this:

```
Equ TITLE$    To 1
Equ AUTHOR$   To 2
Equ COMMENTS$ To 3
```

Do this instead:

```
Equ BOOKS$TITLE$    To 1
Equ BOOKS$AUTHOR$   To 2
Equ BOOKS$COMMENTS$ To 3
```

(A "Create Insert" tool is supplied with the OpenInsight Table Builder that can be used for this purpose - see the "Working With Dictionaries" section below for more information)

Move equated constants into an insert record

Equated constant declarations should be moved into a separate insert record rather than embedded into an individual program. This means that they can be shared between several stored procedures easily, and makes subsequent changes to the declaration easier to manage.

Working with global variables

Use consistent labelled common naming conventions

Suffix all labelled common variable names with an "@" character:

```
Common /$%ZZX_STUFF%$/ zzxVar1@, zzxVar2@, zzxVar3@
```

This makes it easy to identify global variables used in your programs.

Move labelled common declarations into an insert record

Labelled common declarations should be moved into a separate insert record rather than embedded into an individual program. This means that they can be shared between several stored procedures easily, and makes subsequent changes to the declaration simple to manage.

Create new labelled common declarations slightly larger than initially required

When creating a new Labelled common declaration it is a good idea to add a couple of unused variables to it as "spares". Common areas frequently grow during development and this helps to prevent the "labelled common created smaller than declaration" errors that can be encountered if you forget to compile all the programs that use the common declaration.

Rather than this:

```
Common /$%_ZZX_NEW_%$/ zzxInit@, zzxUsr@
```

Do this:

```
Common /$%_ZZX_NEW_%$/ zzxInit@, zzxUsr@, zzxVar3@, zzxVar4@
```

The runtime overhead of adding the extra variables is negligible.

Beware of corrupting system variables during recursion

If you write recursive programs or use the Yield() function be sure to protect system variables from being corrupted – this is especially important during select list processing where it is easy to break the ReadNext process.

Ensure you protect the following variables:

- @Dict
- @Record
- @Id
- @RecCount
- @Rn_Counter

And use Push.Select and Popup.Select to protect your cursor.

For example we use the following internal subroutine construct in many of our programs to protect these system variables when we need to execute a Yield() operation:

```
SafeYield:

   saveAtDict      = @dict
   saveAtRecord    = @record
   saveAtID        = @id
   saveAtRecCount  = @recCount
   saveAtRnCounter = @rn_Counter

   call Push.Select( A, B, C, D )
   call Yield()
   call Pop.Select( A, B, C, D )

   transfer saveAtDict     to @dict
   transfer saveAtRecord   to @record
   transfer saveAtID       to @id
   transfer saveAtRecCount to @recCount

   * @rn.Counter is not a "real" variable and cannot
   * be 'transfer'd - it is held in the engine internally
   * as a pure integer and not via a descriptor like a
   * normal Basic+ variable.
   @rn_Counter = saveAtRnCounter

return
```

Program Structure

Don't use GOTO

Code containing goto can almost always be rewritten into a structured goto-less form. Without goto the code will be easier to understand, easier to format, and easier to maintain. If you want to see more compelling arguments against the use of goto then see chapter 17 of Code Complete (*this* document will not continue the "Goto Holy War")

Avoid multiple exit points

Try to avoid multiple exit points in your programs and internal subroutines as this can make it difficult to determine exactly which conditions will lead to the "real code" being executed. It also makes it more difficult to implement a single clean-up point before you return.

For example, try to avoid this:

```
Open "SYSENV" To hSysEnv Else
   Call FsMsg()
   Return
End
If ( @appID = "LIBRARY" ) Else
   Call Msg( @window, "Must be the LIBRARY app" )
   Return
End
<...real code goes here...>
```

And structure your code like this instead:

```
Open "SYSENV" To hSysEnv Then
   If ( @appID = "LIBRARY" ) Then
      GoSub DoTheWork
   End Else
      Call Msg( @window, "Must be the LIBRARY app" )
   End
End Else
   Call FsMsg()
End

Return
```

(Although the first example above appears simple to read it is easy for this style of programming to get out of hand and lead to problems)

Don't put multiple statements on one line

Cramming multiple statements onto one line *does not* make the code faster. It only makes the code harder to maintain and read.

Don't do this:

```
For i = 1 To MAX_VAL$; row<2,i> = TRUE$; Next
```

The equivalent readable form is:

```
For i = 1 To MAX_VAL$
   row< 2, i > = TRUE$
Next
```

An exception can be made for trivial operations such as variable initialization. This is ok:

```
srcDeclares    = ""; srcEquates   = ""
srcInitUsers   = ""; srcFirstLast = ""
srcInitBreaks  = ""; srcInitAggs  = ""; srcUpdUsers  = ""
srcUpdAggs     = ""; srcGoBreaks  = ""; srcLastBreak = ""
srcTestBreaks  = ""; srcBandSubs  = ""
```

Also, always use the multiple line form of if-then, even if the body is one line. This makes it easier to add more lines to the body at a later date.

Don't do this:

```
If x = 100 Then eof = TRUE$
```

Do this instead:

```
If x = 100 Then
   eof = TRUE$
End
```

Use the multi-line statement operator when appropriate

Since version 8.0 OpenInsight has supported multi-line statements with the "|" operator. When calling a function that has many long arguments don't try and cram them all onto one line. This makes the code harder to read and usually involves annoying horizontal scrolling.

Don't do this:

```
Call MyFunc( "CREATE", uID, nID, pToSomeBigStructure, sLen )
```

Do this instead:

```
Call MyFunc( "CREATE",            |
             uID,                 |
             nID,                 |
             pToSomeBigStructure, |
             sLen )
```

Don't leave a blank body in a conditional statement

Rather than leave a blank body in a conditional statement use the "null" statement instead. This indicates that the blank body is intentional and that it's not something you forgot to write. The overhead in processing a "null" statement is negligible.

Don't do this:

```
Open "SYSENV" To hSysEnv Then
End Else
   Call FsMsg()
End
```

Do this instead:

```
Open "SYSENV" To hSysEnv Then
   Null
End Else
   Call FsMsg()
End
```

Or this:

```
Open "SYSENV" To hSysEnv Else
   Call FsMsg()
End
```

Beware of inserting code via a $Insert statement

Insert records should really only contain equated constant definitions – in general they should not contain actual code execution statements.

Code inserted into a procedure via a $insert statement suffers from the following disadvantages:

- Inserted code cannot be traced properly with the debugger, making it hard to step through a procedure and possibly obscuring variable initialization and manipulation.
- Changes to the inserted code will force all the hosting stored procedures to need recompiling.
- The BLint() process that the compiler uses to check for suspected unassigned variables will not follow the $insert statement and process the insert record, making compile time errors more difficult to detect.

If you have common code that you need to share between your procedures you should place it within its own stored procedure and call it as a separate function. If you are still intent on including code in an insert record then ensure that the code is:

- Simple
- Well tested
- Unlikely to change

Case statements

When creating Case statements ensure that they are easy to read. If a single case branch is large then move the code to an internal subroutine and call it with a GoSub statement.

Don't do this:

```
Begin Case
   Case ctrlID = "EDL_ID"
      <..50 lines of code here...>
   Case ctrlID = "EDL_TITLE"
      <...90 lines of code here...>
End Case
```

Do this instead:

```
Begin Case
   Case ctrlID = "EDL_ID"
      GoSub onIDLostFocus
   Case ctrlID = "EDL_TITLE"
      GoSub OnTitleLostFocus
End Case
```

Using Comments

Ensure you put *effective* comments into your Basic+ code

We've heard many myths and fables from programmers who are convinced they write self-documenting code and don't need comments. We've also come across many instances where the comments simply describe the code which in most cases makes them redundant, especially when subsequent code updates render them inaccurate (How many programmers correct comments as well as code when fixing bugs?)

Comments should not simply repeat the code verbatim – they should provide clarification on the *intent* of the code.

Well written comments can provide information that is not evident from the code at hand no matter how beautiful or well-written, and this can save many tedious and frustrating hours spent in future modifications or fixes, especially when dealing with very large and dense sections of source code.

We will pour scorn on those who fail to comment their code correctly – there is a special place reserved in hell for their worthless souls.

Prefer single-line comments to multi-line comments for small comment blocks

Don't use multi-line comments for very small comment blocks as it makes large sections of code much harder to "comment out" if necessary (which can happen frequently during code refactoring).

Don't do this:

```
Open "SYSENV" To hSysEnv Then
   /*
      Now get the config information...
   */
   Read cfgRow From hSysEnv, cfgID Then
      /* Do processing */
   End
End Else
   Call FsMsg()
End
```

Do this:

```
Open "SYSENV" To hSysEnv Then
   * Now get the config information...
   Read cfgRow From hSysEnv, cfgID Then
      * Do processing
   End
End Else
   Call FsMsg()
End
```

Ensure you provide a "program" header comment block

When creating a stored procedure you should always ensure you provide a comment block at the top of your programs with the following information (not an exhaustive list):

- Copyright Notice
- Author/Company
- Date created
- Purpose of the Stored Procedure
- Additional comments
- Parameter descriptions
- Return Value description (if any)
- Errors returned
- Revision Notes

E.g.:

```
Function exampleFunc( param1, param2, param3 )
/*
   **Copyright (c) 2010 Sprezzatura Ltd. All rights reserved**

   Author  : Captain C, Sprezzatura Ltd
   Date    : 14 Jan 2010
   Purpose : Example proc to show comment header

   Comments
   ========
   Section for describing the program in more detail, including
   info on warnings, how it fits in with other modules, and so
   on.

   Parameters
   ==========

      param1 -> Stuff coming in
      param2 -> More stuff coming out
      param2 <- Stuff coming out

   Returns
   =======
   TRUE$ if successful, FALSE$ otherwise

   Errors
   ======
   Error information is returned via Set_Status()


   Amended  Version  Date       Reason
   =======  =======  ====       ======
   Mr C     1.0.1    20 Jan 10  Fixed Bug 917

*/
```

Also ensure you mark any changes and revisions to your program within the body of the code as well so other programmers who have to maintain your code can easily see where changes have been made.

Working with Dictionaries

Writing code for calculated columns

When creating calculated columns in your tables don't enter large amounts of code in the formula field – rather you should put the code in a stored procedure and call that instead. This makes the code easier to maintain and far easier to debug.

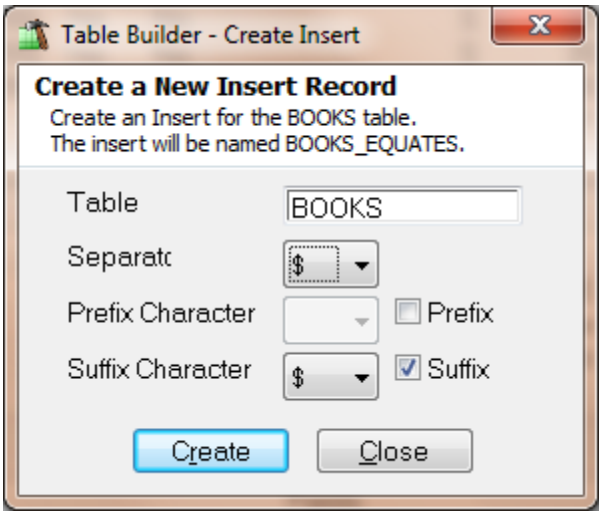
Ensure you use the dictionary description field.

As well as ensuring that you put comments in your Basic+ code you should also ensure you use the Description field when creating dictionary items. Column names are frequently abbreviated or shortened to save typing while constructing query statements and this can obscure their meaning (e.g. FN instead of FORENAME).

There are still places left in hell for programmers who create cryptic column names without a supporting description.

Create an Equated constant insert record for your dictionaries

The OpenInsight Table Builder includes a "Create Insert" tool that allows you to generate an insert record that contains an equated constant for each column defined in a table. Ensure you use it to create and maintain a standard list of constants for use in your code.



Handling Errors

Ensure error conditions are handled

There is a lot of badly written code out there that assumes errors never happen in programs. Perhaps one of the most common constructs we see is the error handling (or lack of) in the Open statement.

NEVER do this:

```
Open "SYSENV" To hSysEnv Else DEBUG
```

or this:

```
Open "SYSENV" To hSysEnv Else Return
```

or this:

```
Open "SYSENV" To hSysEnv Else Null
```

or this (no error trapping at all!)

```
Open "SYSENV" To hSysEnv
```

Ensure you trap and handle all errors properly. At the very least you should be doing something like this:

```
Open "SYSENV" To hSysEnv Then
   < ...Process the table... >
End Else
   Call FsMsg()
End
```

If you really don't want to handle an error condition and you have a good reason for this then at least provide an explanation in your code otherwise you may pick up a reputation as a poor or lazy programmer.

Ideally you should create an error handling framework so you can log and report errors in your program effectively. Development of such a framework, however, is beyond the scope of this document (and rest assured we already have a non-free one if you're interested).

Ensure you know *how* to handle errors

There are at least four different error handling mechanisms in OpenInsight:

1. The Get_Status and Set_Status functions
2. The Get_EventStatus and Set_EventStatus functions
3. The Status() function
4. @file_Error and the Set_FSError function

Ensure you are aware how these different methods work and how to use them effectively. The "Handling Errors In OpenInsight" article in SENL Vol3 Issue 1 covered the rationale behind these and is a good place to start (You can find this article in Appendix A at the end of this document).

Prefer using Get_Status and Set_Status when reporting errors from a Stored Procedure

The Get_Status and Set_Status functions are the core runtime error-reporting mechanisms in OpenInsight and provide a consistent interface for general error-handling. They are used widely within OpenInsight itself and should be adopted by your programs to overcome the usual hodgepodge of arbitrary, ad-hoc error reporting techniques that are usually found out in the wild.

Working with OpenInsight forms

Ensure all controls are properly named

When you create controls with the Form Designer it gives them a default name based on their type and a unique number. Ensure that you change this to something meaningful - Having to use names like “DBCONTROL_53” makes your code difficult to understand.

Ensure all control names are prefixed with their type

It is helpful to prefix control names with a string that identifies their type as it helps to a visual clue as to the properties and capabilities supported by the control when writing code to interact with it. It will also help you write more generic code as it will become part of a naming convention which is useful when writing components like promoted events.

For example for a “Save” button use:

BTN_SAVE

rather than names like:

SAVE, SAVE_BUTTON or BUTTON_23

Below is a list of control types and suggested prefixes:

Control Type	Prefix
Static Text	TXT_
Button	BTN_
EditLine	EDL_
EditBox	EDB_
EditTable	EDT_
ListBox	LST_
ComboBox	CBO_
CheckBox	CHK_
RadioButton	RBN_
GroupBox	GRP_
Bitmap	BMP_
OLE Control	OLE_
Tab Control	TAB_
SplitBar (Vertical)	SPV_
SplitBar (Horizontal)	SPH_
ScrollBar (Vertical)	SBV_
ScrollBar (Horizontal)	SBH_

Prefer the use of Commuter Modules over Event Scripts

Heavy use of event scripts makes application deployment more complex and leads to performance degradation at runtime due to the program stack overflowing more frequently. It also leads to code duplication as it is more difficult to share common code between events.

Event scripts should only be used where there is no other option and even then they should simply call into a commuter module rather than having the code embedded within them.

Don't do this:

```
// "Pre-write" event - need to ensure some data is correct
// before the write is allowed
retVal = 1
dt     = Get_Property( @window : ".EDL_DATE", "INVALUE" )
If dt > Date() Then
   <... do some processing etc...>
End

Return retVal
```

Do this instead:

```
* Assume commuter module has the same name as the
* window

procID = @window[ 1, "*" ]
retVal = Function( @procID( ctrlEntID, "PREWRITE" ) )

Return retVal
```

Adopt a common naming convention for your commuter modules

Commuter modules should have a consistent naming convention. At Sprezzatura we create our commuter modules to have the same name as the forms that they support - For example, if we create an OpenInsight form called ZZ_IDE_APPROW then we create a stored procedure called ZZ_IDE_APPROW as the commuter module for the form as well (Revelation themselves use a similar convention but use the form name suffixed with the string "_EVENTS" as the name of the commuter module instead).

Adopt a common naming convention for your forms

It is good practice to prefix your form names with a type identifier when creating them. As well as helping to create a "namespace" for them and thereby avoid collisions with other system or third party components, it will also do the same for your commuter modules, and make it obvious which stored procedures are actually commuter modules. At Sprezzatura we use prefixes that relate to the "module" that the form is part of - For example most of our general developer tools use the prefix "ZZ_IDE_", and our TCL tools use "ZZ_TCL_". At the very least you should try using a simple prefix such as "FRM_" or "MDI_"

Use concatenated arguments for getting and setting properties

The Get_Property and Set_Property functions support concatenated arguments that allow several properties to be set in a single call which can result in performance improvements.

Don't do this:

```
id  = Get_Property( @window, "ID" )
row = Get_Property( @window, "RECORD" )

Call Set_Property( ctlName, "TEXT", patientName )
Call Set_Property( ctlDOB, "TEXT", birthDate )
```

Do this instead:

```
objxArray =         @window
propArray =         "ID"

objxArray := @rm : @window
propArray := @rm : "RECORD"

dataArray =         Get_Property( objxArray, propArray )

id        =         dataArray[1,@rm]
row       =         dataArray[col2()+1,@rm]


objxArray =         ctlName
propArray =         "TEXT"
dataArray =         patientName

objxArray := @rm : ctlDOB
propArray := @rm : "TEXT"
dataArray := @rm : birthDate

Call Set_Property( objxArray, propArray, dataArray )
```

The vertical layout used above is preferred. We have seen several systems with a horizontal layout (below) which becomes increasingly difficult to read as more items are added:

```
// Do not use this layout!!!
objxArray = ctlName     : @rm : ctlDOB
propArray = "TEXT"      : @rm : "TEXT"
dataArray = patientName : @rm : birthDate
```

Using concatenated arguments – a warning

Although using the @rm concatenation method can lead to improved performance you need to be aware of how it is actually implemented, otherwise it can lead to some subtle bugs that may be hard to track down. This is because internally there are two types of properties; PS (Presentation Server) properties, and Synthetic properties, and the order in which they are processed when combined together could cause your code to behave in a manner that you are not expecting.

- The PS properties are implemented at a low-level inside the Presentation Server (OINSIGHT.EXE) and generally deal with "real" visual properties such as TEXT, SIZE, LIST and so on: basically they wrap the low-level UI attributes of forms and controls. Access to these properties is via two functions called ps_Get_Property and ps_Set_Property. They are not documented but are used internally by the normal Get_Property and Set_Property functions as needed.

- The Synthetic properties are implemented in Basic+ and deal with Form IO logic, data conversion and suchlike; i.e. the things that are specific to OI rather than Windows.

When you *don't* use the @rm concatenation method then Get_Property and Set_Property first try to process the request as a Synthetic property, and, if not found, it is then passed on to the PS. This obviously doesn't cause any issues as no concatenation is involved.

When you *do* use the @rm concatenation method both Get_Property and Set_Property first pass the arrays onto the PS, which deals with the properties it is responsible for. After that the arrays are scanned for any Synthetic properties and processed accordingly.

In many cases this would have no effect unless the order in which the properties are called is critical. Consider the following example:

```
* // Set the TEXT _without_ firing a CHANGED event
* // (This will NOT work!)

objxArray =         "SYSTEM"
propArray =         "BLOCK_EVENTS"
dataArray =         TRUE$

objxArray := @rm : @window : ".EDL_NAME"
propArray := @rm : "DEFPROP"
dataArray := @rm : strName

objxArray := @rm : "SYSTEM"
propArray := @rm : "BLOCK_EVENTS"
dataArray := @rm : FALSE$

Call Set_Property( objxArray, propArray, dataArray )
```

In this case *both* BLOCK_EVENTS property calls (PS properties) will be processed *before* the DEFPROP call (a synthetic property), meaning they will have *no effect* and the CHANGED event will fire. Consider yourself warned!

Avoid the use of shorthand control/property notation in event scripts

If you are forced to use an event script ensure you do not use the old shorthand notation to get and set properties. Rather you should use the full function names in case you ever need to port your code outside of the event script itself. Also the shorthand notation means something else entirely when used outside of an event script (it can be used with the intrinsic Basic+ OLE interface) so it is better not to confuse things.

Don't do this:

```
dt = .edl_date->text
@@window->text = "New Window caption"
```

Do this instead:

```
objxArray =         @window : ".EDL_DATE"
propArray =         "TEXT"

dataArray =         Get_Property( objxArray, propArray )

dt        =         dataArray[1,@rm]


objxArray =         @window
propArray =         "TEXT"
dataArray =         "New Window Caption"

Call Set_Property( objxArray, propArray, dataArray )
```

Only use get and set property within event context

The following functions can only be used in event context. Make sure you don't use them in code shared with your web applications without protecting them with an IsEventContext() call first:

- Get_Property
- Set_Property
- Send_Message
- Utility
- Send_Event
- Post_Event
- Forward_Event

Ensure message dialog boxes are only used in event context too

When displaying messages in code that you share between your desktop and web applications ensure that you check the context first via the IsEventContext() function. *Never* try and use the Msg() or FsMsg() functions while processing a web request or you will hang your web server.

Improving Performance

Use of the loop/remove construct for dynamic array parsing

Using the "< >" operators for sequentially processing large dynamic arrays is a relatively slow operation - every time an element is accessed the array is scanned from the beginning to find the correct location to begin the extract from. Better performance can be gained by using the loop/remove construct to sequentially parse the array instead.

Don't do this:

```
fCount = Count( fArray, @fm ) + ( fArray # "" )
For fNo = 1 To fCount
   fData = fArray< fNo >
   <... process fData ...>
Next
```

Do this instead:

```
pos  = 1
mark = 0
Loop
   Remove fData From fArray At pos Setting mark
   <... process fData ...>
While mark
Repeat
```

One thing worth noting here is that the above use of Loop/Remove is best suited to parsing dynamic arrays that contain only *one* type of system delimiter. This is because the remove statement will return on *any* system delimiter it finds (the "mark" variable indicates the type of delimiter) and this constant checking of mark can add complexity and overhead. A better option is use the "[]" operators as described below.

Use of the "[ ]" operators for string and dynamic array parsing

When you have to access a series of substrings in a large delimited string using the "[]" operators is a much faster way than using the Field() statement in a for/next loop. Field() has to start scanning from the beginning of the string for each access, and this is a relatively slow operation.

For example, don't do this:

```
fCount = Count( fStr, "," ) + ( fStr # "" )
For fNo = 1 To fCount
   fData = Field( fStr, ",", fNo )
   <... process fData ...>
Next
```

Do this instead:

```
// Note the use of the Col2() function to update the position
eof = Len( fStr )
pos = 1
Loop
   fData = fStr[ pos, ”,” ]
   pos   = Col2()+1
   If Len( fData ) Then
      <... process fData ...>
   End
While ( pos < eof )
Repeat
```

Or for parsing through the fields of a dynamic array do this:

```
// Easier than Loop/remove if fArray contains @vm’s
// as well as @fm’s
eof = Len( fArray )
pos = 1
Loop
   fData = fArray[ pos, @fm ]
   pos   = Col2()+1
   If Len( fData ) Then
      <... process fData ...>
   End
While ( pos < eof )
Repeat
```

Use of the Assigned() statement

When testing the state of a variable prefer the use of the intrinsic Basic+ Assigned() statement rather than the Unassigned() stored procedure. Assigned() is implemented as an opcode and is therefore faster.

Unless you are *really really* 110% certain that you will never be passed an unassigned parameter in your programs you should *always* test them with Assigned() before you attempt to use them. We find playing Russian roulette with passed parameters to be a generally unrewarding experience.

Use of the special assignment operators

Always use the special assignment operators (+=,-=,:=) wherever possible as it does not require a separate copy of the variable to be loaded onto the stack prior to resolution.

Don’t do this:

```
text = text : “ and more text”
```

Do this instead:

```
text := “ and more text”
```

Case statements

The Case statement actually compiles down to a series of nested If-Then statements, hence you should check for the most likely condition first.

If you have a large number of conditions that you are testing for then use the Locate-On-GoSub construct instead:

Don't do this:

```
Begin Case
   Case method = "WINMSG"
      GoSub OnWinMsg
   Case method = "OPTIONS"
      GoSub OnOptions
   Case method = "DBLCLK"
      GoSub OnDblClk
End Case
```

Do this:

```
Locate evt In "WINMSG,OPTIONS,DBLCLK" Using "," Setting pos Then
   On pos GoSub OnWinMsg,OnOptions,OnDblClk
End
```

Checking for empty variables.

You should always check for an empty variable by checking its length rather than performing a simple "If-Then" test. This is especially important when dealing with numeric values when "0" is a valid option.

Don't do this:

```
If var1 Then
   <...do stuff...>
End Else
   * We'll get here if var1 is null or 0 !!!
End
```

Do this instead:

```
If Len( Var1 ) Then
   <...do stuff...>
End Else
   * We'll get here only if var1 is null
End
```

Use the Transfer statement

If you are assigning the value of a variable to another and then clearing the original then use the Transfer statement, as this only modifies a descriptor element rather than updating a string in the heap which can be a relatively costly operation.

Don't do this:

```
var2 = var1
var1 = ""
```

Do this instead:

```
Transfer var1 To var2
```

Internationalisation Considerations

Use of binary data manipulation functions with binary structures

When dealing with binary data structures prefer the use of the intrinsic binary manipulation functions rather than the standard Basic+ "character-based" string operators. Using the latter will more than likely produce incorrect results when dealing with applications running in UTF8 mode.

i.e.

- Prefer GetByteSize() instead of Len()
- Prefer GetBinaryData() instead of "= var[a,b]"
- Prefer PutBinaryData() instead of "var[a,b] ="
- Prefer CreateBinaryData instead of Str(s,i)

Use of binary string manipulation functions with text parsing

When parsing/scanning large strings of text prefer the use of the extended "[]" operators, or the BRemove statement introduced in OpenInsight 9.2. Use of these functions will make your application "UTF8-safe" and will not impact your application when running in ANSI mode.

Instead of this:

```
pos   = 1
eof   = GetByteSize( str )

Loop
   subStr = str[pos,@fm]
   pos    = Col2() + 1

   * // process subStr

While ( pos < eof )
Repeat
```

Do this instead:

```
pos   = 1
delim = @fm
dLen  = GetByteSize( delim )
eof   = GetByteSize( str )

Loop
   subStr = str[pos,delim,1] ; * // Extended "[]" operator
   pos    = BCol2() + dLen

   * // process subStr

While ( pos < eof )
Repeat
```

Instead of this:

```
pos  = 1
mark = 1

Loop
   Remove subStr From str At pos Setting mark

   * // process subStr

While mark
Repeat
```

Do this instead:

```
pos   = 1
mark  = 1

Loop
   BRemove subStr From str At pos Setting mark

   * // process subStr

While mark
Repeat
```

Use of Resource Strings

Rather than embedding literal text strings within your stored procedures it is easier to support internationalisation if you store the strings in language-specific data records instead. At runtime you can simply read the correct record based on the user's language and find the correct string to use.

Many Sprezzatura applications and core utility functions use this technique regardless of whether or not they are aimed at the international market because it means strings are easier to update and don't require a stored procedure to be recompiled. Generally we create a core English record in the SYSENV table with a key like ZZX_RESOURCES. Each field in this record contains a text string prefixed with a unique identification code like so:

```
UTL001: Invalid method "%1%" passed to the %2% procedure
UTL002: The sentence contains an unbalanced set of quotes
UTL003: No error codes passed to the %1% method in the %2% procedure
UTL005: %1% %2% error - The "%3%" file already exists
UTL006: No destination file passed to the %1% %2% method
UTL007: No source file passed to the %1% %2% method
UTL008: The %1% %2% procedure cannot process dictionary tables
UTL009: The %1% %2% procedure cannot process index tables
UTL010: No table ID passed to the %1% %2% method
UTL011: No volume ID passed to the %1% %2% method
```

If we wished to offer a German version of these strings then we create a record called ZZX_RESOURCES-DEU that contains the translated strings but with the SAME identifying prefix code.

The strings are then accessed by a custom function that accepts a prefix code, the resource record key, and any arguments that we wish to replace. It also checks to see what language OpenInsight is set to use and tries to find a matching language specific record first, defaulting to the English version if it fails.

E.g.

```
* // Display an error message about unbalanced quotes –
* // zz_GetResStr handles getting the correct language version
* // if appropriate...
errorText = zz_GetResStr( "ZZX_RESOURCES", "UTL002" )
call msg( @window, errorText )
```

Appendix A – Handling Errors in OpenInsight

(Adapted from and originally published in SENL Vol.3 Iss.1 – April 2001)

So just what's with all this Get_Status() and Set_Status() stuff anyway? What was wrong with @file.error and status() like in the good old days eh?

To answer this question we have to take a look at the low-level architecture of OpenInsight, and how it's designed to process information. Back in the early nineties, when RTI first started work on OI, they developed a version of ARev to run natively on the Windows platform and called it OpenEngine. Well, almost. It was designed to be middleware, following a client/server model, and the idea was to allow any Windows application to run Basic+ programs (aka SSP's - System Stored Procedures) and access LH data, and so not all of ARev got ported: The user interface functionality was completely removed, as the idea was this would be supplied by whatever client application that you linked to OpenEngine (examples were supplied of client applications written in C, C++, Object Pascal and VB).

Now to do this a DLL was written called RevCAPI.dll that used DDE to manage communication between a client program and OpenEngine. The basic way this works is that the client uses the functions in RevCAPI.dll to connect to OpenEngine and send it command strings (just like the ones you type in the System Editor Exec line) and then wait for a response.

OpenEngine processes the command and sends back the results to the client application. If the command executed successfully then the client receives a message (DATA_AVAIL as described in the OpenEngine reference manual) telling it everything's OK along with any requested data. If there was an error then OpenEngine sends a special message (PROC_ERROR) to the client along with any information it can find to describe the error. The crucial point here is how to tell OpenEngine to send that PROC_ERROR message?

Well, there are two basic error flags in the Basic+ language, status() and @file.error, but let's face it, their use is not exactly consistent. What if @file.error was null but status() was 1 after a command was executed? What if that was the result of an unsuccessful iconv() function that wasn't critical - is it really an error condition? Does that mean the client should be informed of an error here? What do we do if @file.error is "FS100" and status() is 0? Is that an error? Who knows?

Return values from functions are no use either - there's no defined convention for what values they should be to flag an error, and there are 2 types of SSP (Subroutines and Routines) that don't return a value anyway!

Get_Status() and Set_Status()

To solve this problem the RTI engineers created two new functions, Get_Status() and Set_Status(), that allowed access to a special variable that OpenEngine uses to determine the result of any executed SSP's. If this variable is not empty by the time a command has finished executing then OpenEngine sends back the PROC_ERROR message to the waiting client application along with any text describing the error. Now we have a uniform way of informing the client when an error worthy of its attention has occurred!

So do you need to use them?

At the end of the day, OpenInsight is just another client application that links to OpenEngine (and it's usually referred to as the Presentation Server or PS). However, the difference here is that OpenInsight uses OpenEngine to run special Basic+ SSPs to respond to user-interface events, so it's easy from within your Basic+ user interface code to use status() and @file.error as all your logic is executed *within the context of OpenEngine* and you have full access to them. You'll know when an error has occurred so you can deal with it directly using Basic+ functions and statements to handle the condition, rather than look at see if an error code was flagged in OpenEngine. You don't actually need to worry about informing a separate client application of an error.

However, having said all that, Get_Status() and Set_Status() *do* provide a consistent way of reporting and checking for errors regardless of the execution context, so it's probably a good policy to adopt them in your own code to provide reliable error-trapping mechanism that gels well with the rest of the system.

What's Set_FSError() all about?

This is just a function used to transfer and translate the contents of @file.error across to the error variable used with Get_Status() and Set_Status() so this can be reported in the same way.

So what about Get_EventStatus() and Set_EventStatus() then?

OK, so this leads on to the other pair of functions used for error reporting within Basic+. These were created for use exclusively with OpenInsight and are used when writing Event Code to respond to user interface messages. When the PS needs to run an event like CREATE or READ for example it does so in several stages:

- Run any user-defined Basic+ code first (event scripts)
- Run the default system event handler,
- Run any quick-events,

Now if any of these stages failed for any reason we'd need to know about it, hence the requirement for a way to do this.

We can't use @file.error or status() for the reasons described above, and Get_Status() and Set_Status() are too generic and used for a different purpose, so RTI created another error variable for exclusive use with event handling, accessed with Get_EventStatus() and Set_EventStatus().

Setting this flag at any point in your code will force the event chain to stop. It's also a good way of checking the results of any Forward_Event() calls in your event-handlers such as in Pre and Post Write logic to see if actual write succeeded for example.