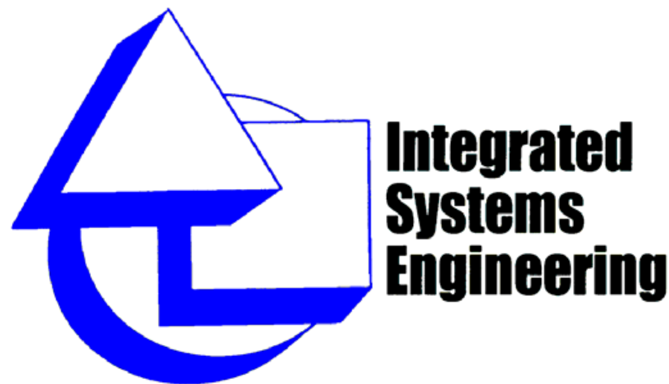
Essential Insight®

Workstation
Scripting Language Reference

Version 8.1



600 South Holmes, Suite 4
St. Louis, Missouri 63122
888-580-6024

Information in this document has been prepared for the sole use licensed users of Essential Insight®. Transmission of all or any part of this information to others, or use for any other purpose is unauthorized without prior written consent of Integrated Systems Engineering, Inc.

This page intentionally left blank

Table of Contents

1.0 OVERVIEW 1

1.1 WORKSCRIPTS 1

1.1.1 Source Editing 1

1.1.2 Source Compilation 1

1.1.3 Workstation Script Execution 1

1.1.4 Workstation Script Variable bUpdate Property 1

1.1.5 Workstation Start Up 2

2.0 LANGUAGE 3

2.1 RESERVED WORDS 3

2.2 VARIABLES 2

2.3 OPERATORS 2

2.3.1 Subtraction – 3

2.3.2 Modulus % 3

2.3.3 Bitwise-AND & 3

2.3.4 Precedence ( ) 3

*2.3.5 Multiplication * 3*

2.3.6 Division / 4

2.3.7 Bitwise-OR | 4

2.3.8 One's complement ~ 4

2.3.9 String Concatenation + ... 4

2.3.10 Addition + ... 4

2.3.11 Less Than < 5

2.3.12 Less Than or Equal To <= 5

2.3.13 Not Equal To <> 5

2.3.14 Assignment = ... 5

2.3.15 Equal To = ... 5

2.3.16 Greater Than > 6

2.3.17 Greater Than or Equal To >= 6

2.3.18 Logical AND AND 6

2.3.19 Negates a Boolean input NOT 6

2.3.20 Logical OR OR 7

3.0 DATA TYPES 8

3.1 SIMPLE DATA TYPE VARIABLE DECLARATIONS 8

3.1.1 BOOLEAN Datatype 8

3.1.2 DATETIME Datatype 9

3.1.3 INTeger Datatype 9

3.1.4 REAL Datatype 9

3.1.5 STRING Datatype 9

3.1.6 TIMER Datatype *10*
3.2 COMPLEX DATA TYPE VARIABLE DECLARATIONS 10
3.2.1 TABLEDATA Data type *11*
3.2.2 LISTARRAY Data type *11*

4.0 CONTROLLING PROGRAM EXECUTION 12

4.1 IF…THEN CONDITIONAL STATEMENTS 12
4.1.1 Single Statement IF *12*
4.1.2 Multiple Statement IF *12*
4.2 LOOP CONTROL STATEMENTS 13
4.2.1 Determinant Loop Control *13*

5.0 BUILT IN FUNCTIONS 14

5.1 STRING MANIPULATION FUNCTIONS 15
5.1.1 ATOF *15*
5.1.2 ATOI *16*
5.1.3 CHAR *16*
5.1.4 FIND *16*
5.1.5 FTOA *16*
5.1.6 ITOA *17*
5.1.7 LEFT *17*
5.1.8 LEN *17*
5.1.9 LOWER *17*
5.1.10 MID *18*
5.1.11 PARSE *18*
5.1.12 PRINT *18*
5.1.13 RIGHT *19*
5.1.14 SPANEXCLUDING *19*
5.1.15 SPANINCLUDING *19*
5.1.16 STRTOHEX *20*
5.1.17 TRIM *20*
5.1.18 TRIMRIGHT *20*
5.1.19 TRIMLEFT *21*
5.1.20 UPPER *21*
5.2 TIME AND DATE FUNCTIONS 21
5.2.1 DAYS *21*
5.2.2 DAYOFWEEK *21*
5.2.3 DAYOFYEAR *22*
5.2.4 HOURS *22*
5.2.5 MINUTES *22*
5.2.6 MONTH *22*
5.2.7 SECONDS *22*
5.2.8 YEAR *22*

5.3 DATABASE TABLE FUNCTIONS 22
5.3.1 WRITETABLE *24*
5.3.2 READTABLE *24*
5.3.3 EDITTABLE *25*
5.3.4 DELETETABLE *25*
5.4 DATABASE QUEUE FUNCTIONS 26
5.4.1 WRITEQUEUE *27*
5.4.2 READQUEUE *28*
5.4.3 EDITQUEUE *28*
5.4.4 DELETEQUEUE *29*
5.4.5 PURGEQUEUE *29*
5.5 DATABASE STORED PROCEDURE EXECUTION FUNCTIONS 29
5.5.1 EXECUTESTOREDPROC *30*
5.6 TIMER FUNCTIONS 31
5.6.1 SETTIMER *32*
5.6.2 STARTTIMER *32*
5.6.3 STOPTIMER *32*
5.6.4 EXPIRETIMER *32*
5.6.5 RESETTIMER *32*
5.7 DEVICE FUNCTIONS 32
5.7.1 DEVWRITE *33*
5.7.2 GETCONNECTIONSTATUS *33*
5.8 WORKSTATION FUNCTIONS 33
5.8.1 WORKSTATIONWRITE *33*
5.9 RANDOM VARIATE FUNCTIONS 34
5.9.1 RAND *34*
5.9.2 RANDMAX *34*
5.9.3 NORMAL *34*
5.9.4 UNIFORM *35*

6.0 COLLECTION DATA TYPES 36

6.1 TABLEDATA PROPERTIES 36
6.1.1 CLEAR Property *36*
6.1.2 COLCOUNT Property *36*
6.1.3 DATA Property *37*
6.1.4 ERROR Property *38*
6.1.5 ERRORMESSAGE Property *38*
6.1.6 ROWCOUNT Property *38*
6.1.7 TABLEDATA Accessors *39*
6.2 LISTARRAY PROPERTIES 40
6.2.1 LISTCLEAR Property *40*
6.2.2 LISTCOUNT Property *40*
6.2.3 LISTADD Property *41*

6.2.4 LISTDELETE Property ... 41
6.2.5 LISTCOMPARE Property ... 41
6.2.6 LISTGETCOMPFLAG Property ... 42
6.2.7 LISTSETCOMPFLAG Property ... 42
6.2.8 LISTARRAY Accessors ... 43

7.0 USER DEFINED FUNCTIONS ... 44

7.1 FUNCTION BODY ... 44
7.1.1 Function Local Variable Declarations ... 44
7.2 FUNCTION ARGUMENTS ... 44
7.2.1 Pass By Value Arguments ... 44
7.2.2 Pass By Reference Arguments ... 45

8.0 COMPLETE WORKSCRIPT EXAMPLES ... 46

8.1 OPERATOR EXAMPLE ... 47
8.2 STRING FUNCTION EXAMPLE ... 52
8.3 DATABASE EXAMPLE ... 53

This page intentionally left blank

1.0 Overview

This document defines the workstation scripting language. The language is syntax is similar to the BASIC language with extensions. The language is used to interpret information from devices, read/write/edit data in database tables, and send control information to devices. The "workscripts" are an integral part of Essential Insight's ability to provide line tracking, error proofing, logging of data and other functionality required by the manufacturing process.

1.1 Workscripts

This section describes workscripts source code entry, compilation, and scheme of execution.

1.1.1 Source Editing

The Essential Insight Studio software application has a built in editor for the Essential Insight workstation scripting language. The workscripts are assigned to associated workstations automatically and are opened for editing within the client. Workscripts may be edited by means of other applications (Notepad, Wordpad, Microsoft Word, etc.), however, the editor within the Essential Insight Studio will format the text for the language. All keywords will appear as blue text, comments will appear as green text, string elements will appear as magenta text, and all other text will be black text. This colorizing of the text greatly improves productivity of the user developing or debugging workscripts.

1.1.2 Source Compilation

The Essential Insight workscripts are compiled to "p-code" for fast interpretation by the Essential Insight workstation. Errors in the workscript syntax are denoted by display of a listing with line numbers and the offending lines noted with error messages. Successful compilation results in a message denoting as such and the creation of the p-code file.

Changes to the source code may be made at anytime (even while the workstation is running!) and the changes compiled at any time. Successful compilation will result in incorporation of the changes the next time the workscript is executed.

1.1.3 Workstation Script Execution

Each workstation script is executed when the workstation receives a message from a device assigned to the workstation or a timer declared in the workstation script expires. Messages from devices are parsed by the device thread and data is assigned to the specified user defined device variables. The assigned data is now available for use when the script executes.

1.1.4 Workstation Script Variable bUpdate Property

Each variable declared in the workstation script has a property associated with it which is accessible using the "dot" operator.

BOOLEAN variable.bUpdate

The property bUpdate is used to determine if a variable has been changed by a message from a device. When a device receives a message, the message is parsed based on the rules defined by the assignment of "device variables". The data is assigned to the device variables declared in the workstation script and the bUpdate property is set to true. Consequently, the bUpdate property can be used in the script to determine which device triggered the execution of the script thereby allowing a specific message handler to take the appropriate actions based on receipt of data from a specific device.

1.1.5 Workstation Start Up

When a user starts a workstation, the workscript is immediately executed to allow initialization of the workstation logic. The workscript is subsequently executed when a device associated with the workstation sends a message to Essential Insight or a timer expires. It is recommended that all scripts contain the following logic to allow appropriate initialization.

```
// Declarations
…
BOOLEAN bStartup = true
…

IF bStartup THEN
        bStartup = false
        // Workstation initialization specific code
ELSE
        // Device message handlers for normal operations
ENDIF
```

2.0 Language

The workscript language syntax is similar to the BASIC language. This section outlines the reserved words, allowable characters in variables and describes the operators available to the programmer.

2.1 Reserved words

The language reserved words are shown in Table 2-1. Variables may not be constructed of character combinations shown in the table. Note all reserved words in the language are case insensitive.

Essential Insight Workscript Language Reserved Words

AND	EXECUTESTOREDPROC	LISTSETCOMPFLAG	SPANEXCLUDING
ATOF	EXPIRETIMER	LOWER	SPANINCLUDING
ATOI	FALSE	MID	STARTTIMER
BOOLEAN	FINDSPANINCLUDING	MINUTES	STEP
BUPDATE	FOR	MONTH	STOPTIMER
CHAR	FTOA	NEXT	STRING
CLEAR	FUNCTION	NORMAL	STRTOHEX
COLCOUNT	GETCONNECTIONSTATUS	NOT	TABLEDATA
DATA	GETCURRENTDATETIME	OR	THEN
DATETIME	GETFIELDCOUNT	PARSE	TIMER
DAYOFWEEK	GETTIMERCOUNT	PRINT	TO
DAYOFYEAR	HOURS	PURGEQUEUE	TRUE
DAYS	IF	RAND	TRIM
DELETEQUEUE	INITQUEUE	RANDMAX	TRIMLEFT
DELETETABLE	INT	READFILE	TRIMRIGHT
DEVWRITE	ITOA	READQUEUE	UNIFORM
EDITQUEUE	LEFT	READTABLE	UPPER
EDITTABLE	LEN	REAL	WORKSTATIONWRITE
ELSE	LISTADD	REF	WRITEFILE
ELSEIF	LISTARRAY	RESETTIMER	WRITEQUEUE
END	LISTCLEAR	RIGHT	WRITETABLE
ENDFUNCTION	LISTCOMPARE	ROWCOUNT	YEAR
ENDIF	LISTCOUNT	SECONDS	
ERROR	LISTDELETE	SENDEMAIL	
ERRORMESSAGE	LISTGETCOMPFLAG	SETTIMER	

Table 2-1

2.2 Variables

Variables must start with an alpha character (A-Z, a-z) or the underscore character (_). The remaining characters of the variable may consist of any combination of the underscore character (_), alpha characters (A-Z, a-z), numeric characters (0-9). There is no length restriction to variable names. Variables names are case sensitive; hence, the variable X is not the same as the variable x. Variables must be declared in the script prior to their use as one of the supported data types (see Section 3.0).

2.3 Operators

The language supports the operators shown in Table 2-2. Subsequent sections provide detailed explanations of each operator and show sample code for each operator's use.

Essential Insight Workscript Operators		
Operator	**Description**	**Data types for which Operations are defined**
-	Subtraction operator	Integer, Real
&	Bit wise AND operator	Integer
()	Precedence Operators	Integer, Real, String, Boolean
*	Multiplication operator	Integer, Real
/	Division operator	Integer, Real
|	Bit wise OR operator	Integer
+	Addition operator	Integer, Real, String
<	Less than operator	Integer, Real, String
<=	Less than or equal to operator	Integer, Real, String, Boolean
<>	Not equal to operator	Integer, Real, String, Boolean
=	Assignment operator	Integer, Real, String, Boolean
=	Equal to operator	Integer, Real, String, Boolean
>	Greater than operator	Integer, Real, String
>=	Greater than or equal to operator	Integer, Real, String, Boolean
AND	Logical AND operator	Boolean
NOT	Negates a Boolean input	Boolean
OR	Logical OR operator	Boolean

Table 2-2

2.3.1 Subtraction –

Subtracts two numbers.

```
nOne     = 1
nTwo     = 2
nThree   = 3
nTotal   = 0
nTotal = (nOne + nTwo + nThree) - nThree
IF nTotal = 3 THEN
         PRINT "Success"
ELSE
         PRINT "Failure"
ENDIF
```

2.3.2 Modulus %

The result of the modulus operator (%) is the remainder when the first operand is divided by the second.

```
nTest    = 10
nModuloResult = 0
nModuloValue = 3
nModuloResult = nTest % nModuloValue
```

2.3.3 Bitwise-AND &

The bitwise-AND operator (&) compares each bit of its first operand to the corresponding bit of its second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

```
nTest    = 10
nModuloResult = 0
nModuloValue = 3
nModuloResult = nTest & nModuloValue
```

2.3.4 Precedence ( )

You can enclose any operand in parentheses without changing the type or value of the enclosed expression.

```
nOne     = 1
nTwo     = 2
nThree   = 3
nTotal   = 0
nTotal = (nOne + nTwo + nThree) - nThree
IF nTotal = 3 THEN
         PRINT "Success"
ELSE
         PRINT "Failure"
ENDIF
```

2.3.5 Multiplication *

Multiplies two expressions (an arithmetic multiplication operator).

```
intA     = 5
intB     = 10
realA    = 5.55
realB    = 10.11
IF ((intA * 2) / 4) > intB THEN
```

```
        boolA = True
ELSE
        boolA = False
ENDIF
```

2.3.6 Division /

Divides one number by another (an arithmetic division operator).

```
intA    = 5
intB    = 10
realA   = 5.55
realB   = 10.11
IF ((intA * 2) / 4) > intB THEN
        boolA = True
ELSE
        boolA = False
ENDIF
```

2.3.7 Bitwise-OR |

The bitwise-inclusive-OR operator (|) compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

```
nTest   = 10
nModuloResult = 0
nModuloValue = 3
nModuloResult = nTest | nModuloValue
```

2.3.8 One's complement ~

The one's complement operator (~), sometimes called the "bitwise complement" operator, yields a bitwise one's complement of its operand. That is, every bit that is set to 1 in the operand is cleared to 0 in the result. Conversely, every bit that is set to 0 in the operand is set to 1 in the result. The operand to the one's complement operator must be an integer type.

```
nTest   = 10
nModuloResult = 0
nModuloValue = 3
nModuloResult = ~ nTest
```

2.3.9 String Concatenation +

An operator in a string expression that concatenates two or more literal strings, variables of string datatype, or functions that return string datatype, into one string expressaion.

```
strValue = RIGHT("1234567890", 5) + " a conversion"
IF strValue + “a test” = “12345 a conversion” THEN
        PRINT "strValue = ", strValue
ENDIF
```

2.3.10 Addition +

Adds two numbers.

```
nOne    = 1
nTwo    = 2
nThree  = 3
```

```
nTotal     = 0
nTotal = (nOne + nTwo + nThree) - nThree
IF nTotal = 3 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
```

2.3.11 Less Than <

Compares two expressions (a comparison operator). When you compare nonnull expressions, the result is TRUE if the left operand has a lower value than the right operand; otherwise, the result is FALSE.

```
real9      = 9.999
real10     = 10.101
IF real9 < real10 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
```

2.3.12 Less Than or Equal To <=

Compares two expressions (a comparison operator). When you compare nonnull expressions, the result is TRUE if the left operand has a lower or equal value than the right operand; otherwise, the result is FALSE.

```
real10  = 10.101
IF real10 <= real10 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
```

2.3.13 Not Equal To <>

Compares two expressions (a comparison operator). When you compare nonnull expressions, the result is TRUE if the left operand is not equal to the right operand; otherwise, the result is FALSE.

```
real9      = 9.999
real10     = 10.101
IF real9 <> real10 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
```

2.3.14 Assignment =

Assigns a value to a variable or property.

```
nSixtyThree        = 63
strA               = "LETTER A"
```

2.3.15 Equal To =

Compares two expressions (a comparison operator). When you compare non null expressions, the result is TRUE if both operands are equal; otherwise, the result is FALSE.

```
ITOA (str9, 9, 2)
ITOA (str10, 10, 2)
ITOA (str11, 11, 2)
IF str10 = str10 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
```

2.3.16 Greater Than >

Compares two expressions (a comparison operator). When you compare nonnull expressions, the result is TRUE if the left operand has a higher value than the right operand; otherwise, the result is FALSE.

```
ITOA (str9, 9, 2)
ITOA (str10, 10, 2)
ITOA (str11, 11, 2)
IF str11 > str10 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
```

2.3.17 Greater Than or Equal To >=

Compares two expressions (a comparison operator). When you compare nonnull expressions, the result is TRUE if the left operand has a higher or equal value than the right operand; otherwise, the result is FALSE

```
real10  = 10.101
real11  = 11.111
IF real11 >= real10 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
```

2.3.18 Logical AND AND

Used to perform a logical conjunction on two expressions. Expressions defined as: A combination of keywords, operators, variables, and constants that yields a string, number, or object. An expression can be used to perform a calculation, manipulate characters, or test data.

```
IF (5 > 2) AND (3 < 4) THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
```

2.3.19 Negates a Boolean input NOT

Negates a Boolean input.

```
realA = 5.55
realB = 10.11
IF NOT(realA > realB) THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
```

2.3.20 Logical OR **OR**

Combines two conditions. When more than one logical operator is used in a statement, OR operators are evaluated after AND operators. However, you can change the order of evaluation by using parentheses.

```
nTwentyOne      = 21
nFortyTwo       = 42
nSixtyThree     = 63
strA    = "LETTER A"
strB    = "LETTER B"
strC    = "LETTER C"
IF (nTwentyOne = nSixtyThree) OR (nFortyTwo = nSixtyThree) THEN
        PRINT "Failure"
ELSE
        PRINT "Success"
ENDIF
IF (strA = strC) OR (strB = strC) THEN
        PRINT "Failure"
ELSE
        PRINT "Success"
ENDIF
```

3.0 Data types

The workstation scripting language supports eight data types; six simple data types, BOOLEAN, DATETIME, INTeger, REAL, STRING, TIMER, and two complex data types TABLEDATA and LISTARRY. Keywords and variable ranges are shown in Table 3.0-1.

Data type	Description	Range
BOOLEAN	Boolean support binary decision	True or False
DATETIME	Date and Time	2003-12-01 00:00:00
INT	Whole number variables	–2147483648 to 2147483647
REAL	Real number variables	1.7E–308 to 1.7E+308
STRING	Character string variables	unlimited array of characters
TIMER	Timer	0.25 sec resolution
TABLEDATA	Database table result set	Defined by table size
LISTARRAY	A list of native data types	Limited by resources of computer
Table 3.0-1		

3.1 Simple Data Type Variable Declarations

The simple data type variables used in a script must be declared using the following form:

DATATYPE <variable name> *{= initial value}*

Note that the initial value is optional. The initial value is used to initialize the variable when it is created in the symbol stack prior to program execution and will be the initial value of the variable when the workscript executes the first time at workstation start.

All simple data type support array declarations using the following format.

DATATYPE <variable name> [index 1][index2][index3]…[index100]

All indexes are zero based, therefore an array declared with 100 elements is accessed from element 0 to element 99.

3.1.1 BOOLEAN Datatype

The BOOLEAN data type variables are declared as
BOOLEAN <variable name> *{= initial value}*

```
BOOLEAN X
BOOLEAN x = TRUE
BOOLEAN bVariable = false
```

The BOOLEAN data type support the use of arrays in the following form;

```
BOOLEAN bArray[5]
BOOLEAN bMultiDimArray[3][4][5][6][7][8][9]
```

There is no current mechanism for initializing arrays as part of the declaration.

3.1.2 DATETIME Datatype

The DATETIME data type is used for date and time variables.

```
DATETIME        dateDateTime
dateDateTime = GETCURRENTDATETIME()
Result
dateDateTime = 2003-02-07 11:00:27
```

3.1.3 INTeger Datatype

The INTeger data type variables are declared as
INT <variable name> *{ = initial value}*

```
INT a
INT B
INT D = 5
INT nThisIsATest = 500000
```

The INT data type support the use of arrays in the following form;

```
INT nOneDimArray[5]
INT nTwoDimArray[6][7]
INT n8DimArray[2][3][4][5][6][7][8][9]
```

There is no current mechanism for initializing arrays as part of the declaration.

3.1.4 REAL Datatype

The real data type variables are declared as
REAL <variable name> *{ = initial value}*

```
REAL X
REAL x = 123.0
REAL dThisIsARealVariable = 2.543E-04
REAL Pi = 2.31415926
```

The REAL data type support the use of arrays in the following form;

```
REAL dReal[5]
REAL dTest[3][4][5][6][7][8][9]
```

There is no current mechanism for initializing arrays as part of the declaration.

3.1.5 STRING Datatype

The string data type variables are declared as

STRING <variable name> *{ = initial value}*

```
STRING X
STRING x = "This is the initial string"
```

The STRING data type support the use of arrays in the following form;

```
STRING strArray[5]
STRING strMultiDimArray[3][4][5][6][7][8][9]
```

There is no current mechanism for initializing arrays as part of the declaration.

3.1.6 TIMER Datatype

The TIMER data type variables are declared as

TIMER <variable name>

```
TIMER tTimer1
```

The TIMER data type support the use of arrays in the following form;

```
TIMER tArray[5]
TIMER tMultiDimArray[3][4][5][6][7][8][9]
```

There is no current mechanism for initializing arrays as part of the declaration.

See Section 5.5 for more information on using timers.

3.2 Complex Data Type Variable Declarations

The complex data type variables used in a script must be declared using the following form:

DATATYPE <variable name>

Note that the initial value of a complex data type cannot be set in the declaration statement. The complex data types are containers for a collection or list of data, consequently the initial value of the variable is an empty collection or list when the workscript executes the first time at workstation start.

The complex data types support array declarations using the following format.

DATATYPE <variable name> [index 1][index2][index3]…[index100]

All indexes are zero based, therefore an array declared with 100 elements is accessed from element 0 to element 99.

3.2.1 TABLEDATA Data type

The TABLEDATA data type is used to retrieve and access data from database tables in an array fashion. By definition there is no initialization of the TABLEDATA type, it is an empty collection of rows and columns which is filled in when a table is read (see Section 5.3.2 for more information about reading the contents of a database table).

The TABLEDATA variables are declared as

TABLEDATA <variable name>

```
TABLEDATA tdTableData1
```

The TABLEDATA data type supports the use of arrays in the following form;

```
TABLEDATA tdArray[5]
TABLEDATA tdMultiDimArray[3][4][5][6][7][8][9]
```

The TABLEDATA data type has properties available for determining the content of the collection and accessors for retrieving the data (see section 6.0 for the usage of the properties and accessors).

3.2.2 LISTARRAY Data type

The LISTARRAY data type defines a collection a values of a data type defined by the LISTARRAY declaration. This type is of specific use when the size of data is unknown at compile time. By definition there is no initialization of a list array as data is added, manipulated, and removed programmatically (see section 6.0 for usage of the properties).

The LISTARRAY variables are declared as

```
STRING LISTARRY <variable name> // a LISTARRAY of string data type values
REAL LISTARRAY <variable name>// a LISTARRAY of real data type values
INT LISTARRAY <variable name>// a LISTARRAY of integer data type values
BOOLEAN LISTARRAY <variable name>// a LISTARRAY of Boolean data type values
TABLEDATA LISTARRAY <variable name>// a LISTARRAY of tabledata data types
LISTARRAY LISTARRAY <variable name>// a LISTARRAY of listarray data type values
```

4.0 Controlling Program Execution

4.1 IF…THEN Conditional Statements

This control structure takes two forms, the single statement form and multiple statement form.

4.1.1 Single Statement IF

The single statement IF is used in the condition when a statement is to be executed only when the expression evaluates to TRUE and no else clause is necessary. ***This form is primarily for backward compatibility (Remove this statement prior to general release of this document)***

IF <expression> **THEN** statement

```
IF A > b THEN PRINT "A = ", A
```

4.1.2 Multiple Statement IF

The multiple statement **IF** allows the execution of multiple statements if the expression evaluates to TRUE and the use of **ELSEIF** and **ELSE** clauses if it does not. This form of the **IF** control allows nesting of **IF**..**ELSEIF**..**ELSE**..**ENDIF** blocks.

IF <expression> **THEN**
statements…
ELSEIF <expression> **THEN**
statements…
ELSE
statements…
ENDIF

```
IF nInteger > 100 THEN
        PRINT nInteger
ELSEIF strString = "This is a test" THEN
        PRINT strString
ELSE
        PRINT "Would you like fries with that?"
ENDIF
```

Nested IF Example:

```
IF nInteger > 100 THEN
        IF (nInteger –100) < 1000 THEN
                nInteger = 5 + 10*nTest
        ELSE
                nInteger = 5 – 10*nTest
        ENDIF
ELSEIF nInteger < nTest/5+100/3 THEN
        …
ELSE
        …
ENDIF
```

There are no restrictions on the number of nested IF blocks.

4.2 Loop Control Statements

Loop control statements consist of determinant loops and indeterminate loops. Determinant loops are FOR..NEXT loops which will execute a predetermined number of times.

4.2.1 Determinant Loop Control

The format of the FOR..NEXT loop is as follows;

```
FOR <variable> = <initial value expression> TO <terminal value expression> STEP <step size expression>
        statements…
NEXT <variable>
```

Note that the **STEP** <step size expression> clause is optional, and when omitted the compiler will default to a step size of 1.

The initial value expression is used to initialize the variable used in the loop. The terminal value expression is evaluated and compared to the variable value at the beginning of the loop. If the variable value is greater than the terminal value expression the loop terminates otherwise the step value is added to the variable and the statements in the loop are executed.

```
FOR i = 1 TO 10 STEP 2
        statements..
NEXT I
```

This loop will execute the statements 5 times with i=1,3,5,7,9.

```
FOR nTest = (nXtream*5 + (12/5))/2 TO nXtream/2
        statements…
NEXT nTest
```

This loop will execute the statements depending on the value of nXtream.

5.0 Built in Functions

This section defines the built in functions used in the scripting language. The functions are listed in alphabetical order in Table 5-1.

Function	Description
ATOF(strRealValue)	See Section 5.1.1
ATOI(strIntegerValue)	See Section 5.1.2
CHAR(nASCII)	See Section 5.1.3
DAYOFWEEK()	See Section 5.2.2
DAYOFYEAR()	See Section 5.2.3
DAYS()	See Section 5.2.1
DELETEQUEUE()	See Section 5.4.4
DELETETABLE()	See Section 5.3.4
DEVWRITE(strDevConn, exp 1, …expN)	See Section 5.7
EDITQUEUE()	See Section 5.4.3
EDITTABLE()	See Section 5.3.3
EXECUTESTOREDPROC	See Section 5.5.1
EXPIRETIMER(tTimerID)	See Section 5.5
FIND(strSource, strSubstring, nStart)	See Section 5.1.4
FTOA(rValue,nDigits, nDigitsAfterDecimal)	See Section 5.1.5
GETCONNECTIONSTATUS()	See Section 5.7
GETCURRENTDATETIME()	See Section 5.2
HOURS()	See Section 5.2.4
ITOA(nSource, nChar)	See Section 5.1.6
LEFT(strExp, nChar)	See Section 5.1.7
LEN(sString)	See Section 5.1.8
LOWER(sString)	See Section 5.1.9
MID(strDest, strExp, nStart, nChar)	See Section 5.1.10
MINUTES(nMinutes)	See Section 5.2.5
MONTH(nMonth)	See Section 5.2.6
NORMAL(rMean, rStdDev)	See Section 5.9.3
PARSE(strSource, strDelimiter)	See Section 5.1.11
PRINT exp1, exp2,… expN	See Section 5.1.12
PURGEQUEUE()	See Section 5.4.5
RAND()	See Section 5.9.1

Function	Description
RANDMAX()	See Section 5.9.2
READQUEUE()	See Section 5.4.2
READTABLE()	See Section 5.3.2
RIGHT(strExp, nChar)	See Section 5.1.13
SECONDS()	See Section 5.2.7
SETTIMER(nID, nPeriod, bAutoReload)	See Section 5.5
SPANEXCLUDING(sSource, sExObj)	See Section 5.1.14
SPANINCLUDING(sSource, sInObj)	See Section 5.1.15
STARTTIMER(nID)	See Section 5.6
STOPTIMER(nID)	See Section 5.6
STRTOHEX(strString, nDigits)	See Section 5.1.16
TRIM(sString)	See Section 5.1.17
TRIMRIGHT(sString)	See Section 5.1.18
TRIMLEFT(sString)	See Section 5.1.19
UNIFORM(rMean, rRange)	See Section 5.9.4
UPPER(sString)	See Section 5.1.20
WORKSTATIONWRITE	See Section 5.8
WRITEQUEUE()	See Section 5.4.1
WRITETABLE()	See Section 5.3.1
YEAR(nYear)	See Section 5.2.8
Table 5-1	

5.1 String Manipulation Functions

5.1.1 ATOF

The ATOF function converts an ASCII string representation of a real number to a real data type. Characters in the string which are non numeric or not a decimal will cause conversion to stop at the offending character position and the result shall be indeterminate.

REAL ATOF (strString)

```
rReal = 0.0
strString = "5.024"
rReal = ATOF (strString)

Result:
rReal = 5.024
```

5.1.2 ATOI

The ATOI function converts an ASCII string representation of an integer number to an integer data type. Characters in the string which are non numeric will cause conversion to stop at the offending character position and the result shall be indeterminate.

INT ATOI (strString)

```
nInteger = 0
strString = "78"
nInteger = ATOI (strString)

Result:
nInteger = 78
```

5.1.3 CHAR

The CHAR function returns the ASCII string value of the integer assigned to a character in the ASCII code.

STRING CHAR(nASCIICharacter)

```
nASCIICharacter = 42  // 'A' character
strString = "abcdefghij"
strString = CHAR (nASCIICharacter)

Result:
strString = "A"
```

5.1.4 FIND

The FIND function returns the starting location of a substring within a source string. If the substring is not found the function returns –1.

INTEGER FIND (strSourceString, strSubString)

```
nStartingLoc = 0
strString = "abcdefghij"
strSubstring = "def"
nStartingLoc = FIND(strString, strSubString)

Result:
nStartingLoc = 3
```

5.1.5 FTOA

The FTOA function converts are real datatype to an ASCII string. The formatting parameters determine the number of digits used in creating the string. The parameter total digits parameter must include the decimal point.

STRING FTOA (rReal, nTotalDigits, nDigitsAfterDecimal)

```
rReal = 123.0987
strString = "abcdefghij"
strString = FTOA (rReal, 7, 2)

Result:
strString = "0123.09"
```

5.1.6 ITOA

The ITOA function converts an ASCII string to an integer datatype. The string will be formatted by the number of digits parameter. If the string representation of the integer value is shorter than the number of digits specified, the string is filled with leading zeros.

STRING ITOA (nInteger, nDigits)

```
nInteger = 123
strString = "abcdefghij"
strString = ITOA (nInteger, 6)

Result:
strString = "000123"
```

5.1.7 LEFT

LEFT extracts a substring from the source string leftmost characters defined by the length parameter and returns a copy of the extracted substring. If the length parameter exceeds the string length, then the entire string is extracted.

STRING LEFT(strSource, nLength)

```
nLength = 3
AString = "ABC123DEF456"
BString = "------------"
BString = LEFT(AString, nLength)

Result:
BString = "ABC"
```

5.1.8 LEN

LEN returns the length of a string as an integer.

INTEGER LEN(strSource)

```
nLength =0
strA = "ABC123DEF456"
nLength = LEN(strA)

Result:
nLength = 12
```

5.1.9 LOWER

The LOWER function returns a copy of the source string converted to lowercase.

STRING LOWER(strSource)

```
strB = “xyz”
strA = "ABC123DEF456"
strB = LOWER(strA)

Result:
strB = “abc123def456”
```

5.1.10 MID

MID extracts a substring of characters from the source string starting at the zero-based starting position, extractd the number of characters defined by the length parameter, and returns a copy of the extracted substring. If the starting position is less than zero a runtime error occurs. If the starting position is greater than the string length, no substring is extracted and the substring is empty (zero length). If the length parameter exceeds the substring length, then the entire substring is extracted.

STRING MID (strSourceString, nStartingPosition, nLength)

```
nStartingPosition = 3
nLength = 6
AString = "ABC123DEF456"
BString = "------------"
BString = MID(AString, nStartingPosition, nLength)

Result:
BString = "123DEF"
```

5.1.11 PARSE

The PARSE function extracts the leftmost substring from a source string up to the defined delimiter character and the remainder of the string is returned in the source string. If the delimiter character is not found in the source string, the entire source string is returned as the substring. The delimiter character is extracted from the source string but not returned in the substring.

STRING PARSE(strSourceString, strDelimiter)

```
strA = “This is a test, of the parse function”
strDelimiter = “,”
strSubString = PARSE(strA, strDelimiter)

Result:
strSubString = “This is a test”
strA = “of the parse function”
```

5.1.12 PRINT

The PRINT function inserts data from the argument list into the workstation output stream and can be viewed using the Essential Insight Studio during workstation operation. This function is used to display variable values in associated workscripts during workscript development. The function prints the contents of the comma delimited expression list, string datatypes are printed verbatim,

integer, real and datetime datatypes are converted to ASCII representations, and boolean values are printed as “true” or “false”. Timer datatypes are not supported.

PRINT exp1, exp2, exp3, … , expN

```
rReal = 123.456
nInteger = 9
strA = “This is a test”
bBoolean = false
PRINT "nInteger: ", nInteger, “ rReal: ”, rReal, “ strA: “, strA, “ bBoolean: “,_ bBoolean

Result:
nInteger: 9 rReal: 123.456 strA: This is a test bBoolean: false
```

5.1.13 RIGHT

RIGHT extracts a substring defined by the lenth parameter from rightmost characters of a string and returns a copy of the extracted substring. If the length parameter exceeds the string length, then the entire source string is extracted.

STRING RIGHT(strSourceString, nLength)

```
nLength = 3
strA = "ABC123DEF456"
strB = "------------"
strB = RIGHT(strA, nLength)

Result:
strB = "456"
```

5.1.14 SPANEXCLUDING

The SPANEXCLUDING function extracts and returns as a string all the characters preceding the first occurance of any character in the character set string.

STRING SPANEXCLUDING(strSourceString, strCharacterSet)

```
strCharacterSet = “;,%”
strA = "This is a test, of span excluding"
strB = "------------"
strB = SPANEXCLUDING(strA, strCharacterSet)

Result:
strB = "This is a test"
```

5.1.15 SPANINCLUDING

The SPANINCLUDING function extracts and returns a substring that contains only characters in included in the character set string, beginning with the first character in the source string and ending with the first character not in the character set string.

STRING SPANINCLUDING(strSourceString, nLength)

```
strCharacterSet = “1234567890P”
strA = "P19238475632%ASLKDFKB"
strB = "------------"
strB = SPANINCLUDING(strA, strCharacterSet)

Result:
strB = "P19238475632"
```

5.1.16 STRTOHEX

The STRTOHEX function returns string containing the ASCII representation of the hexadecimal representation of the source string formatted by the length parameter. If the source string length exceeds the string length parameter, the entire string is converted and returned. If the source string length is less than the length parameter the converted string is packed with leading spaces.

STRING STRTOHEX(strSourceString, nLength)

```
nLength = 10
strA = "ABC"
strB = "------------"
strB = STRTOHEX(strA, nLength)

Result:
strB = "20202020202020414243"
```

5.1.17 TRIM

The TRIM function returns a string with all leading (left side) and trailing (right side) spaces removed from the source string

STRING TRIM(strSourceString)

```
nLength = 10
strA = "   ABC   "
strB = "------------"
strB = TRIM(strA)

Result:
strB = "ABC"
```

5.1.18 TRIMRIGHT

The TRIMRIGHT function returns a string with all leading (left side) and trailing (right side) spaces removed from the source string

STRING TRIMRIGHT(strSourceString)

```
nLength = 10
strA = "   ABC   "
strB = "------------"
strB = TRIM(strA)

Result:
strB = "   ABC"
```

5.1.19 TRIMLEFT

The TRIMLEFT function returns a string with all leading (left side) and trailing (right side) spaces removed from the source string

STRING TRIMLEFT(strSourceString)

```
nLength = 10
strA = "   ABC   "
strB = "------------"
strB = TRIM(strA)

Result:
strB = "ABC   "
```

5.1.20 UPPER

The UPPER function returns a copy of the source string converted to uppercase.

STRING UPPER(strSource)

```
strB = “ABCDEF”
strA = "xyz"
strB = UPPER(strA)

Result:
strB = “XYZ”
```

5.2 Time and Date Functions

The time and date functions provide the capability to manipulate time and date values within the workscripts for applying time/date stamps to data in database tables or queues, and computing elapsed time or comparing time or date data.

5.2.1 DAYS

The DAYS function an integer datatype value specifying a whole number between 1 and 31, inclusive, representing the day of the month.

INT DAYS()

```
nDay = DAYS()
```

5.2.2 DAYOFWEEK

The DAYOFWEEK function returns an integer datatype value representing the numeric day of the week. Sunday is 1, Monday is 2… Saturday is 7.

INT DAYOFWEEK()

```
nDayOfWeek = DAYOFWEEK()
```

5.2.3 DAYOFYEAR

The DAYOFYEAR function returns an integer datatype value between 1 and 366 representing the Julian day of the year. For example January 15 would be 15. February 1 would be 32, etc.

```
nDayOfYear = DAYOFYEAR()
```

5.2.4 HOURS

The HOURS function returns an integer datatype value between 0 and 23, inclusive, representing the hour of the day.

```
nHours = HOURS()
```

5.2.5 MINUTES

The MINUTES function returns an integer datatype value between 0 and 59, inclusive, representing the minute of the hour.

```
nMinutes = MINUTES()
```

5.2.6 MONTH

The MONTH function returns an integer datatype value between 1 and 12, inclusive, representing the month of the year.

```
nMonth = MONTH()
```

5.2.7 SECONDS

The SECONDS function returns an integer datatype value between 0 and 59, inclusive, representing the second of the minute.

```
nSeconds = SECONDS()
```

5.2.8 YEAR

The YEAR function returns an integer datatype value representing the year century and decade (CCDD).

```
nYear = YEAR()
```

5.3 Database Table Functions

Database table access to write, read, edit, and delete records are provided by the **WRITETABLE**, **READTABLE**, **EDITTABLE**, and **DELETETABLE** functions. The functions share a common syntax as follows:

Syntax

Database_Function

(
"*Database_Name*",
"*Table_Name*",
"*Column_List*",
"*Where_Clause_Expression*",
"*Order_By_Expression*",
Error_Code,
Error_Message,
Record_Count,
Column_Variables
)

Arguments

Database_Function

WRITETABLE, READTABLE, EDITTABLE, or DELETETABLE

Database_Name

Specifies the database to access.

Table_Name

Specifies the table to access.

Column_List

The columns to be returned or modified by the function. The column list is a series of string expressions separated by commas which are the names of the columns in the database table.

Where_Clause_Expression

Specifies the conditions for selecting the records to be returned by a READTABLE function. Specifies the rows to be updated by an EDITTABLE function and specifies the rows to be deleted by a DELETETABLE function.

Order_By_Expression

Specifies the sort order of the record set returned by a READTABLE function. The expression must contain column names listed in the column list. Multiple columns can be specified as a comma delimited list, consequently, the order of the columns in the expression define the sort order of the result set.

The ASC keyword may be appended to the order by expression to specify the returned record set should be sorted in ascending order. Conversely, the DESC keyword may be appended to the order by expression to specify the returned record set should be sorted in descending order.

Error_Code

0	No Error
-1	Table Does Not Exist
-2	Invalid Where Clause
-3	Order By Clause Syntax
-4	Field Count Less Than Argument Count
-5	Field Count Greater Than Argument Count

Error_Message
The text of the error message.

Record_Count
The number of records returned by the database function.

Column_Variables
Is one or more column variables to receive the data from the columns specifed in the Column_List separated by commas. There must be a column variable for each column specified in the Column_List

5.3.1 WRITETABLE

The WRITETABLE function will write data directly to the table.

```
WRITETABLE (
        "DatabaseName",_
        "TableName", _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean", _
        "", _
        "", _
        intErrorCode, _
        strErrorMsg, _
        intRetRecCount, _
        strTI_Key, _
        intTI_Integer, _
        numericTI_Numeric, _
        strTI_String, _
        realTI_Real, _
        floatTI_Float, _
        boolTI_Boolean)
```

5.3.2 READTABLE

The READTABLE function is overloaded and has two forms. Each form reads data from a table, however, one format requires data to be in an array, while the other uses the TABLEDATA datatype.
This form will read data from the table. If more than one record is expected to be returned, the variables used to receive the table column data must be declared as arrays.

```
READTABLE (
        "DatabaseName", _
        "TableName", _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean", _
        "TI_Key <> """, _
        "TI_Boolean DESC", _
        intErrorCode, _
        strErrorMsg, _
        intRetRecCount, _
        strTI_Key, _
        intTI_Integer, _
        numericTI_Numeric, _
        strTI_String, _
        realTI_Real, _
        floatTI_Float, _
        boolTI_Boolean)
```

This form will also read data from the specified table, however, all the data is returned in the TABLEDATA datatype as a collection of rows and columns.

```
        READTABLE (string strSelectStatement, TABLEDATA tdTableData)
```

This form is easiest to use and ISE recommends that it be used in all future script development as the other form is planned for deprecation.

5.3.3 EDITTABLE

The EDITTABLE function will modify the record or records in a table that are matched with the Where_Clause_Expression. Only the columns specifed in the Column_List parameter are modified.

```
EDITTABLE (
        "DatabaseName", _
        "TableName", _
        "TI_String", _
        "TI_Key = '000100'", _
        "", _
        intErrorCode, _
        strErrorMsg, _
        intRetRecCount, _
        strTI_String)
```

5.3.4 DELETETABLE

The DELETETABLE function deletes records from a table that match the Where_Clause_Expression criteria.

```
DELETETABLE (
        "DatabaseName",
        "TableName", _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean", _
        "TI_Key = '000101'", _
        "", _
        intErrorCode, _
        strErrorMsg, _
        intRetRecCount)
```

5.4 Database Queue Functions

Essential Insight supports in memory images of database tables called queues. The WRITEQUEUE, READQUEUE, EDITQUEUE, DELETEQUEUE, and PURGEQUEUE function allow writing, reading, editing, deleting, and purging data from a queue. The functions perform the operations on the memory image of a table immediately. The changes to the queue are committed to the database table at intervals defined by an adaptive algorithm. The more changes to the made to a queue, the more often it commits the changes to the database table. Using a queue to write dynamic data is most efficient as the workscript does not block waiting for the database manager to access the table, however, the data may not be available in the table for other applications to access for several seconds. The syntax of the queue functions is identical to the database table functions except for the function name.

Syntax

Database_Function
(
"*Database_Name***",**
"*Table_Name***",**
"*Column_List***",**
"*Where_Clause_Expression***",**
"*Order_By_Expression***",**
*Error_Code***,**
*Error_Message***,**
*Record_Count***,**
Column_Variables
)

Arguments

Database_Function
WRITETABLE, READTABLE, EDITTABLE, or DELETETABLE

Database_Name
Specifies the database to access.

Table_Name
Specifies the table to access.

Column_List

The columns to be returned or modified by the function. The column list is a series of string expressions separated by commas which are the names of the columns in the database table.

Where_Clause_Expression

Specifies the conditions for selecting the records to be returned by a READTABLE function. Specifies the rows to be updated by an EDITTABLE function and specifies the rows to be deleted by a DELETETABLE function.

Order_By_Expression

Specifies the sort order of the record set returned by a READTABLE function. The expression must contain column names listed in the column list. Multiple columns can be specified as a comma delimited list, consequently, the order of the columns in the expression define the sort order of the result set.

The ASC keyword may be appended to the order by expression to specify the returned record set should be sorted in ascending order. Conversely, the DESC keyword may be appended to the order by expression to specify the returned record set should be sorted in descending order.

Error_Code

0	No Error
-1	Table Does Not Exist
-2	Invalid Where Clause
-3	Order By Clause Syntax
-4	Field Count Less Than Argument Count
-5	Field Count Greater Than Argument Count

Error_Message

The text of the error message.

Record_Count

The number of records returned by the database function.

Column_Variables

Is one or more column variables to receive the data from the columns specifed in the Column_List separated by commas. There must be a column variable for each column specified in the Column_List

5.4.1 WRITEQUEUE

The WRITEQUEUE function will write data directly to the queue and the written records are committed to the table at a later time.

```
WRITEQUEUE (
        "DatabaseName",_
        "TableName", _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean", _
        "", _
        "", _
        intErrorCode, _
        strErrorMsg, _
        intRetRecCount, _
        strTI_Key, _
        intTI_Integer, _
        numericTI_Numeric, _
        strTI_String, _
        realTI_Real, _
        floatTI_Float, _
        boolTI_Boolean)
```

5.4.2 READQUEUE

This function is overloaded and has two forms. Each form will read data from the queue, however, one format requires data to be in an array, while the other uses the TABLEDATA datatype..

This form will read data from the queue. If more than one record is expected to be returned, the variables used to receive the table column data must be declared as arrays.

```
READQUEUE(
        "DatabaseName", _
        "TableName", _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean", _
        "TI_Key <> """, _
        "TI_Boolean DESC", _
        intErrorCode, _
        strErrorMsg, _
        intRetRecCount, _
        strTI_Key, _
        intTI_Integer, _
        numericTI_Numeric, _
        strTI_String, _
        realTI_Real, _
        floatTI_Float, _
        boolTI_Boolean)
```

This form will also read data from the specified queue, however, all the data is returned in the TABLEDATA datatype as a collection of rows and columns.

```
READQUEUE (string strSelectStatement, TABLEDATA tdTableData)
```

This form is easiest to use and ISE recommends that it be used in all future script development as the other form is planned for deprecation.

5.4.3 EDITQUEUE

The EDITQUEUE function will modify the record or records in a queue that are matched with the Where_Clause_Expression. Only the columns specifed in the Column_List parameter are modified. The modified records are updated in the database table at a later time.

```
EDITQUEUE (
        "DatabaseName", _
        "TableName", _
        "TI_String", _
        "TI_Key = '000100'", _
        "", _
        intErrorCode, _
        strErrorMsg, _
        intRetRecCount, _
        strTI_String)
```

5.4.4 DELETEQUEUE

The DELETEQUEUE function deletes records from the specifed queue that match the Where_Clause_Expression criteria. The deleted records are removed from the table at a later time.

```
DELETEQUEUE (
        "DatabaseName",
        "TableName", _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean", _
        "TI_Key = '000101'", _
        "", _
        intErrorCode, _
        strErrorMsg, _
        intRetRecCount)
```

5.4.5 PURGEQUEUE

The PURGEQUEUE function removes a row from the queue but does not mark it for deletion from the table at a later time. Records purged from the queue are removed from the memory image of the database table but not from the database table.

```
PURGEQUEUE (
        "DatabaseName", _
        "TableName", _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean", _
        "TI_Key = '000002'", _
        "", _
        intErrorCode, _
        strErrorMsg, _
        intRetRecCount)
```

5.5 Database Stored Procedure Execution Functions

Essential Insight supports calling stored procedures from the relational database management system through Open Data Base Connectivity (ODBC) protocol. This capability supports return values, input variables, output variables, and returned result sets. It is assumed that the workstation script author is completely familiar with the stored procedures available in the relational database management system.

Syntax

{INT nReturnValue = } EXECUTESTOREDPROC

(

"*Database_Name***",**

"*Stored_Procedure_Name***",**

*Error_Code***,**

*Error_Message***,**

*TABLEDATA Returned_Result_Set***,**

Parameter_Variables

)

Arguments

Database_Name

Specifies the database to access.

Stored Procedure Name

Specifies the store procedure to execute.

Error_Code

A variable of Integer datatype to receive the an error number.

0 No Error

-1 Stored Procedure Does Not Exist

-99 Relational Database Management System specific error

Error_Message

A variable String datatype to receive the text of an error message.

Returned_Result_Set

A variable of type TABLEDATA in which a return result set will be stored, if a result set is returned. This variable must be specified even if the stored procedure does not return a result set.

Parameter_Variables

Zero or more parameter variables to provide data to or receive the data from the parameters associated with the specifed stored procedure. There must be a parameter variable for each parameter associated with the specified stored procedure. Literal values are allowed only for input parameters which provide information to the stored procedure.

5.5.1 EXECUTESTOREDPROC

The EXECUTESTOREDPROC function executes the specifed stored procedure. The function may be used in two forms, the first of which is as a standalone function call where a return value from the stored procedure is ignored and not turned to the workstation script. The second form is

as a function call on the right hand side of an equal sign where the return value from the stored procedure is also returned to a workstation script variable. If the stored procedure does not return a value, the workstation script return value will always be -1.

```
EXECUTESTOREDPROC (
        "DatabaseName", _
        "StoredProcedureName", _
        intErrorCode, _
        strErrorMsg, _
        tblReturnedResultSet,_
        nParam1, "LiteralStringParam2", strParam3)
```

or

```
INT nReturnValue = EXECUTESTOREDPROC (
        "DatabaseName", _
        "StoredProcedureName", _
        intErrorCode, _
        strErrorMsg, _
        tblReturnedResultSet,_
        nParam1, "LiteralStringParam2", strParam3)
```

or

```
INT nMyValue = 75 + EXECUTESTOREDPROC (
        "DatabaseName", _
        "StoredProcedureName", _
        intErrorCode, _
        strErrorMsg, _
        tblReturnedResultSet,_
        nParam1, "LiteralStringParam2", strParam3)
```

5.6 Timer Functions

The timer functions are used to set, start, stop, reset, and make timers expire. When a timer associated with a workscript expires, the workscript is executed. This functionality is used to support resetting the workstation state when expected events do not occur in within a specified time, usually indicating an error condition. Timers allow the workscript to reset automatically if an error occurs.
The timer variable name has the bUpdate attribute set when the timer expires and causes the workscript to execute, consequently, the user can determine which timer caused the workscript to execute as follows.

```
TIMER tTimer
…
INT nTimerPeriod = 12          // 3 Seconds
SETTIMER(tTimer, nTimerPeriod, true)
STARTTIMER(tTimer)
…
IF tTimer.bUpdate THEN
        // tTimer expiration handler code goes here
ENDIF
```

5.6.1 SETTIMER

The SETTIMER function is used to set the timer period and to specify if the timer should automatically reload and continue timing after the expiration event. The timer resolution is 0.25 sec, consequently a timer period of 2.5 sec would require nTimerPeriod = 10.
If bAutoReloadAndContinue = true, the timer will expire and reset to the nTimerPeriod and continue to time. This creates a free running timer that will cause the workscript to execute every nTimerPeriod * 0.25 seconds. If bAutoReloadAndContinue = false, the timer will expire and cause the workscript to execute one time and then stop timing. This creates a one shot timer that will expire only once at nTimerPeriod * 0.25 seconds.

SETTIMER(nTimerPeriod, bAutoReloadAndContinue)

```
nTimerPeriod = 12   // 3 Seconds
SETTIMER(tmrTimerID, nTimerPeriod, true)
```

5.6.2 STARTTIMER

The STARTTIMER function is used to start a timer that has been initialized by the SETTIMER function.

```
STARTTIMER(tmrTimerID)
```

5.6.3 STOPTIMER

The STOPTIMER function is used to stop a timer that has been initialized by the SETTIMER function. The timer internal count is not reset and the timer may be started again by the STARTTIMER function to continue timing.

```
STOPTIMER(tmrTimerID)
```

5.6.4 EXPIRETIMER

The EXPIRETIMER function is used to expire a timer causing the workscript to execute on the next timer tick. If the auto reload and continue flag is set to true, the timer will reset and continue timing.

```
EXPIRETIMER(tmrTimerID)
```

5.6.5 RESETTIMER

The RESETTIMER function resets the timer internal counter to the initial timer count. This is used to cause a timer not to time out if an error condition does not exist.

```
RESETTIMER(tmrTimerID)
```

5.7 Device Functions

Messages received from devices are automatically parsed and assigned to device variables. To send data to a device, the process can be reversed, new data can be assigned to device variables

and the variables sent to the device. The device variables are assembled into a message to the device using the same parsing rules defined to parse a message, but the process is reversed. Con

5.7.1 DEVWRITE

The DEVWRITE function is used to "write" messages to devices. The function requires the first argument to be a string expression that evaluates to a valid device connection name followed by a list of device variables to be assembled into the message to be sent to the device.

DEVWRITE (string_connection_expression, device_var_1, device_var_2,…device_var_N)

If the string expression for the connection does not evaluate to a valid device connection name, the system automatically puts an error message into the workstation debug data stream. If the specified devices variables are declared in the workstation script, but not defined in the device parsing rules list, the system automatically puts an error message advising which of the specified device variables are not defined for the device into the workstation debug data stream.

5.7.2 GETCONNECTIONSTATUS

The GETCONNECTIONSTATUS function is used to determine the status of the connection associated with a specified device. The function returns BOOLEAN true if the connection is good, indicating the device socket or internal connection is connected and the device is online. The function also returns by reference the device status and the device socket status as strings.

```
BOOLEAN GETCONNECTIONSTATUS(connection_string_expresssion,
                            string_variable_for_device_status,
                            string_variable_for_device_socket_status)
```

```
BOOLEAN bConnStatus
STRING strDeviceStatus
STRING strDeviceSocketStatus
STRING strConnection = "MyDeviceConnection"
…
bConnStatus = GETCONNECTIONSTATUS(strConnection, strDeviceStatus, strDeviceSocketStatus)
```

5.8 Workstation Functions

Workstation functions are used to communicate between workstations.

5.8.1 WORKSTATIONWRITE

The WORKSTATIONWRITE function is used to send a message from a source workstation to a target workstation such that workstations can exchange data. The first argument is a valid workstation name of the target workstation in the current system configuration. The remaining arguments are variable names which must be declared in both the source and target workstation scripts. The source workstation calls WORKSTATIONWRITE, which causes the values of the

source workstation variables to be sent to the target workstation and the target workstation variable values are updated, the BUPDATE property is set, and the target workstation script is executed.

```
BOOLEAN bValue
INT nValue
STRING strValue
STRING strTargetWorkstation = "Target Workstation"
…
WORKSTATIONWRITE(strTargetWorkstation, bValue, nValue, strValue)
```

5.9 Random Variate Functions

Random variate functions provide random numbers with statistics specified by the function and arguments.

5.9.1 RAND

The RAND function generates a random integer uniformly distributed between 0 and the value returned by RANDMAX (see Section 5.9.2). This function is primarily used as the input random variate generator for the user to create random variates with custom statistics.

```
INT nValue
INT nValue1 = 5
…
nValue = RAND()
nValue = nValue1 + RAND()
```

5.9.2 RANDMAX

The RANDMAX function returns the maximum value returned by the RAND function.

```
INT nValue
INT nValueMax
REAL rValueMax
REAL rValue
REAL rValueScaled
…
nValueMax = RANDMAX()
nValue = RAND()
rValue = nValue
rValueMax = nValueMax
rValueScaled = rValue / rValueMax
```

5.9.3 NORMAL

The NORMAL function returns a random variate of a real datatype with normal statistics with a specified mean and standard deviation.

```
REAL rValue
REAL rMean = 0.0
REAL rStdDev = 1.0
…
rValue = NORMAL(rMean, rStdDev)
```

5.9.4 UNIFORM

The UNIFORM function returns a random variate of a real datatype with uniform statistics with a specifed mean, range.

```
REAL rUniformValue
REAL rMean = 0.0
REAL rRange = 1.0
…
rUniformValue = UNIFORM(rMean, rRange)
```

6.0 Collection Data Types

The collection data types TABLEDATA and LISTARRAY have properties that allow programmatic operations on the data in the collection.

6.1 TABLEDATA Properties

The TABLEDATA data type properties allow the programmatic access the collection of database table information contained within the collection. The collection can be thought of as a two dimensional array of data organized in same format as a database table, except that the data is organized by the declaration of the columns and ORDER BY clauses in the SELECT statement used to populate the collection.

The properties are accessed by use of the "dot" operator after the declared variable name, e.g.

VARIABLE.PROPERTY

Each property may take arguments and return values about the collection.

6.1.1 CLEAR Property

The CLEAR property deletes the contents of the TABLEDATA collection. The CLEAR property does not take any arguments.

BOOLEAN tabledata.CLEAR

```
TABLEDATA tMyResults
BOOLEAN bSuccess
…
bSuccess = tMyResults.Clear
```

The property returns TRUE if the action is successful, FALSE if a runtime error occurred.

6.1.2 COLCOUNT Property

The COLCOUNT property is used to determine the number of columns in a row of the collection and is used to set the upper limit for iterating the columns of any row. The property returns an integer value defining the number of columns.

```
INT i
STRING strData
INT nRow = 0
INT nColumn = 0
TABLEDATA tMyResults
…
nColumn = tMyResults.COLCOUNT
FOR i = 0 TO nColumn – 1
        strData = tMyResults.DATA[nRow][i]
NEXT i
…
```

6.1.3 DATA Property

The DATA property is used to access the data contained in the collection. There are two formats available to access data in the collection.

This first format allows access by ordinal location of the rows and columns:

SIMPLE_VARIABLE = TABLEDATA_VARIABLE.DATA[ROW][COLUMN]

Row order is determined by the ORDER BY clause in the select statement used to populate the collection. The column order is determined by the column declaration order of the select statement used to populate the collection. Attempts to access either rows or columns beyond the row or column count in the collection results in a runtime error. The user is cautioned to use the ROWCOUNT and COLCOUNT properties to determine the limits for iterating the data. NOTE: The row and column variables are zero based, consequently, accessing all the rows or columns requires iterating from 0 to row count – 1 or 0 to column count – 1.

The second format allows access by ordinal location of rows and column name:

SIMPLE_VARIABLE = TABLEDATA_VARIABLE.DATA[ROW][STRING_EXPRESSION]

If the specified string expression for the column name does not exist in the collection, a runtime error occurs.

The simple variable must be declared to be the appropriate type for the data retrieved from the specific column in the database table, as conversion of dissimilar data types (e.g. real to string) is not guaranteed.

```
STRING strColumnName
STRING strData
INT nRow = 0
INT nColumn = 0
TABLEDATA tMyResults
…
strData = tMyResults.DATA[nRow][nColumn]
…
strData = tMyResults.DATA[nRow][“MyColumn”]
…
strColumnName = “MyData”
strData = tMyResults.DATA[nRow][strColumnName]
```

6.1.4 ERROR Property

The ERROR property is used to determine if a database access error occurred when the TABLEDATA collection was populated via READTABLE or READQUEUE function call. The property returns true if an error occurred of false if successful.

```
STRING strMySelectStatement = “SELECT * FROM MyTable”
TABLEDATA tMyResults
…
READTABLE(strMySelectStatement, tMyResults)
IF tMyResults.Error = false THEN
        //        perform normal operations
ELSE
        //        error handler
ENDIF
```

6.1.5 ERRORMESSAGE Property

The ERRORMESSAGE property is used in conjunction with the ERROR property to determine the source of the error when populating the TABLEDATA collection via a READTABLE or READQUEUE function call. The property returns a string containing the error message from the database manager.

```
STRING strMySelectStatement = “SELECT * FROM MyTable”
STRING strErrorMsg
TABLEDATA tMyResults
…
READTABLE(strMySelectStatement, tMyResults)
IF tMyResults.Error = false THEN
        //        perform normal operations
ELSE
        //        error handler
        strErrorMsg = tMyResults.ERRORMESSAGE
ENDIF
```

6.1.6 ROWCOUNT Property

The ROWCOUNT property is used to determine the number of rows in the collection and is the upper limit for iterating through the rows of the collection.

```
TABLEDATA tMyResults
INT nRowCount = 0
…
READTABLE(strMySelectStatement, tMyResults)
nRowCount = tMyResults.ROWCOUNT
```

6.1.7 TABLEDATA Accessors

The TABLEDATA collection represents a result set based on a SQL query. The TABLEDATA datatype is a read only datatype, meaning that an accessor of the datatype cannot appear on the left hand side of an assignment statement. Since it represents a result set, it is only populated when passed as a parameter to the intrinsic functions READTABLE or READQUEUE. All previous data is lost on subsequent calls to READTABLE and READQUEUE.

The contents of the table data collection are accessed either by row index and column index, to allow iteration of the rows and columns in the result set. The column index reflects the column order specified in the SQL select statement and is zero based.

<variable> = <TABLEDATA Datatype>[row index][column index]

```
STRING strMySelectStatement = “SELECT * FROM MyTable”
STRING strErrorMsg
TABLEDATA tMyResults
INT nRowCount = 0
INT nColCount = 0
INT i
INT j
…
READTABLE(strMySelectStatement, tMyResults)
IF tMyResults.Error = false THEN
        nRowCount = tMyResults.ROWCOUNT
        nCOLCount = tMyResults.COLCOUNT
        FOR i = 0 TO nRowCount – 1
                FOR j = 0 TO nColCount – 1
                        nMyIntValue = tMyResults.[i][j]
                NEXT j
        NEXT i
ELSE
        //      error handler
        strErrorMsg = tMyResults.ERRORMESSAGE
ENDIF
```

Alternatively, a row and column may be accessed by specifying the row index and the column name.

<variable> = <TABLEDATA Datatype>[row index][“<column name>”]

```
STRING strMySelectStatement = "SELECT * FROM MyTable"
STRING strErrorMsg
TABLEDATA tMyResults
INT nRowCount = 0
INT nColCount = 0
INT i
…
READTABLE(strMySelectStatement, tMyResults)
IF tMyResults.Error = false THEN
        nRowCount = tMyResults.ROWCOUNT
        FOR i = 0 TO nRowCount – 1
                nMyIntValue = tMyResults.[i]["MyIntValueColumn"]
        NEXT i
ELSE
        //      error handler
        strErrorMsg = tMyResults.ERRORMESSAGE
ENDIF
```

6.2 LISTARRAY Properties

The LISTARRAY data type properties allow the programmatic access the collection of data contained within the collection. The collection can be thought of as a one dimensional array of data organized as a list. The LISTARRAY data type is used in the instances where an array is not acceptable as the number of elements to be stored is unknown at compile time.

The properties are accessed by use of the "dot" operator after the declared variable name, e.g.

VARIABLE.PROPERTY

Each property may take arguments and return values about the collection.

6.2.1 LISTCLEAR Property

The LISTCLEAR property deletes the contents of the LISTARRAY collection. The LISTCLEAR property does not take any arguments.

BOOLEAN listarray.LISTCLEAR

```
INT LISTARRAY lnMyArray
BOOLEAN bSuccess
…
bSuccess = lnMyArray.LISTCLEAR
```

The property returns true if the list is successfully cleared, false if a runtime error occurs.

6.2.2 LISTCOUNT Property

The LISTCOUNT property returns an integer which is the number of elements in the LISTARRAY collection. The LISTCOUNT property takes no arguments.

INTEGER listarray.LISTCOUNT

```
STRING LISTARRAY lstrMyArray
INT nListElementCount
…
nListElementCount = lstrMyArray.LISTCOUNT
```

6.2.3 LISTADD Property

The LISTADD property allows elements to be programmatically added to the collection. The property takes a single argument of the data type to be added to the collection. The property returns an integer reflecting the number of elements in the collection after the addition of the new element. The LISTADD property always adds the new element to the end of the collection, and the list element compare flag is set to TRUE by default.

INTEGER listarray.LISTADD(appropriate_data_type_expression)

```
STRING LISTARRAY lstrMyArray
INT nListElementCount
…
nListElementCount = lstrMyArray.LISTADD(strMyString)
```

6.2.4 LISTDELETE Property

The LISTDELETE property allows elements to be programmatically removed from the collection. The property takes a single argument, an integer representing the ordinal position of the element to remove. The property returns an integer reflecting the number of elements in the collection after the deletion of the specified element.

INTEGER listarray.LISTDELETE(integer_expression)

```
STRING LISTARRAY lstrMyArray
INT nElementToDelete
INT nListElementCount
…
nListElementCount = lstrMyArray.LISTDELETE(nElementToDelete)
```

6.2.5 LISTCOMPARE Property

The LISTCOMPARE property allows comparison of the elements in the collection to a specified expression. The property takes a single argument, an expression appropriate to the data type of the LISTARRAY which is compared to each of the elements in the collection in ordinal order (e.g. first to last). If any element in the collection matches the specified argument, the property returns the element number of the element that matched, otherwise it returns -1. The returned element position is zero based, that is 0 is the first element and LISTCOUNT – 1 is the last element. Data conversion for arguments other than real to integer and integer to real are not attempted and a

runtime error will occur if the specified argument data type is not the data type of the LISTARRAY.

INTEGER listarray.LISTCOMPARE(expression)

Each element in the list array has a Boolean flag which specifies if the element will be considered during a comparison operation. If the comparison flag is set to FALSE, the element will be excluded from the comparison operation. If the comparison flag is set to TRUE, the element will be included in the comparison operation. The comparison flag of an element is set by default to true when the element is added to the list.

A STRING LISTARRAY has special properties. Each string element in the collection is compared character by character to the specified argument string expression. In order to facilitate the matching of specifically formatted strings, the LISTCOMPARE function allows certain characters to be construed as “wildcard” characters. This allows pattern matching of strings when appropriate.

The percent sign character (%) is treated as a wildcard character when embedded in a string element, meaning that any character in the specified string expression at that character position will be considered a match. Likewise, an ampersand character (&) is treated as an alpha character wildcard character and a pound sign (#) is treated as a numeric wildcard character.

6.2.6 LISTGETCOMPFLAG Property

The LISTGETCOMPFLAG property returns the state of the element comparison flag.

```
INT LISTARRAY lnMyArray
BOOLEAN bCompFlagState
INT nElement
…
bCompFlagState = lnMyArray.LISTGETCOMPFLAG(nElement)
…
```

6.2.7 LISTSETCOMPFLAG Property

The LISTGETCOMPFLAG property sets the state of the element comparison flag, it return TRUE if the element was found an successfully modified.

```
INT LISTARRAY lnMyArray
BOOLEAN bCompFlagState
BOOLEAN bSuccess
INT nElement
…
bSuccess = lnMyArray.LISTSETCOMPFLAG(nElement) = FALSE
…
```

6.2.8 LISTARRAY Accessors

The list array contents are accessed by the index of the elements in the list. Note that the index is zero based. The accessor has the following syntax,

<variable> = <LISTARRY Datatype>[index]

```
STRING LISTARRAY lstrMyArray
INT nListElementCount
INT i
…
nListElementCount = lstrMyArray.LISTCOUNT
FOR I = 0 TO nListElementCount – 1
        strMyString = lstrMyArray[i]
NEXT i
```

Conversely, an element within the collection may be altered by using the list array accessor on the left side of an assignment statement. The index of the element must be valid, in that the element specified must be on the list or an error occurs.

```
STRING LISTARRAY lstrMyArray
INT nListElement
…
lstrMyArray.LISTADD("HisString")
lstrMyArray.LISTADD("MyString")
lstrMyArray.LISTADD("HerString")
…
nListElement = lstrMyArray.LISTCOMPARE("MyString")
IF nListElement > -1 THEN
        lstrMyArray[nListElement] = "YourString"
ENDIF
```

7.0 User Defined Functions

The Essential Insight scripting language supports the use of user defined functions to facilitate object oriented code design and code reuse.

7.1 Function Body

Each user defined function has the following defined syntax.

```
FUNCTION DATATYPE FunctionName(DATATYPE Argument 1, DATATYPE Argument
2,…)
        RETURN return value
ENDFUNCTION
```

A user function body must appear after the END statement associated with the workstation script main routine. The keyworks FUNCTION and ENDFUNCTION deliniate the start and end of the function body. The RETURN keyword is required as functions by definition return a value. The value returned may be any of the simple datatypes defined in Section 3.1.

7.1.1 Function Local Variable Declarations

A variable of any datatype defined in Section 3.0 may be defined and used within the scope of the function. Variables defined within the function are local in scope and exist only when the function has scope. Consequently, if the value of a variable in the function must be retained after the function returns, it is necessary to return the variable contents to the caller as the return value or by reference. This method preserves the value for the next time the function is called.

```
FUNCTION BOOLEAN MyFunction()
        BOOLEAN bReturnValue = false
        RETURN bReturnValue
ENDFUNCTION
```

The example above shows the declaration and initialization of a local BOOLEAN variable used as the return value.

7.2 Function Arguments

A variable of any datatype defined in Section 3.0 may be passed to the function via the formal argument list.

7.2.1 Pass By Value Arguments

Arguments defined without the keyword REF are pass by value arguments. This means the value passed by the caller is copied onto the frame of the function and can be manipulated by the function without fear of corrupting the callers value of the argument. The following example demonstrates the use of a call by value variable.

```
bFunctionReturnValue = MyFunction(nIntegerValue)

…

FUNCTION BOOLEAN MyFunction (INT nCount)
        BOOLEAN bReturn = false
        IF nCount > 0 THEN
                bReturn = true
                nCount = -1
        ELSE
                bReturn = false
        ENDIF
        RETURN bReturn
ENDFUNCTION
```

The function examines the value of nCount and returns true if nCount is greater than 0, false otherwise. Note that the local value of nCount is modified, but not returned to the caller, consequently, the value of nIntegerValue in the caller is not modified when the function returns.

7.2.2 Pass By Reference Arguments

To define arguments that can be modified by the function and returned to the caller, use the REF keyword in the formal argument list of the function body. The following function demonstrates use of call by reference.

```
bFunctionReturnValue = MyFunction(REF nIntegerValue)

…

FUNCTION BOOLEAN MyFunction (REF INT nCount)
        BOOLEAN bReturn = false
        IF nCount > 0 THEN
                bReturn = true
                nCount = -1
        ELSE
                bReturn = false
        ENDIF
        RETURN bReturn
ENDFUNCTION
```

8.0 Complete WorkScript Examples

These workscripts are working examples, used in our development phase.

8.1 Operator example

```
BOOLEAN         boolA, boolB
INT       nTwentyOne, nFortyTwo, nSixtyThree
INT       nTest, nModuloResult, nModuloValue
INT       nOne, nTwo, nThree, nTotal, intA, intB
REAL      real9, real10, real11, realA, realB
STRING  str9, str10, str11, strA, strB, strC
PRINT " -----------------------  = (Equals) ----------------------------"
// ---------------------------------------------------------------------
// = (Equals)
// Compares two expressions (a comparison operator).
// When you compare nonnull expressions,
// the result is TRUE if both operands are equal;
// otherwise, the result is FALSE.
// ---------------------------------------------------------------------
real9       = 9.999
real10  = 10.101
real11  = 11.111
ITOA (str9, 9, 2)
ITOA (str10, 10, 2)
ITOA (str11, 11, 2)
IF real10 = real10 THEN
          PRINT "Success"
ELSE
          PRINT "Failure"
ENDIF
IF str10 = str10 THEN
          PRINT "Success"
ELSE
          PRINT "Failure"
ENDIF
PRINT " -----------------  > (Greater Than) ----------------------------"
// ---------------------------------------------------------------------
// > (Greater Than)
// Compares two expressions (a comparison operator).
// When you compare nonnull expressions, the result is TRUE
// if the left operand has a higher value than the right operand;
// otherwise, the result is FALSE.
// ---------------------------------------------------------------------
IF real11 > real10 THEN
          PRINT "Success"
ELSE
          PRINT "Failure"
ENDIF
IF str11 > str10 THEN
          PRINT "Success"
ELSE
          PRINT "Failure"
ENDIF
PRINT " --------------------  < (Less Than) ----------------------------"
// ---------------------------------------------------------------------
// < (Less Than)
// Compares two expressions (a comparison operator).
// When you compare nonnull expressions, the result is TRUE
// if the left operand has a lower value than the right operand;
// otherwise, the result is FALSE.
// ---------------------------------------------------------------------
IF real9 < real10 THEN
          PRINT "Success"
ELSE
```

```
        PRINT "Failure"
ENDIF
PRINT " ----------------  <> (Not Equal To) ----------------------------"
// ---------------------------------------------------------------------
// <> (Not Equal To)
// Compares two expressions (a comparison operator).
// When you compare nonnull expressions, the result is TRUE
// if the left operand is not equal to the right operand;
// otherwise, the result is FALSE.
// ---------------------------------------------------------------------
IF real9 <> real10 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
PRINT " ----------------  >= (Greater Than or Equal To) ----------------"
// ---------------------------------------------------------------------
// >= (Greater Than or Equal To)
// Compares two expressions (a comparison operator).
// When you compare nonnull expressions, the result is TRUE
// if the left operand has a higher or equal value than the right operand;
// otherwise, the result is FALSE.
// ---------------------------------------------------------------------
IF real10 >= real10 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
IF real11 >= real10 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
PRINT " ----------------  <= (Less Than or Equal To) -------------------"
// ---------------------------------------------------------------------
// <= (Less Than or Equal To)
// Compares two expressions (a comparison operator).
// When you compare nonnull expressions, the result is TRUE
// if the left operand has a lower or equal value than the right operand;
// otherwise, the result is FALSE.
// ---------------------------------------------------------------------
IF real10 <= real10 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
IF real9 <= real10 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
// ---------------------------------------------------------------------
// Note you must convert Interger to ASCII before you drop into string compares.
// ---------------------------------------------------------------------
IF 5 > 2 AND 3 < 4 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
PRINT " ----------------  ( ) Expressions in Parentheses ---------------"
// ---------------------------------------------------------------------
// ( ) Expressions in Parentheses
// You can enclose any operand in parentheses without changing
// the type or value of the enclosed expression.
// ---------------------------------------------------------------------
PRINT " ----------------  Logical AND Operator -------------------------"
```

```
// ---------------------------------------------------------------------
// Logical AND Operator
//
// Used to perform a logical conjunction on two expressions.
// Expressions defined as: A combination of keywords,
// operators, variables, and constants
// that yields a string, number, or object.
// An expression can be used to perform a calculation,
// manipulate characters, or test data.
// ---------------------------------------------------------------------
IF (5 > 2) AND (3 < 4) THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
nTwentyOne      = 21
nFortyTwo       = 42
nSixtyThree     = 63
strA    = "LETTER A"
strB    = "LETTER B"
strC    = "LETTER C"
PRINT " ----------------  Logical OR Operator --------------------------"
// ---------------------------------------------------------------------
// OR
// Combines two conditions.
// When more than one logical operator is used in a statement,
// OR operators are evaluated after AND operators.
// However, you can change the order of evaluation
// by using parentheses.
// ---------------------------------------------------------------------
IF (nTwentyOne = nSixtyThree) OR (nFortyTwo = nSixtyThree) THEN
        PRINT "Failure"
ELSE
        PRINT "Success"
ENDIF
IF (strA = strC) OR (strB = strC) THEN
        PRINT "Failure"
ELSE
        PRINT "Success"
ENDIF
strA = "ABCDE"
strB = "FGHIJ"
boolA   = True
boolB   = False
intA    = 5
intB    = 10
realA   = 5.55
realB   = 10.11
IF ((intA * 2) / 4) > intB THEN
        boolA = True
ELSE
        boolA = False
ENDIF
// ---------------------------------------------------------------------
// Variation 2 much more compact.
// ---------------------------------------------------------------------
boolA = ((intA * 50) / 4) > intB
realA = 5.55
realB = 10.11
IF NOT(realA = realB) THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
realA = 5.55
realB = 10.11
```

```
IF NOT(realA < realB) THEN
        PRINT "Failure"
ELSE
        PRINT "Success"
ENDIF
realA = 5.55
realB = 10.11
IF NOT(realA > realB) THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
nOne      = 1
nTwo      = 2
nThree    = 3
nTotal    = 0
nTotal = (nOne + nTwo + nThree) - nThree
IF nTotal = 3 THEN
        PRINT "Success"
ELSE
        PRINT "Failure"
ENDIF
nTest     = 10
nModuloResult = 0
nModuloValue = 3
PRINT " ------------------  Modulo Operator ----------------------------"
// -------------------------------------------------------------------
// The result of the modulus operator (%) is the remainder
// when the first operand is divided by the second.
// -------------------------------------------------------------------
PRINT "nModuloResult: ", nModuloResult, " nTest: ", nTest, " nMoluloValue: ", nModuloValue
PRINT "nModuloResult = nTest % nModuloValue"
nModuloResult = nTest % nModuloValue
PRINT "nModuloResult = ", nModuloResult, " should be 1"
PRINT " ------------------  Bitwise AND Operator -----------------------"
// -------------------------------------------------------------------
// The bitwise-AND operator (&) compares each bit
// of its first operand to the corresponding bit of its second operand.
// If both bits are 1, the corresponding result bit is set to 1.
// Otherwise, the corresponding result bit is set to 0.
// -------------------------------------------------------------------
PRINT "nModuloResult: ", nModuloResult, " nTest: ", nTest, " nMoluloValue: ", nModuloValue
PRINT "nModuloResult = nTest & nModuloValue"
nModuloResult = nTest & nModuloValue
PRINT "nModuloResult = ", nModuloResult, " should be 2"
PRINT " --------------  Bitwise inclusive OR Operator ------------------"
// -------------------------------------------------------------------
// The bitwise-inclusive-OR operator (|) compares each bit
// of its first operand to the corresponding bit of its second operand.
// If either bit is 1, the corresponding result bit is set to 1.
// Otherwise, the corresponding result bit is set to 0.
// -------------------------------------------------------------------
PRINT "nModuloResult: ", nModuloResult, " nTest: ", nTest, " nMoluloValue: ", nModuloValue
PRINT "nModuloResult = nTest | nModuloValue"
nModuloResult = nTest | nModuloValue
PRINT "nModuloResult = ", nModuloResult, " should be 11"
PRINT " ----------------  One's compliment operator --------------------"
// -------------------------------------------------------------------
// The one’s complement operator (~),
// sometimes called the “bitwise complement” operator,
// yields a bitwise one’s complement of its operand.
// That is, every bit that is set in the operand is 0 in the result.
// Conversely, every bit that is 0 in the operand is set in the result.
// The operand to the one’s complement operator must be an integral type.
// -------------------------------------------------------------------
PRINT "nModuloResult: ", nModuloResult, " nTest: ", nTest
```

```
PRINT "nModuloResult = ~ nTest"
nModuloResult = ~ nTest
PRINT "nModuloResult = ", nModuloResult, " should be -11"
END
```

8.2 String Function Example

```
STRING strValue = "1234567890"
STRING strSubString = "67890"
INT nDest
PRINT " ----------------- ATOI Expression Test ---------------------------"
nDest = ATOI("12345") + 100000
PRINT "nDest = ", nDest
PRINT " ----------------- ATOI Test --------------------------------------"
ATOI(nDest,strValue)
PRINT "nDest = ", nDest
PRINT " ----------------- ITOA Expression Test ---------------------------"
strValue = ITOA(5,4) + " a conversion"
PRINT "strValue = ", strValue
PRINT " ----------------- ITOA Test --------------------------------------"
ITOA(strValue,5,4)
PRINT "strValue = ", strValue
PRINT " ----------------- MID Expression Test ----------------------------"
strValue = MID("1234567890", 5,4) + " a conversion"
PRINT "strValue = ", strValue
PRINT " ----------------- MID Test ---------------------------------------"
MID(strValue, "1234567890", 5,4)
PRINT "strValue = ", strValue
PRINT " ----------------- LEFT Expression Test ---------------------------"
strValue = LEFT("1234567890", 5) + " a conversion"
PRINT "strValue = ", strValue
PRINT " ----------------- LEFT Test --------------------------------------"
LEFT(strValue, "1234567890", 5)
PRINT "strValue = ", strValue
PRINT " ----------------- RIGHT Expression Test --------------------------"
strValue = RIGHT("1234567890", 5) + " a conversion"
PRINT "strValue = ", strValue
PRINT " ----------------- RIGHT Test -------------------------------------"
RIGHT(strValue, "1234567890", 5)
PRINT " ----------------- LEN Expression Test ----------------------------"
nDest = LEN(strValue) + 2
PRINT "nDest = ", nDest
PRINT " ----------------- FIND Expression Test ---------------------------"
nDest = FIND(strValue, "9", 0) + 2
PRINT "strValue = ", strValue
PRINT "nDest = ", nDest
PRINT " ----------------- SPANINCLUDING Expression Test ------------------"
strValue = SPANINCLUDING(strValue, "01234567")
PRINT "strValue = ", strValue
strValue = "0123456789"
PRINT " ----------------- SPANEXCLUDING Expression Test ------------------"
strValue = SPANEXCLUDING(strValue, "ABCDE5")
PRINT "strValue = ", strValue
PRINT " ----------------- LOWER Expression Test --------------------------"
strValue = LOWER("aAbBcCdDeEfFgGhHiIjJkKlL")
PRINT "strValue = ", strValue
PRINT " ----------------- UPPER Expression Test --------------------------"
strValue = UPPER("aAbBcCdDeEfFgGhHiIjJkKlL")
PRINT "strValue = ", strValue

END
```

8.3 Database Example

```
// ---------------------------------------------------------------------------
// Write a row with everything specified.
// ---------------------------------------------------------------------------
BOOLEAN         boolTI_Boolean
DATETIME        dateTI_DateTime
DATETIME        dateTI_LastUpdate
INT             intErrorCode, intRetRecCount, intTI_Integer, numericTI_Numeric
REAL            realTI_Real, floatTI_Float
STRING          strTI_Key, strTI_String, strErrorMsg
STRING          strDatabaseName, strTableName
STRING          strSelectString, strSelectKey, strSelectValue
boolTI_Boolean      = False
dateTI_DateTime = GETCURRENTDATETIME()
dateTI_LastUpdate = dateTI_DateTime
floatTI_Float       = 1234.4321
intTI_Integer       = 9011
numericTI_Numeric = 901100
realTI_Real         = 12.21
strDatabaseName = "DatabaseName"   // This permits you to change Database Name
strTableName        = "TableName"    // and Table Name in one spot.
strTI_String        = "All Columns Specified."
strSelectKey = "TI_Key"       // This permits changeing key and value in one spot
strSelectValue = "'000202'"    // also permits loops using varying select values.
strSelectString = strSelectKey + " = " + strSelectValue    // Here we string together
strTI_Key           = "000202"
PRINT "------- WRITETABLE -------"
WRITETABLE (strDatabaseName, strTableName, _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
        "", "", intErrorCode, strErrorMsg, intRetRecCount, _
        strTI_Key, intTI_Integer, numericTI_Numeric, strTI_String, _
                realTI_Real, floatTI_Float, _
                boolTI_Boolean, dateTI_DateTime, _
                dateTI_LastUpdate)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- READTABLE -------"
READTABLE (strDatabaseName, strTableName, _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
        strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount, _
        strTI_Key, intTI_Integer, numericTI_Numeric, strTI_String, _
                realTI_Real, floatTI_Float, _
                boolTI_Boolean, dateTI_DateTime, _
                dateTI_LastUpdate)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- EDITTABLE -------"
strTI_String = "Editted by EDITTABLE"
EDITTABLE (strDatabaseName, strTableName, _
        "TI_String", _
        strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount, _
        strTI_String)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- DELETETABLE -------"
DELETETABLE (strDatabaseName, strTableName, _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
        strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount)
PRINT "Error Code:    ", intErrorCode
```

```
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
// Now test the Queue
strTI_String            = "All Columns Specified."
strTI_Key               = "000203"
strSelectKey = "TI_Key"
strSelectValue = "'000203'"
strSelectString = strSelectKey + " = " + strSelectValue
PRINT "------- WRITEQUEUE -------"
WRITEQUEUE (strDatabaseName, strTableName, _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
        "", "", intErrorCode, strErrorMsg, intRetRecCount, _
        strTI_Key, intTI_Integer, numericTI_Numeric, strTI_String, _
                realTI_Real, floatTI_Float, _
                boolTI_Boolean, dateTI_DateTime, _
                dateTI_LastUpdate)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- READQUEUE -------"
READQUEUE (strDatabaseName, strTableName, _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
        strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount, _
        strTI_Key, intTI_Integer, numericTI_Numeric, strTI_String, _
                realTI_Real, floatTI_Float, _
                boolTI_Boolean, dateTI_DateTime, _
                dateTI_LastUpdate)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- EDITQUEUE -------"
strTI_String = "Editted by EDITQUEUE"
EDITQUEUE (strDatabaseName, strTableName, _
        "TI_String", _
        strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount, _
        strTI_String)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- DELETEQUEUE -------"
DELETEQUEUE (strDatabaseName, strTableName, _
        "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
        strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- WRITEQUEUE a bunch -------"
//
// WRITE QUEUE
//
STRING          strMonth[12]
INT             intMonthCount = 0
INT             intKey = 0
INT             intKeyMax = 10100
REAL            realBump = 0.0
REAL            realAdd = 70.007
REAL            floatMax = 99999.99999
strMonth[0] = "January"
strMonth[1] = "February"
strMonth[2] = "March"
strMonth[3] = "April"
strMonth[4] = "May"
strMonth[5] = "June"
strMonth[6] = "July"
strMonth[7] = "August"
strMonth[8] = "September"
```

```
strMonth[9] = "October"
strMonth[10] = "November"
strMonth[11] = "December"
FOR intKey = 110 TO intKeyMax STEP 10
        // ------------------------------------------------------------------
        // Write a row with everything specified.
        // ------------------------------------------------------------------
        ITOA (strTI_Key, intKey, 7)
        intTI_Integer = intKeyMax - ((intKey * 2) / 3)
        IF intMonthCount = 0 THEN
                strTI_String = strMonth[0]
                intMonthCount = intMonthCount + 1
                numericTI_Numeric = intMonthCount * Seconds()
        ELSEIF intMonthCount = 1 THEN
                strTI_String = strMonth[1]
                intMonthCount = intMonthCount + 1
                numericTI_Numeric = intMonthCount * Seconds()
        ELSEIF intMonthCount = 2 THEN
                strTI_String = strMonth[2]
                intMonthCount = intMonthCount + 1
                numericTI_Numeric = intMonthCount * Seconds()
        ELSEIF intMonthCount = 3 THEN
                strTI_String = strMonth[3]
                intMonthCount = intMonthCount + 1
                numericTI_Numeric = intMonthCount * Seconds()
        ELSEIF intMonthCount = 4 THEN
                strTI_String = strMonth[4]
                intMonthCount = intMonthCount + 1
                numericTI_Numeric = intMonthCount * Seconds()
        ELSEIF intMonthCount = 5 THEN
                strTI_String = strMonth[5]
                intMonthCount = intMonthCount + 1
                numericTI_Numeric = intMonthCount * Seconds()
        ELSEIF intMonthCount = 6 THEN
                strTI_String = strMonth[6]
                intMonthCount = intMonthCount + 1
                numericTI_Numeric = intMonthCount * Seconds()
        ELSEIF intMonthCount = 7 THEN
                strTI_String = strMonth[7]
                intMonthCount = intMonthCount + 1
                numericTI_Numeric = intMonthCount * Seconds()
        ELSEIF intMonthCount = 8 THEN
                strTI_String = strMonth[8]
                intMonthCount = intMonthCount + 1
                numericTI_Numeric = intMonthCount * Seconds()
        ELSEIF intMonthCount = 9 THEN
                strTI_String = strMonth[9]
                intMonthCount = intMonthCount + 1
                numericTI_Numeric = intMonthCount * Seconds()
        ELSEIF intMonthCount = 10 THEN
                strTI_String = strMonth[10]
                intMonthCount = intMonthCount + 1
                numericTI_Numeric = intMonthCount * Seconds()
        ELSEIF intMonthCount = 11 THEN
                strTI_String = strMonth[11]
                intMonthCount = 0
                numericTI_Numeric = intMonthCount * Seconds()
        ENDIF
        realBump = realBump + realAdd
        realTI_Real             = realBump
        floatTI_Float  = floatMax - realTI_Real
        IF boolTI_Boolean = True THEN
                boolTI_Boolean = False
        ELSE
                boolTI_Boolean = True
        ENDIF
```

```
		dateTI_DateTime = GETCURRENTDATETIME()
		dateTI_LastUpdate = GETCURRENTDATETIME()
		WRITEQUEUE ("DatabaseName", "TableName", _
		"TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
		"", "", intErrorCode, strErrorMsg, intRetRecCount, _
		strTI_Key, intTI_Integer, numericTI_Numeric, _
			strTI_String, realTI_Real, floatTI_Float, _
			boolTI_Boolean, dateTI_DateTime, dateTI_LastUpdate)
NEXT intKey
// ------- Read a bunch -------
//  Unload QUEUE
INT		intTI_IntegerA[1001], numericTI_NumericA[1001]
INT		intLoopCounter
STRING		strTI_KeyA[1001], strTI_StringA[1001]
REAL		realTI_RealA[1001], floatTI_FloatA[1001]
BOOLEAN		boolTI_BooleanA[1001]
DATETIME	dateTI_DateTimeA[1001]
READQUEUE ("DatabaseName", "TableName", _
		"TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime", _
		"TI_Key > ''", "", intErrorCode, strErrorMsg, intRetRecCount, _
		strTI_KeyA, intTI_IntegerA, numericTI_NumericA, _
			strTI_StringA, realTI_RealA, floatTI_FloatA, _
			boolTI_BooleanA, dateTI_DateTimeA)
PRINT "ERROR CODE  : ", intErrorCode
PRINT "Record Count: ", intRetRecCount
intRetRecCount = intRetRecCount - 1
FOR intLoopCounter = 0 TO intRetRecCount STEP 1
		PRINT "Record Begin: ", strTI_KeyA[intLoopCounter], _
			" ", intTI_IntegerA[intLoopCounter], _
			" ", numericTI_NumericA[intLoopCounter], _
			" ", strTI_StringA[intLoopCounter], _
			" ", realTI_RealA[intLoopCounter], _
			" ", floatTI_FloatA[intLoopCounter], _
			" ", boolTI_BooleanA[intLoopCounter], _
			" ", dateTI_DateTimeA[intLoopCounter], _
			" End."
NEXT intLoopCounter
// Now lets run the TABLE AND QUEUE functions against a table without keys.
strDatabaseName = "DatabaseName"   // This permits you to change Database Name
strTableName		= "TableNameNoKey"    // and Table Name in one spot.
strTI_String		= "All Columns Specified."
strSelectKey = "TI_Key"       // This permits changeing key and value in one spot
strSelectValue = "'000204'"    // also permits loops using varying select values.
strSelectString = strSelectKey + " = " + strSelectValue    // Here we string together
strTI_Key		= "000204"
PRINT "------- WRITETABLE -------"
WRITETABLE (strDatabaseName, strTableName, _
		"TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
		"", "", intErrorCode, strErrorMsg, intRetRecCount, _
		strTI_Key, intTI_Integer, numericTI_Numeric, strTI_String, _
			realTI_Real, floatTI_Float, _
			boolTI_Boolean, dateTI_DateTime, _
			dateTI_LastUpdate)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- READTABLE -------"
READTABLE (strDatabaseName, strTableName, _
		"TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
		strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount, _
		strTI_Key, intTI_Integer, numericTI_Numeric, strTI_String, _
			realTI_Real, floatTI_Float, _
			boolTI_Boolean, dateTI_DateTime, _
			dateTI_LastUpdate)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
```

```
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- EDITTABLE -------"
strTI_String = "Editted by EDITTABLE"
EDITTABLE (strDatabaseName, strTableName, _
            "TI_String", _
            strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount, _
            strTI_String)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- DELETETABLE -------"
DELETETABLE (strDatabaseName, strTableName, _
            "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
            strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
// Now test the Queue
strTI_String            = "All Columns Specified."
strTI_Key               = "000205"
strSelectKey = "TI_Key"
strSelectValue = "'000205'"
strSelectString = strSelectKey + " = " + strSelectValue
PRINT "------- WRITEQUEUE -------"
WRITEQUEUE (strDatabaseName, strTableName, _
            "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
            "", "", intErrorCode, strErrorMsg, intRetRecCount, _
            strTI_Key, intTI_Integer, numericTI_Numeric, strTI_String, _
                        realTI_Real, floatTI_Float, _
                        boolTI_Boolean, dateTI_DateTime, _
                        dateTI_LastUpdate)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- READQUEUE -------"
READQUEUE (strDatabaseName, strTableName, _
            "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
            strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount, _
            strTI_Key, intTI_Integer, numericTI_Numeric, strTI_String, _
                        realTI_Real, floatTI_Float, _
                        boolTI_Boolean, dateTI_DateTime, _
                        dateTI_LastUpdate)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- EDITQUEUE -------"
strTI_String = "Editted by EDITQUEUE"
EDITQUEUE (strDatabaseName, strTableName, _
            "TI_String", _
            strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount, _
            strTI_String)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
PRINT "------- DELETEQUEUE -------"
DELETEQUEUE (strDatabaseName, strTableName, _
            "TI_Key, TI_Integer, TI_Numeric, TI_String, TI_Real, TI_Float, TI_Boolean, TI_DateTime, TI_LastUpdate", _
            strSelectString, "", intErrorCode, strErrorMsg, intRetRecCount)
PRINT "Error Code:    ", intErrorCode
PRINT "Error Message: ", strErrorMsg
PRINT "Ret Rec Count: ", intRetRecCount
END
```