CitectSCADA / Vijeo Citect

Cicode Programming Guidelines

Revision History		
Date	**Name**	**Comments**
5/17/2006	Eric Black	Started 1st draft
7/21/2006	Eric Black	Added Tim Van Wyk's comments; expanded coupling section
11/1/2006	Eric Black	2nd Draft
11/3/2006	Eric Black	3rd Draft. Modified control blocks, documentation
4/10/2013	Eric Black	Updated for Citect v7.0 – 7.3. Added Performance section.

Table Of Contents

Cicode Programming Guidelines

Introduction

When writing code, the immediate goal is to 'make it work'. However, the way code is written significantly affects how easily it can be debugged, modified, understood, reused, and how efficient it is. The purpose of this document is to provide guidelines to help find a balance between those goals. Of course, the programmer has to use their own judgment as well, based on project requirements and special circumstances.

The Cicode language is used in the code snippets, but the principles can be adapted to other languages as well.

Structure

Whitespace

Add white space (spaces, tabs, new lines) to code to make it easier to read. This makes functions and blocks of code stand out. Use tabs instead of spaces if possible, and leave the tab width at the default of 4 spaces. Indent nested statements by one tab for each level of nesting. Add two blank lines between functions.

Add one blank line before each block of code within a function. Blocks may be defined by block statements such as IF THEN and SELECT CASE or just a group of related statements. Put a space after commas and before and after operators (+ - / *) etc. Do not put spaces between parentheses or brackets, or after a function name.

```
CreateFoo ( sChar,0,1 );          Poor
CreateFoo(sChar, 0, 1);           Better
```

Functions

Functions should be declared on multiple lines. All statements and declarations in a function should be indented. If the arguments list is too long, it should be wrapped to multiple lines at logical points, with each line indented.

```
PRIVATE
STRING
FUNCTION
GetName(INT nUser)
       <variable declarations>

       <statements>
END

OR…

PRIVATE STRING FUNCTION GetName(INT nUser)
       <variable declarations>

       <statements>
END
```

```
INT FUNCTION PLUSAESum_FileAddEntry(STRING sTime = "", STRING sTimeMS = "",
                                    INT nActive = -1, INT nType = -1, STRING sName = "",
                                    STRING sDesc1 = "", STRING sDesc2 = "",
                                    STRING sCustom1 = "", STRING sCustom2 = "",
                                    STRING sCustom3 = "", STRING sCustom4 = "")
```

If a few functions are very similar and only have one or two commands, it may make them easier to compare if they are each on a single line.

```
//Get and set the current status string
STRING FUNCTION GetStatus()               RETURN msStatus;    END
       FUNCTION SetStatus(STRING sStatus) msStatus = sStatus; END
```

Block Statements

Statements like IF THEN and SELECT CASE should not be condensed to one line. An exception is when there is a group of similar, short block statements and their purpose is clearer if each is on one line. CASE and ELSE statements should be at the same indentation level as the SELECT or IF statement. Blocks of statements should be indented one tab stop.

```
SELCT CASE nMonth
CASE 1
       RETURN "January";
CASE 2
       …
END SELECT
```

ELSE IF statements are not supported in Cicode. Although they can be simulated by changing the whitespace, this should be avoided because it causes multiple end statements with no indenting…making the code less readable.

```
IF (Foo = 1) THEN
       Bar();
ELSE IF (Foo = 2) THEN            Poor
       Baz();
END
END


IF (Foo = 1) THEN
       Bar();
ELSE
       IF (Foo = 2) THEN          Better
              Baz();
       END
END
```

Multiple statements should not be combined on one line.

Line Continuation

Lines should not exceed about 100 characters because code is much harder to read if the user has to scroll back and forth. If a statement spans multiple lines, they should be indented. Splits in an expression should be before the operator.

```
Expr = Value1 + Value2 + Value3 + Value4 + Value5
       + Value 6 + Value7 + Value8;
```

Scope

Declarations of items with different scope should not be mixed together. First declare global variables, then module variables, public functions, and finally private functions.

Naming

Names serve as automatic documentation. Use names that explain what the item is, does, or what units it uses. Avoid names like fn, n1, val, x, etc. Try to use full words instead of abbreviations. Although shortened names may be faster to type, they're harder to read and can be misinterpreted. A few abbreviations may be used if they're frequently used and documented.

Names should be consistent. If the LotNumber field is read from a database, it should be stored in variables like iLotNumber or sLotNumber, not iLotNo, iLotNum, or iBatchNumber.

Capitalization

For variables, use Pascal case (mixed case with the first letter of each word capitalized). The data type prefix should be lower case. Underscores are not needed between words. If a name contains acronyms, only the first letter should be capitalized so it is easy to see where the next word begins.

```
iMixProcessStartTimeSeconds;
sHtmlCode;
```

Label constants defined in the Project Editor | System menu | Labels should be all capitals with underscores to separate the words.

```
DATA_FORMAT_TIME
```

Cicode pseudo-constants use Pascal case with a c prefix. The programmer is expected to refrain from writing to these variables since Citect does not enforce it.

```
INT cLogFile = "log.txt";
```

Files

Cicode functions should be grouped into separate files based on their general purpose or relation to one another.

Standard naming of Cicode files should be defined and followed, based on project requirements. The filename should describe the general purpose of the functions, and may be prefixed with the project abbreviation like 'SEW_Navigation.ci'.

Any Professional Services Plus Cicode files should be prefixed with 'Plus_'.

Variables

Try to avoid using the same generic variables for storing all temporary values (e.g. sTemp). Instead, create a separate variable for each instance. This makes it easier to read the code and see what the purpose of the variable is.

Cicode variables should have a one-letter prefix to indicate the type. This is not necessarily the data type. For example 'h' indicates 'handle', but a database handle is an integer, an ActiveX object handle is an object, and some handles may be strings.

Type	Prefix	Used For
any	c	pseudo-constants
INT	n or i	integer value, count
INT OBJECT STRING	h	handle to a resource or ActiveX object
STRING	s	text
REAL	r	floating-point value
TIMESTAMP	t	tag timestamp
QUALITY	q	tag quality

Do not use the exact same names for local, module, global variables, and variable tags. Although legal, it leads to confusion and bugs. Module and global variables should have an additional prefix ('m' or 'g') before the type prefix. Local variable names can be shorter because they only apply to a single function so part of their purpose is defined by the function's purpose. Module and global variables need more structured names so they don't conflict and their purpose is clear.

Functions

Function names should clearly indicate their purpose within their scope. A private function name can be shorter since it is known to only support the other functions in the same file. Public functions that are specific to a customer's project may be prefixed with the accepted abbreviation for that project or customer such as 'PGE_'. If a function is generic enough to be used with other projects it may be prefixed with 'Plus_' to associate it with the Professional Services Plus functions.

Declarations

Variables

Only one variable should be declared on each line to improve readability. If multiple declarations are listed together the names should be indented the same amount.

```
STRING mcTitle       = "Monthly Report";
STRING mcDescription= "Pump status";
STRING msHeaders[2][50];                    // header table [rows][cols]
INT    miHeaderCount;
```

Global and module variables should be declared at the top of the file, before any functions. Local variables should be declared at the beginning of the function, before any statements. It is not necessary to specify MODULE or LOCAL when declaring these variables.

Initial values may be specified in the declaration. If a statement will write to the variable before it is read by any other statements, there is no need to set an initial value. However, if it will be read before any statements write to it, the initial value should be declared, even if it is 0. The initial value may be a value returned from a function, but it should not be a function that executes an action.

```
INT iResult = 0;

WHILE (iResult <> -1) DO
       iResult = Foo();
END
```

```
INT nStartTime = TimeCurrent();              OK
INT nDevError = DevOpen("LogDevice", 0);     Poor
```

Functions

Public functions do not need to be labeled PUBLIC since it's the default.

In Cicode, the compiler may miss some syntax errors and as the code is modified odd errors may appear. To avoid this, follow these rules:

1. The END keyword must not be omitted from function declarations.
2. Every statement that doesn't end with a keyword (THEN, DO, END, END SELECT, etc.) must end with a semicolon. Exceptions are function declarations and the beginning of a SELECT CASE statement, and CASE statements.

```
STRING FUNCTION QueGetElement(INT hQue, INT nElement, INT nType = 2)      No Semicolon
       STRING sValue;
       STRING sReturn;

       QuePeek(hQue, nElement, sValue, mcQueGetElement);

       SELECT CASE nType                                                  No Semicolon
       CASE 1                                                             No Semicolon
              sReturn = nElement;
       CASE 2                                                             No Semicolon
              sReturn = sValue;
       CASE ELSE                                                          No Semicolon
              sReturn = "";
       END SELECT                                                         No Semicolon

       RETURN sReturn;
END                                                                       No Semicolon
```

Documentation

Comments are important. A function or variable's purpose may be clear to the author but another programmer may have to spend significant time reading the code to understand it. Also, people with minimal programming or project knowledge may need to modify the code in the future.

Comments are only useful if they're accurate. They should be in English with reasonably correct spelling and grammar. They should also be brief. Longer comments are less likely to get updated as the code is changed.

Good comments don't repeat the code or explain it. They clarify its intent. If the purpose of a line of code is unclear it generally should be rewritten. However, sometimes performance is more critical and it may be necessary to clarify the statement or function with a comment.

Single and multi-line comments

Common Cicode practice is to use '//' for inline, single, or multi-line comments and '!' for temporarily disabled lines of code (even though both methods work identically).

The '/* … */' block comment method should be avoided. Although it allows many lines of comments with no prefix on each line, it is harder to distinguish comments from surrounding code, and long comments will not be recognized by the Cicode Editor (See [CtCicode]MLCommentThreshold parameter). The Comment and Uncomment buttons on the format toolbar make it easy to make multiple lines of text or code into a comment using the // method.

Comments should not have closing characters at the end of each line. It only looks neat until someone tries to edit the text.

```
////////////////////////////////////////////////
// MY COMMENT BLOCK                           /
//                                            /
// This is an example of a comment block that's     /
// almost impossible to maintain. Don't do it !!!   /
////////////////////////////////////////////////
```

Files

Files should begin with a comment listing the copyright, filename, and purpose.

```
// Copyright (C) 2013 Schneider Electric
//
// PLUSXml.ci
// Functions for reading/writing XML files
```

Functions

Functions should begin with a header explaining the purpose and use of the function and return value.

```
//Stores data to display in a report
//
//Modified:   5/17/06       Joe White     Fixed time value rounding bug
//
//Arguments:
//
//nRow        Row to set (1 to 300)
//nColumn     Column to set (1 to 50)
//rValue      Value to store (may be LONG, REAL, Boolean, or Citect time/date)
//
//Returns: 0 if successful, otherwise a Cicode error number.
//
//Note:       Although function accepts a REAL, local REALS are 64 bit so it can
//hold a long integer value without losing precision
//
INT FUNCTION MWS_HtmlSetData(INT nRow, INT nColumn, REAL rValue)
```

It is not necessary to repeat the function name in the comment unless it is so long that the function declaration is off the screen. When headers are copied and pasted onto other functions people often forget to change the function name, causing confusion.

If certain sections of the standard header don't apply, delete them. Arguments don't need to be documented if they're self-explanatory and there are no special cases. If a valid range applies, it should be listed, though.

If a function is only called from a specific location, it may be added to the header.

```
//Note:       Called automatically when the user tries to close the page
//            (see Page Environment Variable: CloseWindow)
```

Pseudocode

Pseudocode is a simplified, high-level description of code. It is meant to be less precise and detailed than the actual code but more structured and precise than the function's description. It may be helpful to include pseudocode in the function header if the code is executing a complex algorithm. However, when the code is modified the pseudocode is often left unchanged, causing more confusion than good. If code is written clearly, with occasional comments, pseudocode is not normally necessary.

```
//function quicksort('array')
```

```
//     if length('array') ≤ 1
//           return 'array'  // an array of zero or one elements is already sorted
//     select and remove a pivot value 'pivot' from 'array'
//     create empty lists 'less' and 'greater'
//     for each 'x' in 'array'
//           if 'x' ≤ 'pivot' then append 'x' to 'less' else append 'x' to 'greater'
//     return concatenate(quicksort('less'), 'pivot', quicksort('greater')) //recursive
```

Variables

Variable declarations should be followed by an inline comment if necessary. A descriptive variable name can eliminate the need for a comment in many cases.

Statements

Block statements do not need a comment at the end that restates the original expression because it can lead to incorrect comments if the statements are modified and the comments are not updated. If the expression being evaluated is unclear, an inline comment may be added after the expression.

If there are so many lines of code or nested block statements that it is hard to tell what the END statement refers to, it may be better to move some of the code to separate functions.

```
IF (Foo = TRUE) THEN       // The Foo process is running
      Bar();
END //IF (Foo = TRUE)             Poor
```

To Do

If more is required at a later time to complete a piece of code, it can be marked with one of the following codes and a description:

// TBD	To be determined. Once a decision is made code changes may be needed
// TODO	To be completed
// NOTE	Something noteworthy in the code which may need attention later or may need to be documented
// XREF <location of duplicate>	When a function or structure is duplicated in another file for whatever reason, a maintenance flag needs to be placed in both code locations. This will ensure when one copy is changed, the other copy is updated.

Consistent use of these codes allows text searches to turn up any code that needs to be reviewed before completing a project.

Loose vs. Tight Coupling

The idea of coupling refers to the connection between different functions or systems. The type of coupling affects performance and the ease of making changes without affecting related functions. This is not a black and white matter where one method is good and one is bad or one is loose and the other is tight. There are many degrees of coupling and the following table gives examples comparing looser and tighter methods.

Looser Coupling	Tighter Coupling

Looser Coupling	Tighter Coupling
Functions store data in local variables or their own arrays and pass data using arguments and return values.	Functions share data in module or global variables and arrays.
Each function performs a specific task.	One function determines what task another function will perform by passing flags.
Simple data is passed from one function to another.	Complex data is passed from one function to another, like packing multiple bits into an integer or multiple values into a string.
All programming is done in Cicode.	External code is called via DLLs, ActiveX, DOS commands, etc., adding project dependencies.
Standard, documented API calls are used.	Custom DLL functions are used or undocumented API calls. These may be easier or perform better, but may be broken if software or hardware is upgraded.
Interaction with other software is kept to a minimum.	Code depends on 3rd party software such as OPC servers, ActiveX objects, Message Queueing, IIS, SQL Server, etc. Corrupt or missing installations or installing different versions may break the Citect project.
Some function arguments are optional and have default values.	Function arguments are all required.
Multiple small functions carry out individual tasks	A few large functions carry out multiple related tasks.
Functions pass data through intermediate functions to interact with the user, databases, printers, APIs, etc.	Functions directly access the data source/destination.

Loose coupling means that one function doesn't have to be concerned about the internal implementation of another. A change in one function will require few changes in the other functions that use it. Loose coupling is a sign of a well-structured system. Individual functions make sense without studying all related functions, and they can stand alone. This makes the code easier to understand, maintain, and re-use in other projects. If certain code is constantly changing, it should be loosely coupled with other code.

However, the level of coupling will vary depending on project requirements, such as high performance—which may require tighter coupling. Functions in one file may be more tightly coupled with one-another, but loosely coupled with functions in other files.

For example, in one project data needed to be gathered from dozens of similar groups (stations) and formatted into several HTML reports. Originally, one long function was written to gather all the data, saving it in arrays, and another long function converted it to HTML format and created the report files. This tight-coupling was efficient, but hard to modify and debug. When more stations were added, the data-gathering function had to be rewritten. When an additional report was needed, both functions had to be duplicated with different names and arrays.

Loose coupling was applied in several ways:

1. The functions were rewritten as multiple loosely-coupled functions in two files. One file contained functions to gather data and the other formatted data into HTML code.

2. Instead of both functions accessing the same arrays, they were modified to each store their own data. After data was gathered, it was passed to a function in the other file. This function checked for errors, changed the format, and stored it in another array. Changes to the way data is gathered no longer affect the functions creating the HTML.
3. Instead of hard-coding the list of stations and their configurations into a function, they were moved to a database. A set of functions were created to present the station list to the data-gathering code. This loose coupling means the station list could be moved from its DBF file to a SQL Database or some other format and only the presentation function would be affected, while the data-gathering and HTML functions would be unaffected.
4. Pseudo-constants and intermediate variables were used to replace 'magic' numbers and long expressions, making the code much more readable and self-documenting.
5. Instead of one function taking the data and formatting it directly into HTML code, multiple layers of functions were used (see chart). Each was loosely connected to the others by passing simple values or counters.

	Function Layer
1	Functions aware of the entire report
2	Functions aware of individual sections
3	Functions aware of individual lines
4	Functions aware of individual values

Although this added many functions, it made debugging easier because each function had a well-defined role so it was easy to find the source of problems. All the additional function calls and passing of variables did have a small performance impact, but it was found to be insignificant compared to the benefits.

Performance

Most user Cicode executes quickly and development time can easily be wasted trying to optimize code that has no noticeable impact on the runtime. However some operations can severely impact the entire runtime process and should be considered. Take into account not only how long the code takes to execute but how often it will run and on which runtime process.

To monitor CPU usage of Cicode tasks, open the Citect Kernel in the runtime (DspKernel() Cicode command) and type 'PAGE TABLE CICODE <Enter>' in the Main window. If a task has a high CPU%, it may need to be optimized. Note that operations that hang the runtime while executing may not show as using much CPU—see below for examples.

Infinite Loops

WHILE and FOR loops are often used for processing large groups of values, and WHILE loops may be used to make a function run in an infinite loop in the background. Citect will execute the loop as quickly as possible, and will interrupt (preempt) it as required so other tasks can execute. However, the loop can consume all unused CPU time and make the runtime seem sluggish. Decide whether the loop needs to execute as fast as possible. If not, add the Sleep() or SleepMS() command before the end of the loop so it only executes as fast as needed.

SQL Queries

SQL queries that take a long time to execute or that return large recordsets will hang the runtime process until they complete. It will also hang temporarily if the SQL server is on another computer that is not currently running or connected to the network. This problem is fixed with the ADO.NET support in Citect 7.30. In 7.20 and older versions the problem can be improved by optimizing the query to run faster, such as by adding indexes to the related tables or saving the query in the database as a stored procedure. If a large amount of data is being returned, try limiting the number of records with the WHERE or TOP query commands. Only request the fields that are needed. Or, try executing the query from another process such as the Report server. See the Startup Functions Setup page in the Computer Setup Wizard and the ServerRPC() Cicode function (Citect 7.20 and above) to run code in other processes.

DLL Calls

It is preferable to use a built in command instead of using a DLL or ActiveX object if possible, for performance and reliability. DLL and ActiveX calls will hang the Citect process until they return. They can also crash the runtime.

If a DLL function or ActiveX method is called multiple times, the handle can be saved in a module variable to improve performance. The variable name should reflect the name of the function/method.

```
INT hMakeSureDirectoryPathExists = -1;

INT FUNCTION CreatePath(STRING sPath)
       sPath = PathToStr(sPath);

       IF (hMakeSureDirectoryPathExists = -1) THEN
              hMakeSureDirectoryPathExists = DLLOpen("dbghelp.dll",
                                    "MakeSureDirectoryPathExists", "AC");
       END

       RETURN DLLCallEx(hMakeSureDirectoryPathExists, sPath);
END
```

The DLL/object may provide asynchronous methods to execute commands that take a long time to complete. Another option is to run Citect in multiprocess mode (enabled in the Computer Setup Wizard), and run the offending code in a less critical process, like the Report server. See the Startup Functions Setup page in the Computer Setup Wizard and the ServerRPC() Cicode function (Citect 7.20 and above) to run code in other processes.

File Access

File access is much slower than memory access. If the same data will be needed often and it is not changing, it may be better to read it once and store it in Cicode arrays or queues—see the QueOpen() function.

Trying to read, write, or log to a file on another PC which is not connected to the network can cause the runtime process to hang temporarily. It is best if server processes only log to local files. For example, if the primary and standby Citect servers both log alarms to a file server which has failed, both Citect servers will hang for 30 seconds each time they try to log.

If custom Cicode is constantly writing to files, it can be modified to check for errors and stop trying to write to that file for several minutes to minimize the effect on the runtime. Or, log to a local file and use the FileCopy() function to copy the file to the network location only when needed, so the hang happens

less often. See also the Citect Toolbox item Q1087 CitectSCADA Plus Tools which provides some asynchronous file functions.

Code Tricks

It may seem more efficient to do a complex set of operations in a single long command or to use multiple logic operators instead of nested IF statement blocks. However, the performance increase is likely very little and the code may become much harder to understand and maintain.

A programmer may also find unintended ways to use functions or operators that seem to be more efficient than the normal, documented methods. However, unintended, undocumented functionality can easily be broken by product upgrades or service packs.

String Handling

String manipulation is normally slower than numeric operations. Storing numeric values in strings may be easier at times, but may mean doing string searches or multiple string-to-numeric or numeric-to-string conversions if the value needs to be modified or formatted.

Reading and writing individual characters in a string with StrGetChar() and StrSetChar() may be more efficient than some normal string operations, but normal string operations tend to be much easier to use and require less code to be written. StrGetChar() and StrSetChar() should be reserved for processing buffer values that may contain null characters.

Consider how often the code will run. If it will only run a few times a day (even a few hundred times) and it is easier to work with strings, it may have no impact on the runtime. However, if the code is a low-level function such as formatting a value, it may be called thousands of times in a row such as when processing records from a large database, and could have more impact on the runtime. Functions called from expressions on a graphic page (foreground Cicode tasks) should also be efficient because they may be called hundreds of times a second if there are many objects calling the function.

INI Parameters

Cicode functions often need to read Citect.ini parameters using ParameterGet(). This is very quick for a single read, but can become a problem with many Cicode tasks reading many parameters, or reading them often in a loop. This is even more of a consideration if the hard drive is already busy, such as on a trend server.

One way to avoid this is to read the parameter value once and store it in a global or module Cicode variable. However, to better support online changes it can be good to be able to change parameter values during the runtime. See the Citect Toolbox item Q1347 Citect.ini Parameter Buffer for a function that stores parameter values in memory and only re-reads them from disk after a timeout.

Macro Functions and Include Files

Creating macro (label) functions should be avoided if possible. They are hard to debug since the source code is not shown in the Cicode editor. They make the code less portable to other projects since macros have to be copied from labels.dbf. Macros also cannot be called from the Kernel, MsgRPC(), or TaskNew(). Macro functions are appropriate for doing things that cannot be done in Cicode such as writing back to function arguments.

Include (.cii) files should be avoided if possible for similar reasons. A Cicode function can usually replace an include file that contains Cicode commands. However, there may be no other option when the include file contains data that won't fit in a certain field, such as the format of a Citect System Device.

Constants

Constants (Labels) or pseudo-constants (Cicode variables) should be used to make code more readable, to reduce typing errors, and to make modification easier.

Passing 'magic' numbers to a function makes it hard to tell what the call is doing since you have to look up the values in the documentation.

```
hPen = _ObjectCallMethod(hPens, "Create", 4097, 3);  Unclear
```

Using pseudo-constants means more typing but the code is more readable.

```
INT    cPaPenTypeAnalog = 4097;
INT    cPaPenNameModeCustom = 3;
STRING cCreatePenMethod = "Create";

hPen = _ObjectCallMethod(hPens, cCreatePenMethod, cPaPenTypeAnalog, cPaPenNameModeCustom);
```

If the constant name is mistyped, the compiler will report it. But, if a 'magic' number or hard-coded text string is mistyped the mistake will be harder to find.

Constants may be saved in Labels (Project Editor | System menu | Labels), but this makes it harder to see the value when debugging. It also makes the code less portable to other projects because the constants have to be copied from labels.dbf along with the Cicode. Also, all labels are in one flat list—they are not grouped for organization.

It is preferable to use Cicode variables as pseudo-constants. They can be assigned an initial value, or it can be set by a startup function. They should never be written to after initialization. Pseudo-constants may be defined as local variables if they are only needed in one function, but if they are used in multiple functions they should be module or global variables to avoid duplicate definitions.

Decision Control Blocks

Nesting

IF, WHILE, SELECT CASE, and FOR blocks can be nested inside of one another. Try to avoid many levels of nesting as it makes the procedure harder to follow.

Calling a separate function instead of creating a nested loop also allows breaking out of the loop early to avoid unnecessary processing using the RETURN statement.

IF Statements

It is easier to follow the logic if the expression is checking for the expected result. The unexpected results could be handled later by the ELSE statement.

```
IF (DevEof(hDev)) THEN
       // Do nothing
ELSE
       ProcessRecord();
END

IF (DevEof(hDev) = FALSE) THEN    More readable
       ProcessRecord();
END
```

Select Case

The SELECT CASE block can sometimes replace multiple nested IF statements, making the logic more readable. Always use CASE ELSE as the final case to check for invalid values, even if they are not expected. An error should be logged or displayed. Keep the actions of each case simple—calling other functions if complex handling is needed.

Paths and Filenames

Use pseudo-constants to store paths and filenames instead of hard-coding them each time they're used. For commonly referred-to paths, use Citect path substitution. Any general use function that accepts a path as an argument should use PathToStr() to ensure path substitution is handled.

Scope

Variables

Use the smallest scope you can. It is better to pass a local variable's value from one function to another than to have multiple functions using a module or global variable. Multiple functions writing to the same variable make it more difficult to debug when an incorrect value is written.

Functions

Functions that only support other functions in the same file should be declared as private. This reduces naming conflicts and the need for long, structured names. It also reduces errors since functions that are only meant to be supporting functions can't be called from outside of that file. Note that functions called by TaskNew() or a form callback must be public. In that case the private label can be used to indicate the intent, but must be commented out.

```
// PRIVATE
INT FUNCTION OKButtonCallback()
```

Semaphores

Use semaphores whenever a function uses shared data or a resource that won't allow simultaneous access, like DLL calls. Specify in the comments if semaphores should be used around calls to a set of functions. See the EnterCriticalSection() and Sem*xxx*() functions.

Error Checking & Reporting

Cicode threads should normally begin by calling ErrSet(1). This disables Citect's automatic error checking (and task termination) and allows the code to check the return values of functions for fatal errors. For example the code doesn't call ErrSet(1) and a function call like DevOpen() fails, Citect will terminate the task before the code can check the return value. Disabling automatic error checking is vital for functions that run in an infinite loop, so that they don't get terminated. Do not disable automatic error checking if the code does not check for errors.

Check DLL calls for errors using the IsError() function.

Return error codes from functions even if they're not currently used by the calling function. They are useful for debugging.

Non-interactive error messages should be logged—see the ErrLog() function. A common format is: '<function name>(<argument values>): <error message> Error <error code>'. For private functions that

may have duplicated names, use '<filename>.<function name>(<argument values>): <error message> Error <error code>'

Note that excessive error logging will slow down the runtime. If Cicode will constantly be logging messages, consider using DebugMsg() and DebugMsgSet() to enable/disable logging as needed. Or, use a custom logging function like Citect Toolbox item Q1348 ErrLogEx() that allows different levels of logging to be enabled/disabled in the runtime.

User error messages may be displayed in a popup message box (Message() function). Use the correct icon to indicate the severity (information, warning, or critical stop). The error message and error number, if applicable, should both be displayed in the message.

Expressions

Integer variables may be used to store Boolean (digital) values. When using these in expressions, it is more readable to write an inequality expression instead of using the 'NOT' operator.

```
IF NOT bFoo THEN

IF bFoo = FALSE THEN              More readable
```

If multiple operations are combined in one expression, each operation should be enclosed in parentheses to improve readability and avoid order of operations problems.

```
IF (nMode = 1) OR (bIgnoreMode = TRUE) THEN
```

Use parentheses around 'is equal' statements like: a = (b = c)

A long, complex expression may work efficiently and only take one line of code, but it can be hard to read and debug. Use multiple lines with different local variables to store intermediate values. The intermediate variable names automatically document what each part of the calculation is doing, eliminating manual documentation. This also makes it clear which part of the expression is causing an unexpected result when debugging.

```
Long expression requires extra documentation for clarity
iOutChar = StrToChar(StrMid(sData, nChar, 1)) BITXOR StrToChar(StrMid(sKey, nChar MOD iKeyLength, 1));

Separate expressions are easier to read and self-documenting
iInChar = StrToChar(StrMid(sData, nChar, 1));
iKeyChar = StrToChar(StrMid(sKey, nChar MOD iKeyLength, 1));
iOutChar = iInChar BITXOR iKeyChar;
```

Dead Code

Remove unused code unless it is necessary for future testing. This may include code that is executed but no longer needed, obsolete functions, old or test versions of functions, and commented-out lines of code. This makes the code easier to read and makes it easier to find the correct functions. If it may be necessary to refer to an old version of a function, this should be available from project backups or version control software.

Resource Leaks

Close databases, devices, and other handles when finished with them unless there is a performance or resource issue with re-opening them. If a database is accessed often, it should be left open with a global or module handle. Be careful not to re-open a resource that is already open unless the function specifically states that it will return the existing handle.

References

Source Code	http://en.wikipedia.org/wiki/Category:Source_code
Coupling:	http://www.phptr.com/articles/article.asp?p=349749&seqNum=5&rl=1 http://en.wikipedia.org/wiki/Coupling_%28computer_science%29
Cohesion	http://en.wikipedia.org/wiki/Cohesion_%28computer_science%29
Comments	http://en.wikipedia.org/wiki/Comment
Hungarian Notation:	http://en.wikipedia.org/wiki/Hungarian_notation
Standards:	http://ei.cs.vt.edu/~cs2604/Standards/Standards.html
VB Style:	http://www.htservices.com/Tools/VBandC/VB6Guidelines.doc
Unmaintainable Code Guidelines	http://thc.org/root/phun/unmaintain.html