Crestron SIMPL+®
Software

Language Reference Guide

CRESTRON

This document was prepared and written by the Technical Documentation department at:

Crestron Electronics, Inc.
15 Volvo Drive
Rockleigh, NJ 07647
1-888-CRESTRON

Contents

SIMPL+ Language Reference Guide

This page intentionally left blank.

Introduction

SIMPL+® is a language extension that enhances SIMPL Windows by using a procedural “C-like” language to code elements of the program that were difficult, or impossible, with SIMPL alone. This help system provides specific information about the SIMPL+ language syntax, and can be used as a reference manual.

For a tutorial on SIMPL+ programming, consult the SIMPL+ Programming Guide (Doc. 5789). The latest version of the guide can be obtained from the Downloads | Product Manuals section of the Crestron website (www.crestron.com).

Software Requirements

SIMPL+ has several versions. Earlier versions of SIMPL+ do not contain features and constructs found in later revisions. Each version of SIMPL+ requires a minimum revisions of SIMPL Windows and Control System Update (UPZ or, for 2-Series control systems, CUZ) files. The specifications are listed below.

Software Requirements

SIMPL+ VERSION	MINIMUM SIMPL WINDOWS REQUIRED	MINIMUM UPZ	MINIMUM CUZ
Version 1.00	1.30.01	5.04.11	N/A
Version 2.00	1.40.02	5.10.00	N/A
Version 3.00	2.00	N/A	1.00

Licensing of SIMPL+ Cross Compiler

Crestron SIMPL+ Cross-Compiler Version 1.1 is simply an Installshield-installed version of the Coldfire GNU C Compiler, which is available on Crestron's FTP site in the SIMPL Windows directory as directory GNUSOURCE in ftp://ftp.crestron.com/Simpl_Windows and in the \GNUSource directory of the Programming Tools CD.

It includes and references code that is available from www.cygwin.com/cvs.html

Some files are deleted by the Installshield procedure which are not necessary for general use of the C compiler, in order to save space on user PCs. But it is an unmodified version of this code. The original executables and the source code for them can be obtained from the authors at the above sites

The source code has also been gathered underneath a single directory for your convenience and is available on Crestron's FTP site in the SIMPL Windows directory as directory GNUSOURCE in ftp://ftp.crestron.com/Simpl_Windows and in the \GNUSource directory of the Programming Tools CD. They also include GNU utilities, which are copyrighted by the Free Software Foundation.

Other Crestron software simply executes this code as a separate executable, and does not incorporate GNU source code into Crestron software. Crestron's standard licensing agreement does not apply to this software; only the license described here applies.

The text for the license agreement below is also available from www.gnu.org/copyleft/gpl.html

What's New

Converting from an X-Generation to a 2-Series Target

- Select 2-Series Target within SIMPL+ environment (From the SIMPL+ application menu, select **Build | 2-Series Control System Target**. The X-Generation Target may be deselected if no longer needed).
- Recompile the SIMPL+ program.

X-Generation Target and 2-Series Target Differences

- I/O Datatypes (DIGITAL_INPUT, etc.) can no longer be passed to functions as arguments.
- Global variables can no longer be declared within User or Crestron Libraries.
- If TerminateEvent resides within a Wait Statement Block, it will only exit the Wait Statement Block's function scope - NOT the PUSH, CHANGE, RELEASE or EVENT in which it resides.

The following functions are no longer available in the 2-Series Control System:

GetCIP()

SetCIP()

GetCresnet()

SetCresnet()

GetSlot()

SetSlot()

_OEM functions

#ANALOG_INPUT_JOIN

#ANALOG_OUTPUT_JOIN

#DIGITAL_INPUT_JOIN

#DIGITAL_OUTPUT_JOIN

#STRING_INPUT_JOIN

#STRING_OUTPUT_JOIN

Programming Environment

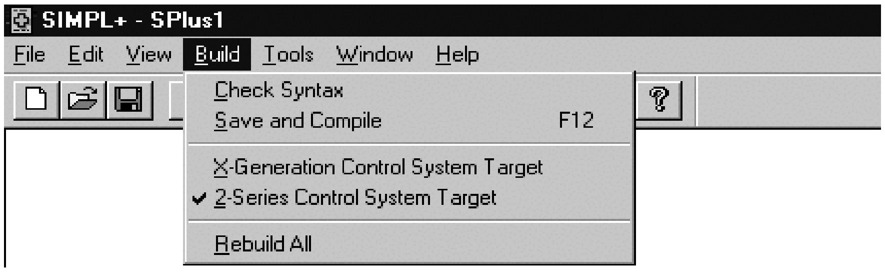
Programming Environment Overview

While running SIMPL Windows, select **File | New SIMPL+** and the SIMPL+ programming environment appears. This section describes the environment for SIMPL+ Version 3.00.

The SIMPL+ Module Information template is filled with commented code that makes it easy to remember the language syntax and structure. Simply locate the necessary lines, uncomment them, and add the appropriate code. To uncomment a line of code, either remove the "//" that appears at the start of the line or remove the multi-line comment indicators /*…*/.

Target Selection

Target Selection Pulldown Menu


SIMPL+ - SPlus1
File Edit View Build Tools Window Help
Check Syntax
Save and Compile F12
X-Generation Control System Target
✓ 2-Series Control System Target
Rebuild All

X Generation (CNX) Control Systems consist of the CEN-TVAV, CNMSX-AV/PRO, and CNRACKX/-DP.

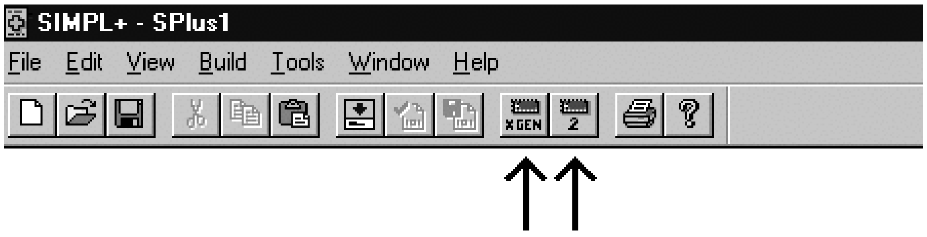
The 2-Series Control Systems currently consist of the AV2, CP2, CP2E, PAC2, PAC2M, PRO2, and RACK2.

Selecting a target implies that the module MUST work for that target and any statements that are not valid for that target are NOT permitted. It does NOT mean that the module won't work for other targets - it may, if it were compiled for other targets at some future time. More functions and support are available for 2-Series systems, so do not limit yourself to the X-Generation usages, if they are not needed.

NOTE: In previous versions of SIMPL+, the settings for the target types were system-wide. Those settings applied to all SIMPL+ modules that were opened and not specific to the active module being edited. In version 3.00, the target type setting is specific only to the active module being edited and saved within that module. The toolbar buttons reflect the target type of the active module within the SIMPL+ environment.

One or both targets may be selected to compile the program for both types of control systems. When compiling a program for a specific type of control system, an error message appears if a wrong control system target is selected that does not support a particular function or syntax.. Shown below are the two target selection buttons of the menu toolbar.

Toolbar Target Selection Buttons.

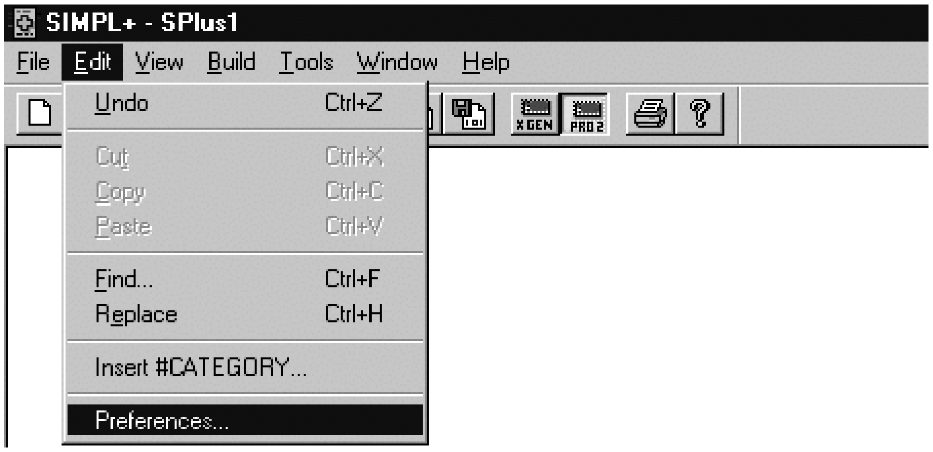


X-GEN - shortcut to **Build | X-Generation Control Systems Target 2** - shortcut to **Build | 2-Series Control Systems Target**. (This is the default setting upon opening SIMPL+.)

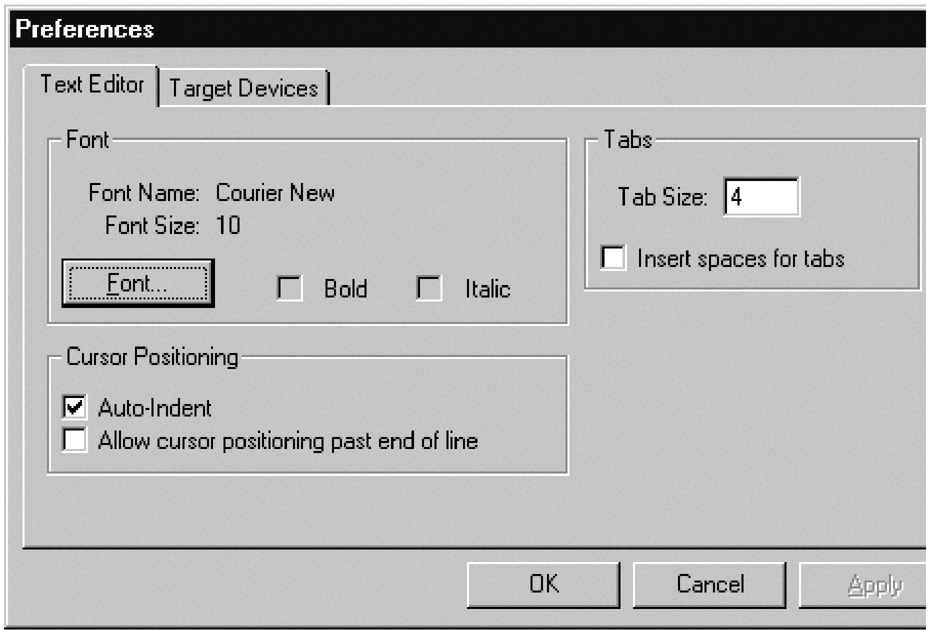
NOTE: If a program is compiled for the wrong type of control system, an error message appears when attempting to upload, and the program must be recompiled.

Edit Preferences

Preferences Toolbar Pull-Down Menu



Text Editor Tab


Preferences

Text Editor | Target Devices

Font

Font Name: Courier New

Font Size: 10

☐ Bold ☐ Italic

Tabs

Tab Size: 4

☐ Insert spaces for tabs

Cursor Positioning

☑ Auto-Indent

☐ Allow cursor positioning past end of line

Font - Used to select font to be used in SIMPL+ Text Editor's main window.

Cursor Positioning, Auto-Indent - When the 'enter' key is pressed, the cursor will automatically indent to the same initial tab position as in the current line.

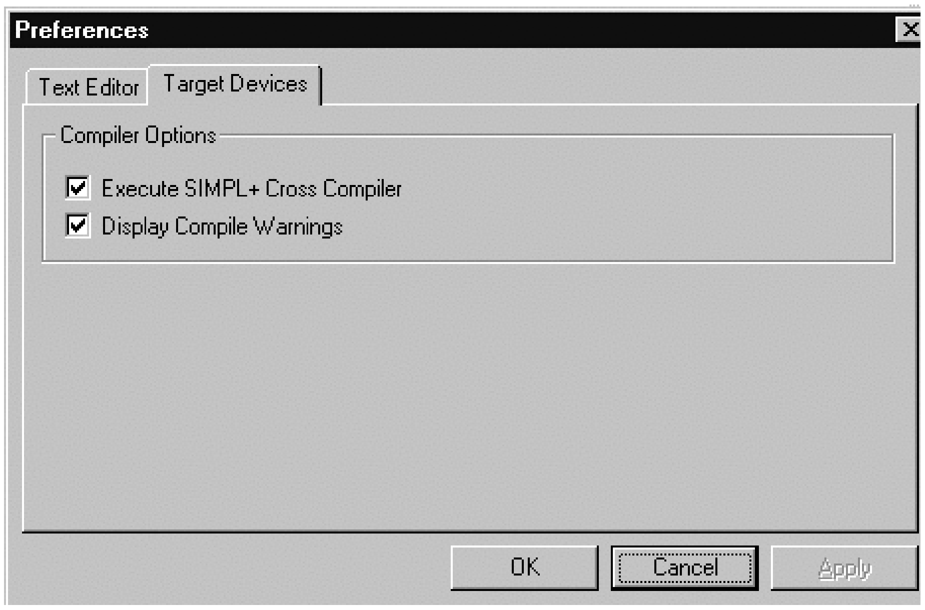
- To manually indent a block of text, highlight the block and press **TAB**.
- To manually outdent a block of text, highlight the block and press **SHIFT** and **TAB**.
- If you have manually inserted spaces for tabs, then pressing **SHIFT TAB** will only outdent by only one space.

Cursor Positioning, Allow cursor positioning past end of line - If checked, the cursor will be allowed to be placed anywhere within the text editor. This includes any white-space area. Disabling this option will force the cursor to the end of the current line selected when the cursor is clicked on any white-space past the end of the line.

Tab Size - The number of spaces that equal 1 tab character.

Insert Spaces for tabs - Spaces will be inserted in place of the tab character.

Target Devices Tab



Compiler Options

[✓] Execute SIMPL+ Cross Compiler

[✓] Display Compile Warnings

Execute SIMPL+ Cross Compiler - After target files are compiled, the cross compiler can be launched from the SIMPL+ environment. This will enable you to generate the target file that will be uploaded to the operating system. Normally, the SIMPL Windows environment will handle this, since it is responsible for uploading the target file to the operating system.

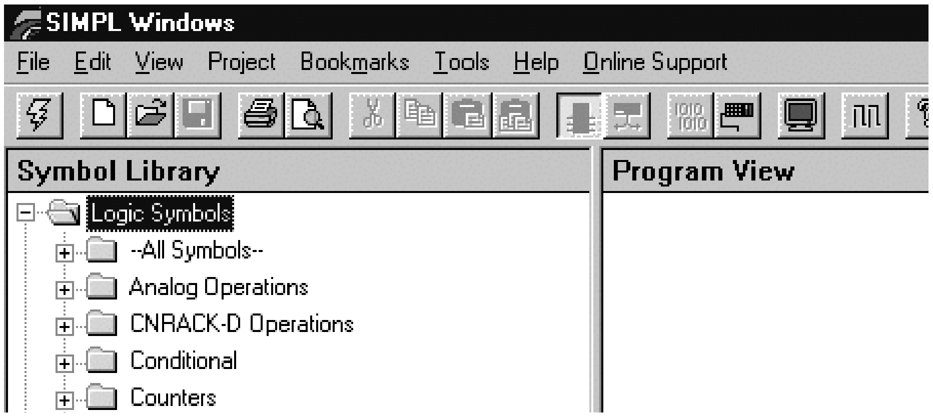
Display Compile Warnings - When selected, the compiler displays all program warnings during compile in the compile output window. The total number of warnings will always be displayed whether this option is selected or not.

Insert Category

Displays a list of all available categories for the symbol tree in the SIMPL Windows environment. This list is for reference only.

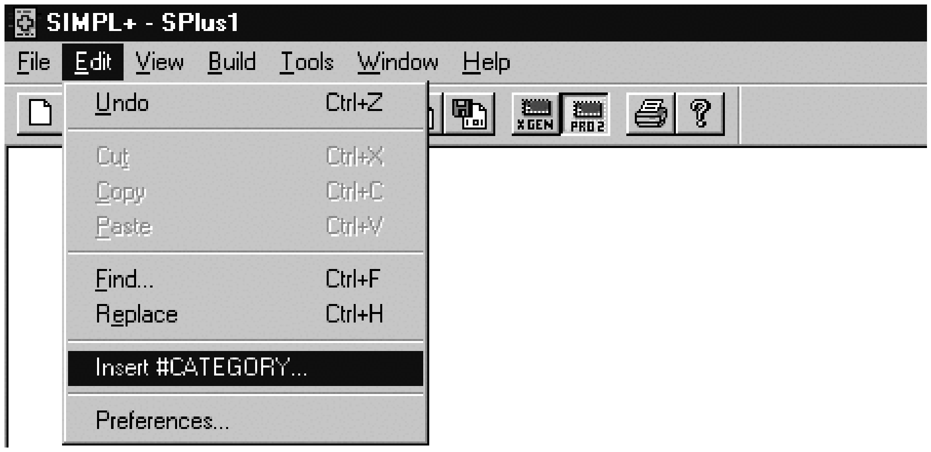
To specify a category for a SIMPL+ module, the #CATEGORY directive must be used with a category specified in this list. If a category name is typed in that does not exist in the Symbol Tree Category list, the SIMPL+ module will default to the category type, Miscellaneous.

Symbol Tree Category List in SIMPL Windows

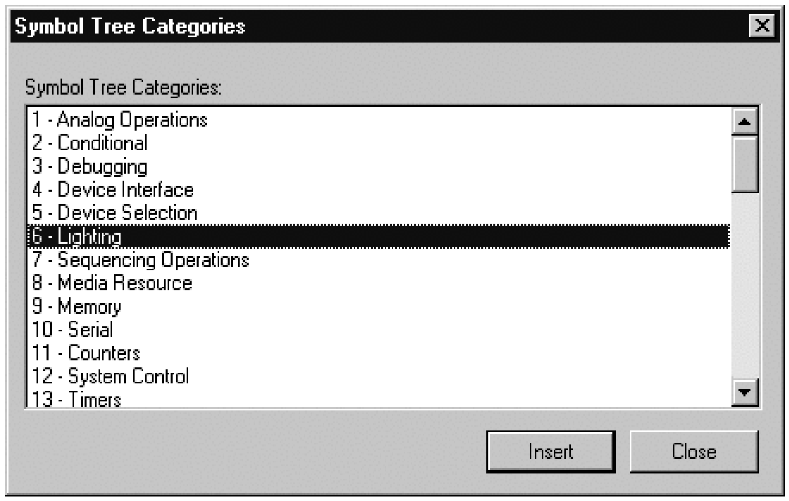


- Logic Symbols
  - --All Symbols--
  - Analog Operations
  - CNRACK-D Operations
  - Conditional
  - Counters

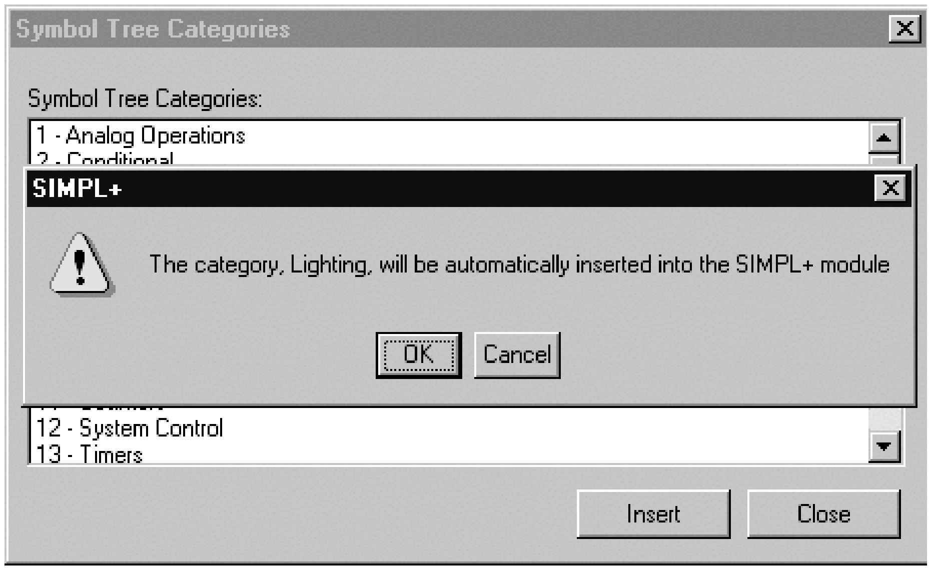
Insert #CATEGORY Toolbar Pull-Down Menu in SIMPL+



Symbol Tree Category Pop-Up Window



Category Selection Insertion Box



The category, Lighting, will be automatically inserted into the SIMPL+ module

General Information

Conventions Used

Variable names are placed in <> when discussing syntax. For example, PUSH <variable>.

Optional parameters are placed in [ ]. For example, when a list has many parameters, it would be described as <var1>[, <var2>...] When discussing array notation, [ ] is used for array subscripting and is not used to mark optional code.

Examples are placed in a Computer Style font, i.e.,

```
MyVariable = ATOI(SomeOtherVariable);
```

Variable Names

Variable names in SIMPL+ may be up to 30 characters long and may not contain any of the operators specified in the “Operators” section. Valid characters in a variable name are a-z, A-Z, 0-9, #, _, and $ but may not begin with 0-9.

Variable names may not duplicate existing function or keyword names.

Variable names in SIMPL+ are not case sensitive. For example, declaring a variable “joe” can be used as “jOe” or “JOE” or any variation of case.

NOTE: Version 3.00.12 users: variable names may be 120 characters for 2-Series systems.

Comments

It is beneficial to comment code to make it more readable and for documentation. Comments do not exist in any form after code generation and are not required.

SIMPL+ has two styles of comments, single line and block comments. Single line comments start with the characters //. The rest of the line (until a carriage return) is considered a comment. If they occur within a quoted string, such as in PRINT, they are NOT treated as comment characters, but rather as two backslash (Hex 2F) characters.

Examples:

```
PRINT(“Hello, World!\n”);  // This stuff is a comment.
PRINT(“hello, // world!\n”); // This stuff is a comment,
// but the string actually
// printed is hello,
// world.
```

The second form of comment characters are the block comments. /* starts a block comment and */ ends a block comment. This is useful for commenting out large sections of code or writing large sections of documentation. Note that nested comments are not supported. Also, if /* or */ appear inside of a quoted string such as in an PRINT statement, they are not considered comments but part of the string.

Examples:

```
/*
This
is
all
a comment!
*/
PUSH Trig
{
// code that does something.
}
```

Relative Path Names for Files

Your current working directory is reset to the default ("\" or root) whenever "StartFileOperations" is performed. It is changed only by "SetCurrentDirectory".

File names can consist of full path names or relative path names.

- Full path names have the same restrictions as DOS file names in characters and format, with a maximum length of 256 characters.
- Relative path names do not begin with a "\" and start from the current working directory.

Operators

Operators Overview

SIMPL+ operators perform functions between two or more variables. SIMPL+ operators consist of Arithmetic, Bitwise, and Rational Operators.

Arithmetic Operators

OPERATOR	NAME	EXAMPLE	EXPLANATION
-	Negation	-X	Negate the value of X (2's Complement of X).
*	Multiplication	X *Y	Multiply X by Y (signed arithmetic).
/	Unsigned Division	X / Y	Divide X by Y, truncates result (unsigned arithmetic).
S/	Signed Division	X S/ Y	Divide X by Y, truncates result (signed arithmetic).
MOD	Signed Modulo	X MOD Y	Remainder after dividing X by Y (signed arithmetic).
UMOD	Unsigned Modulo	X UMOD Y	Remainder after dividing X by Y (unsigned arithmetic). Only 2-Series Systems.
+	Addition	X + Y	Add the value of Y to X.
-	Subtraction	X - Y	Subtract the value of Y from X.

Bitwise Operators

OPERATOR	NAME	EXAMPLE	EXPLANATION
<<	Shift Left	X << Y	Shift X to the left by Y bits; 0 is Shifted in.
>>	Shift Right	X >> Y	Shift X to the right by Y bits; 0 is Shifted in.
{{	Rotate Left	X {{ Y	Rotate X to the left by Y bits; full 16 bits used. Same as RotateLeft().
}}	Rotate Right	X }} Y	Rotate X to the right by Y bits; full 16 bits used. Same as RotateRight().
NOT	1's Complement	NOT(X)	Change 0 bits to 1, 1 bits to 0.
&	Bitwise AND	X & Y	AND the bits of X with the bits of Y.
|	Bitwise OR	X | Y	OR the bits of X with the bits of Y.
^	Bitwise XOR	X ^ Y	XOR the bits of X with the bits of Y.

NOTE: For the Shift and Rotate operators, only the lower 5-bits of Y are used, giving values of Y ranging from 0 to 31. For example, if Y=600, the lower 5-bits equate to 24. Rotating a 16-bit number through 16 positions gives the original number back. Therefore, for rotating 24, the result is equivalent to rotating through 8. Shifting greater than 16 will always give a 0 as a result.

Relational Operators

OPERATOR	NAME	EXAMPLE	EXPLANATION
=	Comparison	X = Y	True if X is equal to Y, False otherwise.
=	Assignment	X = Y	Assigns the contents in Y to X. The assignment operator cannot be used within expressions.
!	Complement	! X	If X = 0, X changes to 1. If X is different from 0, evaluates to 0.
<>	Not Equal To	X <> Y	X is not equal to Y.
<	Unsigned Less Than	X < Y	X is less than Y (unsigned).
>	Unsigned Greater	X > Y	X is greater than Y (unsigned).
<=	Unsigned Less Than or Equal	X <= Y	X is less or equal to Y (unsigned).
>=	Unsigned Greater Than or Equal	X >= Y	X is greater or equal to Y (unsigned).
S<	Signed Less Than	X S< Y	X is less than Y (signed).
S>	Signed Greater Than	X S> Y	X is greater than Y (signed).
S<=	Signed Less Than or Equal	X S<= Y	X is less or equal to Y (signed).
S>=	Signed Greater Than or Equal	X S>= Y	X is greater or equal to Y (signed).
&&	Logical AND	X && Y	True if X and Y are both non-zero. False otherwise.
||	Logical OR	X || Y	True if either X or Y is non-zero. False otherwise.

All of the above operators, with the exception of the negation (-), NOT, and complement (!) operators, are called binary operators. Binary operators take two values, perform an operation, and return a third value as a result. For example, 5 + 6 would return the value of 11. The arguments for a given operator are called its operands. In the above example, the + sign is the operator and 5 and 6 are the operands.

The negation, NOT, and complement operators are called unary operators, which means it takes a single number and performs an operation. In this case, the negation operator performs a negate, or 2's complement. A 2's complement takes a 16-bit number, bitwise inverts it, and adds 1. The operand in a negation is the value being negated. Operands do not have to be simple numbers. They may also be variables or the results of a function call. For example, in the expression -X, the - sign is the operator and the variable X is the operand.

String Operators

OPERATOR	NAME	EXAMPLE	EXPLANATION
=	Assignment*	A$ = B$	Assigns the value in B$ to A$.
***NOTE:** Not allowed in expressions because of possible confusion with comparison.			
=	Comparison	A$ = B$	A$ equal B$
<>	Not Equal To	A$ <> B$	A$ is not equal to B$
<	Less Than	A$ < B$	A$ is less than B$
>	Greater Than	A$ > B$	A$ is greater than B$

For less than and greater than operations, the string is evaluated in ASCII order. For example, the comparison “ABC” > “ABD” would be false. The system looks character by character; the first two characters are identical in both strings, and when it evaluated the characters C (ASCII 67) vs. D (ASCII 68), the result is false. The comparison `“ABC”<“ABCD”` is true because a shorter string alphabetically precedes one that is identical but longer.

Signed vs Unsigned Arithmetic

ANALOG_INPUT, ANALOG_OUTPUTs, and INTEGER in SIMPL+ are 16-bit quantities. A 16-bit quantity can range from 0 - 65535 when it is treated without having a sign (positive or negative). If a 16-bit number is treated as signed in SIMPL+, the range becomes -32768 to 32767. The range from -32768 to -1 maps into 32768 to 65535. Expressed mathematically, the mapping is 65536 - AbsoluteValue(Number). The values are treated differently depending on whether signed or unsigned comparisons are used. Another way is as follows.

Signed	0 - 32767	32768 -	65535
Unsigned	0 - 32767	-32768 -	-1

Assignments may be directly done with negative constants, for example:

```
INTEGER I, J;
I = -1;
J = 65535;
```

Results in I being equivalent to J.

Example:

```
IF (65535 S> 0)
    X=0;
ELSE
    X=1;
```

Above, the value of X is set to 1 since in signed arithmetic, 65535 is the same as -1, which is not greater than 0.

```
IF (65535 > 0)
    X=0;
ELSE
    X=1;
```

Above, the value of X is set to 0 since in unsigned arithmetic, 65535 is greater than 0.

Datatype Conversions

SOURCE	DESTINATION	ACTION
INTEGER	LONG_INTEGER	Lower 2 bytes of destination = source. Upper 2 bytes cleared.
INTEGER	SIGNED_INTEGER	The 2 bytes of source moved to destination. 2 byte number now treated as signed.
INTEGER	SIGNED_LONG_INTEGER	Lower 2 bytes of destination = source. Upper 2 bytes cleared.
LONG_INTEGER	INTEGER	Lower 2 bytes of source moved to destination, treated as unsigned.
LONG_INTEGER	SIGNED_INTEGER	Lower 2 bytes of source moved to destination, treated as signed.
LONG_INTEGER	SIGNED_LONG_INTEGER	The 4 bytes of destination = source, now treated as signed.
SIGNED_LONG_INTEGER	INTEGER	Lower 2 bytes of source moved to destination.
SIGNED_LONG_INTEGER	SIGNED_INTEGER	Lower 2 bytes of source moved to destination.
SIGNED_LONG_INTEGER	LONG_INTEGER	The 4 bytes of destination = source, now treated as unsigned.
SIGNED_INTEGER	INTEGER	Lower 2 bytes of source moved to destination, 2 byte number now treated as unsigned.
SIGNED_INTEGER	LONG_INTEGER	2 byte source is sign extended to 4 bytes
SIGNED_INTEGER	SIGNED_LONG_INTEGER	2 byte source is sign extended to 4 bytes

Operator Precedence & Grouping

In an expression where many operators are present, some operators have “priority” over others. Operators with the same precedence level are evaluated strictly left to right. Grouping is used to change the way an expression is evaluated.

Operator Precedence & Grouping

PRECEDENCE LEVEL	OPERATORS
1	- (Negate)
2	! NOT
3	* / S/ MOD
4	+ -
5	{{ }}
6	<< >>
7	> < >= <= S> S> S>= S<=
8	= <>
9	&
10	^
11	|
12	&&
13	||

As an example, the expression:

```
3+5*6
```

Evaluates to 33 since the multiplication is performed first. It may be beneficial to use grouping to show which operations are performed first. Grouping is simply starting an expression with '(' and ending with ')'. Therefore, the expression 3+5*6 is equivalent to 3+(5*6). Grouping is very important if you want to override the default behavior and have one piece of the expression evaluated first. Therefore, to make sure the + is evaluated first, the expression is written as (3+5)*6, for a result of 48.

Numeric Formats

Numeric (Integer) constants may be expressed in three formats; decimal, hexadecimal, or quoted character.

Decimal constants are specified by writing a decimal number. Hexadecimal constants are specified by prefacing the hex constant by 0x. Quoted character constants are a single character placed between single quotes (') and have the numeric value specified on an ASCII chart.

Example:

```
INTEGER I;
I=123;   // Specify Decimal constant.
I=0xABC; // Specify a Hexadecimal constant (Decimal value
2748)
I='A';   // Specify a character constant (Decimal value 65)
INTEGER K;
K=54;    // Specify Decimal constant
K=0x36;  // Specify a Hexadecimal Constant (Decimal
             // Value 54)
K='6';   // All three of these are the same value
             // (Decimal value 54)
```

The three forms may be used interchangeably and are used to make code more readable.

Example:

```
STRING A$[10], B$[10], C$[10];
INTEGER I;
BUFFER_INPUT COM_IN$[50];
// A$, B$, and C$ contain identical values
// after these lines run.
A$=CHR('A');
B$=CHR(65);
C$=CHR(0x41);
// Preserve the lower nibble of a word, mask the rest out.
I = VAL1 & 0x000F;
// Read until a comma is detected in the stream.
DO
{
I = GetC(COM_IN$)
}
UNTIL (I = ',');
```

Task Switching

Task Switching for X-Generation (CNX) Control Systems

Each SIMPL+ module runs as a separate task in the X-Generation (CEN-TVAV, CNMSX-AV/PRO, CNRACKX/-DP) Control System. In order to insure that no SIMPL+ program takes up too much time, each task is allotted a certain amount of time to run. If the task exceeds this time limit, the system will switch out and allow other tasks (including the SIMPL program) to run.

The system will not arbitrarily switch out at any point in time. Even if the task limit is exceeded, the system will force a task switch only at predetermined points.

The system will perform a task switch when a PROCESSLOGIC, DELAY, or PULSE function is encountered. When a task switch is performed, the output I/O definitions are updated (refer to ANALOG_OUTPUT, DIGITAL_OUTPUT, STRING_OUTPUT for further information). Note that a WAIT does not cause a task switch.

When a WHILE, DO-UNTIL, or FOR construct encounters its last statement, or any construct that causes a “backwards branch”, the system checks to see if a timeout has occurred. If the timeout has occurred, then the system will task switch away. When the module is given time to run, it will resume at the top of the construct.

For this reason, a designer of a SIMPL+ module should take care to design with this in mind. A particular concern is if the outputs need to be updated in a specific fashion and have a loop, which may potentially cause the system to switch away. One solution would be to store the output variables in intermediate arrays or variables, and assign the intermediate variables to the output variables before the event terminates.

Example:

```
DIGITAL_INPUT trig;
ANALOG_OUTPUT i;
INTEGER j;
PUSH trig
{
  j=0;
  FOR(j=0 to 32000)
  {
    i = j;
  }
}
```

A SIMPL program drives the trig signal and monitors the state of the analog_output with an ANALOG DEBUGGER (Speedkey: TEST2) symbol. If the system did not task switch out, the only TEST2 output would show 32000. If this program were run, there would be many outputs, indicating each time the FOR loop exceeded the allotted time, the SIMPL program would be given time to run and the TEST2 symbol would post the results.

If it were critical that the analog_output were only updated with the final value, the following alternative solution could be used:

```
DIGITAL_INPUT trig;
ANALOG_OUTPUT i;
INTEGER j, q;
PUSH trig
{
  j=0;
  FOR(j=0 to 32000)
  {
    q = j;
  }
  i = q;
}
```

This program output would only show the final result; the TEST2 would be triggered once with the value 32000. The system will still perform whatever task switching it requires.

When an event has task switched away, it is possible that the event may be retriggered and a new copy of the event will start running. Therefore, SIMPL+ events are considered to be re-entrant. The event may be reentered only a limited number of times before an Rstack overflow error occurs (refer to “Common Runtime Errors” that begins on page 302). In order to prevent the event from running multiple times, consider the following example:

```
DIGITAL_INPUT trig;
INTEGER I;
PUSH trig
{
  FOR(I = 0 TO 32000)
  {
    // code
  }
}
```

This code will task switch away at some point in the FOR loop. If trig is hit again while the event is task switched out, a new copy will run. This code can be changed to prevent multiple copies from running.

```
DIGITAL_INPUT trig;
INTEGER I, Running;
PUSH trig
{
  IF(!Running)
```

```
    {
      Running = 1;
      FOR(I = 0 TO 32000)
      {
         // code
      }
      Running = 0;
    }
  }
  FUNCTION MAIN()
  {
     Running = 0;
  }
```

In this case, a new variable, Running is declared and set to 0 on system startup in the MAIN. When the event is triggered, if Running is 0, then it will be set to 1, and the FOR loop will execute. Assume now the event has a task switch. If trig is hit again, the event will start, but will immediately exit because IF statement evaluates to false. When the task resumes, and ultimately completes, Running will be set to 0 again so the bulk of the function may execute again.

NOTE: The event is STILL reentering. It is being forced to terminate immediately and prevent reentry more than one level deep.

Task Switching for 2-Series Control Systems

In the 2-Series Control Systems, each SIMPL+ module also runs as one or more concurrent tasks in the control system. The MAIN and each event handler run as separate tasks sharing a common global data space.

To insure that no SIMPL+ program takes too much time, each task is allotted a certain amount of time to run. If the task exceeds this time limit, the system will switch out and allow other tasks (including the SIMPL program) to run. It should also be noted that the task would run until it has completed the operation, the allotted time expires or a task switching call is executed.

Unlike the X-Generation systems, the system will arbitrarily switch out at any point in time. If this may result in undesirable behavior, then the programmer should control his task switching by issuing a PROCESSLOGIC function.

The system will perform a task switch when a PROCESSLOGIC or DELAY function is encountered. The PULSE will no longer cause a task switch because it is no longer needed for the logic processor to process the digital output pulse. Note that a WAIT does not cause a task switch but will execute in its own task.

All outputs are processed by the logic processor as soon as assigned. As soon as the SIMPL+ module releases the processor, all the outputs are seen by the logic processor. Also, the programmer can read back DIGITAL_OUTPUTS and ANALOG_OUTPUTS without having to insert a PROCESSLOGIC in between.

To use the example from the Task Switching for X-Generation Control System discussion:

```
DIGITAL_INPUT trig;
ANALOG_OUTPUT i;
ANALOG_OUTPUT NewNumber;
INTEGER j;
PUSH trig
{
  j=0;
  NewNumber = 1234;
  j = NewNumber; //j = 1234, not old value of NewNumber
  FOR(j=0 to 32000)
  {
    i = j;
  }
}
```

A SIMPL program drives the trig signal and monitors the state of ANALOG_OUTPUT with an ANALOG DEBUGGER (Speedkey: TEST2) symbol. The TEST2 output would show all numbers from 0 to 32000. If it were critical that the ANALOG_OUTPUT were only updated with the final value, the following alternative solution could be used:

```
DIGITAL_INPUT trig;
ANALOG_OUTPUT i;
INTEGER j, q;
```

```
PUSH trig
{
  j=0;
  FOR(j=0 to 32000)
  {
    q = j;
  }
  i = q;
}
```

This program output would only show the final result; the TEST2 would be triggered once with the value 32000. The system will still perform whatever task switching required. As with the X-Generation series, re-entrance can still be a problem. When an event has task switched away, the event may be retriggered and a new copy of the event will start running. Therefore, SIMPL+ events are considered to be re-entrant. The amount of times that this could occur is dependent upon the available memory in the system. In order to prevent the event from running multiple times, refer to the re-entrant example in the X-Generation task switching section.

The programmer should exercise caution when using looping constructs without constraints (i.e. while(1)) or depend upon outside influence. Because each event will run for the allotted time unless specified otherwise, PROCESSLOGIC calls should be used to reduce the CPU overhead. Consider the following:

```
DIGITAL_INPUT diInput1, diInput2;
INTEGER I, LastNumSeconds;
PUSH diInput1
{
  WHILE (diInput1)
  {
     // do something
  }
}
main()
{
LastNumSeconds = 0;
   WHILE (1)
   {
     seconds = GetNumSeconds();
     IF (seconds <> LastNumSeconds)
     {
        // do something
     }
   }
}
```

At the loop in MAIN, the programmer wants to perform an operation every second. This code will achieve that goal. However, a side effect of the code is that every time the task is scheduled to run, it will sit in a very tight loop checking for a change in the number of seconds. Since the allotted time for a SIMPL+ task to run is in fractions of a second, it is very unlikely to change during the allotted time. Unless the programmer puts in a DELAY which will put the task to “sleep” for a period of time, this task will dominate the CPU time.

The programmer who writes the MAIN() function should also be aware that the MAIN() function begins running when the SIMPL Windows program is initializing. The module’s inputs do not have their programmed state until sometime after the first break in the program execution due either to a process logic statement or expiration of a time slice.

The PUSH event indicates a more subtle problem. The programmer wants to loop in the event until the input diInput1 is released. Once the task containing the event is started, it will run for its allotted time and no other inputs will change. If the signal attached to the diInput1 signal goes low, the event will not see to the change until the event switches out and the diInput1 low signal is processed.

The following is an alternative:

```
DIGITAL_INPUT diInput1, diInput2;
INTEGER I, LastNumSeconds;
PUSH diInput1
{
  WHILE (diInput1)
  {
     // do something

     ProcessLogic();
  }
}
MAIN()
{
LastNumSeconds = 0;
   WHILE (1)
   {
     seconds = GetNumSeconds();
     IF (seconds <> LastNumSeconds)
     {
        // do something
     }
     delay(10);
   }
}
```

Here, a 100ms delay is put in the MAIN loop. That means that the task will only wake up 10-times per second. It will still catch the change of the seconds to within a 1/10 of a second and lessen system requirements.

The PROCESSLOGIC call in the PUSH event handler will immediately cause a task switch to be performed. This will allow a low transition on the diInput1 signal to be seen immediately, making the system more responsive.

One more operational difference between the X-Generation and 2-Series control systems is the event interaction. For example:

```
DIGITAL_INPUT diEvent1, diEvent2;
PUSH diEvent1
    {
       PRINT(“Starting Event 1\n”);
       DELAY(500); // 5 sec delay
       PRINT (“Event 1 done\n”);
    }
PUSH diEvent2
    {
       PRINT (“Starting Event 2\n”);
       DELAY (1500); // 15 sec delay
       PRINT (“Event 2 done\n”);
    }
```

The output from the X-Generation system would be:

```
Starting Event 1
Starting Event 2
Event 2 Done
Event 1 Done
```

The order dictates that the second delay (15 seconds) will hold off the first delay. As soon as the second delay has finished, the first delay is checked. Therefore, the two events complete at approximately the same time (15 seconds).

The output from the 2-Series system would be:

```
Starting Event 1
Starting Event 2
Event 1 Done
Event 2 Done
```

The events run independently. When the 5-seconds expires for the first delay, the first event continues and prints its message. The second delay expires 10 seconds later and the message is displayed.

Language Constructs & Functions

Language Constructs & Functions Overview

Functions take one or more comma-separated parameters and return a result. The following template shows how each language construct and function is explained.

Name:

The name used to refer to the construct or function.

Syntax:

The SIMPL+ specific language requirements for this particular construct or function. This section demonstrates exactly how to enter the statement in a SIMPL+ program.

For completeness, the general syntax for SIMPL+ functions is shown below:

```
<Return Value Type> FunctionName(<Parameter 1 Type> [,
<Parameter 2 Type> ...]);
```

The Types are described as STRING, INTEGER, LONG_INTEGER, SIGNED_INTEGER, and SIGNED_LONG_INTEGER.

If a STRING is specified as a return type, a STRING or STRING_OUTPUT variable may be used.

If an INTEGER or LONG_INTEGER is specified as a return type, an INTEGER, LONG_INTEGER, ANALOG_OUTPUT or DIGITAL_OUTPUT may be used.

If a STRING is specified as a parameter, a STRING, STRING_INPUT, BUFFER_INPUT or literal string (i.e. “Hello””) may be used.

If an INTEGER, LONG_INTEGER, SIGNED_INTEGER or SIGNED_LONG_INTEGER is specified as a parameter, an INTEGER, LONG_INTEGER, ANALOG_INPUT, ANALOG_OUTPUT, DIGITAL_INPUT or DIGITAL_OUTPUT may be used. A literal integer (i.e. 100) may also be used. Note that for DIGITAL_OUTPUT values, a value of 0 is equivalent to digital low, and any other value is a digital high.

Description:

General overview of what this function does.

Parameters (applies to functions only):

Specifics on each of the parameters listed.

Return Value (applies to functions only):

Values placed in the return variable include error conditions. Error conditions are results that occur if one or more of the input values does not have values that are legal for that function.

Example:

A code example of how this function is typically used.

Version:

The version of SIMPL+ in which the construct or function was made available and any revision notes about differences between various versions of SIMPL+. All constructs and functions are available in all subsequent versions except where noted.

Control System:

The control system platform for which the function is valid. Unless specified, the construct or function is valid for both X-Generation (e.g., CEN-TVAV, CNMSX-AV/PRO, CNRACKX/-DP) and 2-Series control systems. SIMPL+ is not available in the control systems preceding the X generation - CNMS, CNRACK/-D/-DP, CNLCOMP/-232, and ST-CP.

Arrays

Various one and two dimensional arrays are supported. All input and output arrays are 1-based, meaning that the first element has index 1, not 0. Internal variables are 0-based, meaning that the first element has index 0. In both cases, the index of the last element is the same as the dimension of the array.

Do not confuse the declaration of the length of STRINGs with the declaration of arrays. E.g. STRING s$[32] is a single string of length 32, and STRING ManyS$[10][32] is an array of 11 strings of length 32 each. You must use the BYTE function to access the character at a particular position in a string, but you can use the array index to access a particular string in an array of strings. Positions in a string are 1-based. Refer to the discussion of Minimum Size Arrays in Declaration Overview on page 45.

One dimensional arrays of the following types are supported:

DIGITAL_INPUT

DIGITAL_OUTPUT

ANALOG_INPUT

ANALOG_OUTPUT

STRING_OUTPUT

BUFFER_OUTPUT

STRUCTURES

One dimensional arrays of strings are also supported, although since the declaration also contains a string length, it looks like a 2-dimensional array:

STRING_INPUT

BUFFER_INPUT

STRING

One and two dimensional arrays of the following types are supported:

INTEGER

LONG_INTEGER

SIGNED_INTEGER

SIGNED_LONG_INTEGER

Declaration Examples:

DECLARATION	**MEANING**
DIGITAL_INPUT in[10];	10 digital inputs, in[1] to in[10]
INTEGER MyArray[10][20];	11 rows by 21 columns of data, from MyArray[0][0] to MyArray[10][20]
STRING PhoneNumbers[100][32];	101 strings that are a maximum of 32 characters long, e.g. PhoneNumbers[0] to PhoneNumbers[100]
STRING_INPUT in$[32];	One input string called in$ that is 32 characters long.
STRING_OUTPUT out$[10];	Ten output strings, out$1 to out$[10]. Their length does not have to be specified.
STRING_INPUT in$[5][32];	Five input strings, in$[1] to in$[5] that are 32 characters long.
<struct_type> myStruct[10];	11 structure elements from myStruct[0] to myStruct[10].

Compiler Directives

Compiler Directives Overview

Compiler directives are used by the SIMPL+ compiler to control attributes of the symbol without generating the actual SIMPL+ code.

#CATEGORY

Name:

#CATEGORY

Syntax:

```
#CATEGORY “<category ID>”
```

Description:

A Category is the name of the folder in the Logic Symbols library tree where the module is shown. To specify a category for a SIMPL+ module, the #CATEGORY directive must be used with a category specified in the list shown in the SIMPL+ Editor. Just click "Edit" then "Insert Category" for a list of categories. Choose one and the appropriate line of code is added to your SIMPL+ program.

Example:

```
#CATEGORY “6” // Lighting
```

If a category ID does not exist in the Symbol Tree Category list, the SIMPL+ module will default to the Miscellaneous category type.

Version:

SIMPL+ Version 3.00

Control System:

2-Series only

#CRESTRON_LIBRARY

Name:

#CRESTRON_LIBRARY

Syntax:

```
#CRESTRON_LIBRARY “<Crestron Library Name>”
```

Description:

Directs the compiler to include code from a Crestron provided library. The module name specified is the Crestron Library Filename without the CSL extension.

Example:

```
#CRESTRON_LIBRARY “Special Integer Functions”
```

Directs the compiler to include the Crestron Library “Special Integer Functions.CSL” from the Crestron SIMPL+ Archive.

Version:

SIMPL+ Version 3.00 - Global variables can no longer be declared within Crestron Library (.csl) files.

SIMPL+ Version 2.00

#DEFAULT_NONVOLATILE

Name:

#DEFAULT_NONVOLATILE

Syntax:

```
#DEFAULT_NONVOLATILE
```

Description:

Program variables retain their value if hardware power is lost. The compiler will default all variables declared within the SIMPL+ module as nonvolatile. Individual variables can use the Volatile keyword to override this default. See also #DEFAULT_VOLATILE on page 36.

Example:

```
#DEFAULT_NONVOLATILE
```

Version:

SIMPL+ Version 3.00

Control System:

2-Series only

#DEFAULT_VOLATILE

Name:

#DEFAULT_VOLATILE

Syntax:

```
#DEFAULT_VOLATILE
```

Description:

Program variables will not retain their value if hardware power is lost. The compiler will default all variables declared within the SIMPL+ module as volatile. Individual variables can use the Nonvolatile keyword to override this default. See also #DEFAULT_NONVOLATILE on page 35.

Example:

```
#DEFAULT_VOLATILE
```

Version:

SIMPL+ Version 3.00

Control System:

2-Series only. On an X-generation system, all variables are non-volatile.

#DEFINE_CONSTANT

Name:

#DEFINE_CONSTANT

Syntax:

```
#DEFINE_CONSTANT <constant_name> <constant_value>
```

Description:

Define a <constant_value> that will be substituted anywhere in the current source file where <constant_name> is used.

Example:

```
#DEFINE_CONSTANT ETX 0x03
INTEGER I;
I=ETX;
```

Assigns the value of 0x03 to the variable I.

Version:

SIMPL+ Version 1.00

#HELP

Name:

#HELP

Syntax:

```
#HELP “<help text>”
```

Description:

Several #HELP lines can be specified. When F1 is hit either on the symbol in the Symbol Library, in either the Program View or the Detail view, the help text will be displayed. If this directive or the #HELP_BEGIN … #HELP_END directive is not present, the help text shown is “NO HELP AVAILABLE”. Note that it is preferable to use the #HELP_BEGIN … #HELP_END directives rather than #HELP since it is easier to edit and read the code.

Example:

```
#HELP “This is line 1 of my help text”
#HELP “This is line 2 of my help text”
```

Version:

SIMPL+ Version 1.00

#HELP_BEGIN … #HELP_END

Name:

#HELP_BEGIN … #HELP_END

Syntax:

```
#HELP_BEGIN
Help Text Line 1
Help Text Line 2
etc.
#HELP_END
```

Description:

The #HELP_BEGIN, #HELP_END pair makes it easier to create help since each line does not need a separate #HELP directive. When F1 is hit either on the symbol in the Symbol Library, in either the Program View or the Detail view, the help text will be displayed. If this directive or #HELP is not present, the help text shown is “NO HELP AVAILABLE”. Note that the text will show up exactly as typed between the begin/end directives (including blank lines).

Example:

```
#HELP_BEGIN
This is help line 1.
This is help line 3.
#HELP_END
```

Version:

SIMPL+ Version 1.00

#HINT

Name:

#HINT

Syntax:

```
#HINT “Hint Text”
```

Description:

The #HINT shows up in the status bar and provides a short tactical clue as to the function of the symbol, in the same way that Crestron-defined built-in symbols do. If the hint is specified, it will be visible when the symbol is highlighted in the User Modules section of the Symbol Library. The text shows up as the symbol name as it is stored on disk, followed by a colon, followed by the text. For example, a symbol with the name “My Symbol” might be stored on disk with the filename MYSYM.USP. If the hint is specified as #HINT “This is my symbol!” then the status bar will show “MYSYM.USP : This is my symbol!”. If no #HINT is specified, then only the filename is shown.

Example:

```
#HINT “This module controls a CNX-PAD8 Switcher”
```

Version:

SIMPL+ Version 1.00

#IF_DEFINED … #ENDIF

Name:

#IF_DEFINED … #ENDIF

Syntax:

```
#IF_DEFINED <constant_name>
<code>
#ENDIF
```

Description:

Results in compilation of the <code> only if <constant_name> has previously been defined. This construct is generally useful for putting in code for debugging purposes, giving the ability to easily turn the debugging on and off during compilation.

Example:

```
#DEFINE_CONSTANT DEBUG 1
DIGITAL_OUTPUT OUT$;
INTEGER I;
FOR(I=0 to 20)
{
#IF_DEFINED DEBUG
PRINT("Loop index I = %d\n", I);
#ENDIF
OUT$ = ITOA(I);
}
```

The value of the loop is printed only if the DEBUG constant is defined. In order to prevent compilation of the code, delete the line that defines the constant or comment it out.

Version:

SIMPL+ Version 2.00

#SYMBOL_NAME

Name:

#SYMBOL_NAME

Syntax:

```
#SYMBOL_NAME “<name of symbol>”
```

Description:

By specifying <name of symbol>, this name will show up on the header of the symbol in the detail view as well as in the USER SIMPL+ section of the Symbol Library. If this directive is not present, the default name shown in the Symbol Library/Program View/Detail view is the name of the USP file as saved on disk. For example, if the file is saved as “Checksum Program.USP”, the tree views will show “Checksum Program” as the name.

Example:

```
#SYMBOL_NAME “My SIMPL+ Program”
```

Version:

SIMPL+ Version 1.00

#USER_LIBRARY

Name:

#USER_LIBRARY

Syntax:

```
    #USER_LIBRARY “<User Library Name>”
```

Description:

Directs the compiler to include code from a User written library. The module name specified is the User Library Filename without the USL extension that is used by User Libraries. Pathnames are not allowed as the USL modules are stored in the User SIMPL+ path (refer to Edit | Preferences | Paths in SIMPL Windows). User libraries can be created by saving a SIMPL+ module as type SIMPL+ library, instead of the default type SIMPL+ file.

Example:

```
   #USER_LIBRARY “My Functions”
```

Directs the compiler to include the User Library “My Functions.USL” from the User SIMPL+ directory.

Version:

SIMPL+ Version 3.00 - Global variables can no longer be declared within User Library (.usl) files.

SIMPL+ Version 2.00

#IF_NOT_DEFINED … #ENDIF

Name:

#IF_NOT_DEFINED … #ENDIF

Syntax:

```
#IF_NOT_DEFINED <constant_name>
<code>
#ENDIF
```

Description:

Results in compilation of the <code> only if <constant_name> has not been previously defined. This construct is generally useful for putting in code for debugging purposes, giving the ability to easily turn the debugging on and off during compilation.

Example:

```
#DEFINE_CONSTANT DEBUG 1
DIGITAL_OUTPUT OUT$;
INTEGER I;
FOR(I=0 to 20)
{
#IF_DEFINED DEBUG
PRINT("Loop index I = %d\n", I);
#ENDIF
#IF_NOT_DEFINED_DEBUG
OUT$ = ITOA(I);
#ENDIF
}
```

The value of the loop is only printed if the DEBUG constant is defined. The output OUT$ is only generated if the debug constant is not defined (if debug mode is not turned on). In order to generate "release" code, the debug constant can be deleted or commented out.

Version:

SIMPL+ Version 2.00

Declarations

Declarations Overview

Declarations control the name, type, and number of inputs and outputs on a SIMPL+ symbol. The name is shown as a cue on the symbol in SIMPL Windows and is used as the variable name in the body of the SIMPL+ program. When the symbol is drawn in SIMPL Windows, the inputs are shown in the order of DIGITAL_INPUTs, ANALOG_INPUTs, STRING_INPUTs. The outputs are shown in the order of DIGITAL_OUTPUTs, ANALOG_OUTPUTs, STRING_OUTPUTs. When specifying a declaration, several variable names can be put after a declaration or multiple declaration statements may be used.

For example:

```
ANALOG_INPUT val1, val2, val3;
```

is equivalent to:

```
ANALOG_INPUT val1, val2;
ANALOG_INPUT val3;
```

Allowable I/O List Combinations

SIMPL+ Version 2.00 and later gives the ability to define arrays in the Input/Output Lists. SIMPL+ version 3.01 and later introduced the ability to declare multiple fixed-size arrays in the input/output lists, and a minimum expanded size to variable-size arrays.

The following are the allowable combinations:

- Zero or more DIGITAL_INPUTs
- Zero or more DIGITAL_INPUT arrays, the last is variable-size, the others are fixed-size.
- Zero or more ANALOG_INPUTs, STRING_INPUTs, or BUFFER_INPUTs in any combination.
- Zero or more ANALOG_INPUT, STRING_INPUT, or BUFFER_INPUT array, the last is variable-size, the others are fixed-size.
- Zero or more DIGITAL_OUPUTs
- Zero or more DIGITAL_OUTPUT array, the last is variable-size, the others are fixed-size.
- Zero or more ANALOG_OUTPUTs, STRING_OUTPUTs in any combination.
- Zero or more ANALOG_OUTPUT or STRING_OUTPUT array, the last is variable-size, the others are fixed-size.

Fixed and Variable Size Arrays

Although SIMPL+ symbols can only handle one variable size DIGITAL_INPUT array, one variable-size DIGITAL_OUTPUT array, one variable-size ANALOG/STRING/BUFFER input array, and one variable size ANALOG/STRING/OUTPUT array, it is convenient to be able to refer to other inputs and outputs with array notation. Therefore, SIMPL+ allows an unlimited number of fixed-size input or output arrays, that are essentially single input or output values but array notation can be used. Every member of these fixed-size arrays is always shown in the symbol. All arrays, except the last one of each kind, are fixed-size arrays. The last one is variable-size, meaning that the symbol initially shows the first array value. The user can press ALT+ to expand the symbol to its maximum number of array inputs or outputs. In addition, a minimum size can be declared in all variable-size arrays, meaning that the minimum number of array members is always shown, not just the first one, and the array can be expanded from there.

NOTE: The minimum array size number must be from 1 to the size of the array. If a minimum array size is specified on any array, but it is the last one within any type, it will be a compile error.

Example:

```
DIGITAL_INPUT YesVotes[10]
DIGITAL_INPUT NoVotes[10}
DIGITAL_INPUT AbstainVotes[10,5];
```

The symbol will show 10 digital inputs labelled: YesVotes[1], YesVotes[2] ...YesVotes[10], followed by 10 more labelled: NoVotes[1], NoVotes[2] ...NoVotes[10], followed by 5 labelled: AbstainVotes[1], AbstainVotes[2] ...AbstainVotes[5]. You can continue to expand the last one up to AbstainVotes[10].

Predefined Names:

The names "on" and "off" are reserved. Assigning "on" to a variable sets the variable to 1, assigning "off" sets that variable to 0.

The following shows equivalent, given that VALUE is a DIGITAL_OUTPUT:

```
VALUE = 1;   and   VALUE = on;
VALUE = 0;   and   VALUE = off;
```

ANALOG_INPUT

Name:

ANALOG_INPUT

Syntax:

```
ANALOG_INPUT <var1>[,<var2>...];
ANALOG_INPUT <var[size]>;
ANALOG_INPUT <var[size[,<min>]]>
```

Description:

Routes analog inputs from the outside SIMPL program into a SIMPL+ program with the specified variable names. ANALOG_INPUT values are 16-bit numbers. They are treated as signed or unsigned values inside of a SIMPL+ program depending on the operators or functions being used.

NOTE: ANALOG_INPUT variables may not be passed to functions in Version 3.00 for the 2-Series Control Systems. If you need to pass an ANALOG_INPUT variable to a function, assign it to a locally declared variable and pass that variable to the function.

NOTE: <min> is the number of inputs shown at a minimum in SIMPL Windows. The Default is 1. The user can expand the minimum up to the full size. Only the last array of a type can have <min>. Refer to Arrays on page 31, and Declarations on page 45.

For an array of ANALOG_INPUTs, the maximum value of SIZE is 65535. Valid indices are 1 through the specified size.

Example:

```
ANALOG_INPUT ramp1;
```

Signifies that one analog input is coming into the SIMPL+ program from the SIMPL Program.

```
ANALOG_INPUT light_levels[25];
```

Signifies that up to 25 analog inputs are coming into the SIMPL+ program from the SIMPL Program, referenced as light_levels[1] through light_levels[25]. One is shown as a minimum but the symbol input can be expanded by the user up to 25.

```
ANALOG_INPUT temp_set_pts[20,4];
```

Signifies that up to 20 analog inputs exist, referenced as temp_set_pts[1] through temp_set_pts[20]. Four are shown at a minimum, and the symbol inputs can be expanded by the user up to 20.

Version:

SIMPL+ Version 2.00 for ANALOG_INPUT arrays, 3.01 for fixed arrays and minimum sizes.
SIMPL+ Version 2.00 for ANALOG_INPUT arrays.
SIMPL+ Version 1.00 for everything else.

ANALOG_OUTPUT

Name:

ANALOG_OUTPUT

Syntax:

```
ANALOG_OUTPUT <var1>[,<var2>...];
ANALOG_OUTPUT <var[size]>;
ANALOG_OUTPUT<var[size[,<min>]]>;
```

Description:

Routes a value from the SIMPL+ program to the SIMPL program as an analog value. ANALOG_OUTPUT values are 16-bit numbers. They are treated as signed or unsigned values inside of a SIMPL+ program depending on the operators or functions being used. Refer to the discussion on Arrays on page 46.

NOTE: ANALOG_OUTPUTs may be jammed with other analog values from a SIMPL program (i.e., from a RAMP or other analog logic, even other SIMPL+ symbols). When such an output is jammed, the new value is read back into the SIMPL+ symbol and the value of the output is altered.

NOTE: <min> is the number of outputs shown at a minimum in SIMPL Windows. The Default is 1. The user can expand the minimum up to the full size. Only the last array of a type can have <min>. Refer to Arrays on page 31, and Declarations on page 45.

In X-Generation Control Systems, the logic process only sees the last analog that was posted after the SIMPL+ module tasks switched away. Therefore, in a loop that iterates from 1 to 10000, only a few of the values will be seen by the logic process. If all values should be seen to by the logic process, a PROCESSLOGIC statement is required after the assignment to the ANALOG_OUTPUT.

When the SIMPL+ program writes to the ANALOG_OUTPUT, the new value is posted immediately. Therefore, if the value is read back after being assigned, the new value is read back (unlike a DIGITAL_OUTPUT on X-Generation control systems).

In the 2-Series Control Systems, the logic process sees ALL values that are assigned to the ANALOG_OUTPUT. No PROCESSLOGIC is required.

For an array of ANALOG_OUTPUTs, the maximum value of SIZE is 65535. Valid indices are 1 through the specified size.

Example:

```
ANALOG_OUTPUT LEVEL;
```

Signifies that one analog input is being sent from the SIMPL+ program to the SIMPL program.

```
ANALOG_OUTPUT LEVELS[25];
```

Signifies that up to 25 analog outputs, referred to as LEVELS[1] through LEVELS[25] are being sent from the SIMPL+ program to the SIMPL program.

```
ANALOG_OUTPUT LEVELS[25,5];
```

Signifies same as above, except that a minimum of 5 are shown at any time.

NOTE: If LEVEL or any of the elements from LEVELS is jammed from outside the symbol, it will take on that new jammed value.

NOTE: You should use isSignalDefined to test whether the output is connected to an actual signal in the SIMPL Windows program before assigning a value to it. If you assign a value and there is no signal, a message is placed in the system error log.

Version:

SIMPL+ Version 3.01 - Fixed size arrays and minimum sizes.

SIMPL+ Version 3.00 - Can no longer be passed to functions by reference. (2-Series Control Systems only)

SIMPL+ Version 2.00 for ANALOG_OUTPUT arrays.

SIMPL+ Version 1.00 for everything else.

BUFFER_INPUT

Name:

BUFFER_INPUT

Syntax:

```
BUFFER_INPUT <var1[max_length]>[,<var2[max_length]>...];
BUFFER_INPUT <var[size][max_length]>;
BUFFER_INPUT<var[size[,<min>]][max_length]>;
```

Description:

Routes serial inputs from the outside SIMPL program into a SIMPL+ program under the specified variable names. This is used when a serial string coming from a communications port needs to be processed by a SIMPL+ program. When new data comes in on a BUFFER_INPUT, the data is appended to the end of a BUFFER_INPUT. If the buffer is full, the contents are shifted up and the new data is appended to the end. This differs from STRING_INPUTs in that new data entering into a STRING_INPUT variable replaces the previous string contents. BUFFER_INPUTs may be processed with string handling functions. The GETC function may be used to read a character from the beginning of the buffer and shift the contents up by 1. Buffer inputs may be written to, so their data space may be used as a storage spot for doing something such as parsing through a string without declaring temporary storage. Refer to the discussion on arrays on page 46.

NOTE: BUFFER_INPUT variables may not be passed to functions in Version 3.00 for the 2-Series Control Systems. If you need to pass a BUFFER_INPUT variable to a function, assign it to a locally declared variable and pass that variable to the function.

NOTE: <min> is the number of inputs shown at a minimum in SIMPL Windows. The Default is 1. The user can expand the minimum up to the full size. Only the last array of a type can have <min>. Refer to Arrays on page 31, and Declarations on page 45.

MAX_LENGTH may be a value up to 255 in SIMPL+ Version 1.00. SIMPL+ Version 2.00 and later allow for MAX_LENGTH to be up to 65535. For an array of BUFFER_INPUTs, the maximum value of SIZE is 65535. Valid indices are 1 through the specified size.

Example:

```
BUFFER_INPUT FromComPort[100];
```

Signifies that a 100 character buffer with the name “FromComPort” is specified as a BUFFER_INPUT.

```
BUFFER_INPUT ComBuffers[2][100];
```

Signifies that two 100 character buffers have been set up that may be referenced with the names ComBuffers[1] through ComBuffers[2].

```
BUFFER_INPUT ComBuffers[2,2][100];
```

Same as above except both are always shown on the symbol.

Version:

SIMPL+ Version 3.01 for fixed size arrays and minimum sizes.
SIMPL+ Version 2.00 for BUFFER_INPUT arrays and MAX_LENGTH to 65535.
SIMPL+ Version 1.00 for everything else.

DIGITAL_INPUT

Name:

DIGITAL_INPUT

Syntax:

```
DIGITAL_INPUT <var1>[,<var2>...];
DIGITAL_INPUT <var[size]>;
DIGITAL_INPUT <var[size[,min]]>;
```

Description:

Routes digital inputs from the outside SIMPL program into a SIMPL+ program under the specified variable names. DIGITAL_INPUT values are either 0 (digital low) or 1 (digital high). Refer to the discussion on arrays on page 46.

NOTE: DIGITAL_INPUT variables may not be passed to functions in Version 3.00 for the 2-Series Control Systems. If you need to pass a DIGITAL_INPUT variable to a function, assign it to a locally declared variable and pass that variable to the function.

NOTE: <min> is the number of inputs shown at a minimum in SIMPL Windows. The Default is 1. The user can expand the minimum up to the full size. Only the last array of a type can have <min>. Refer to Arrays on page 31, and Declarations on page 45.

For an array of DIGITAL_INPUTs, the maximum value of SIZE is 65535. Valid indices are 1 through the specified size.

Example:

```
DIGITAL_INPUT osc_in, toggle_in;
```

Signifies that two digital inputs are coming into the SIMPL+ program from the SIMPL Program.

```
DIGITAL_INPUT status_bits[8];
```

Signifies that up to eight digital inputs are coming into the SIMPL+ program from the SIMPL Program, referenced under the names status_bits[1] through status_bits[8].

```
DIGITAL_INPUT flags[8,2];
```

Signifies up to eight digital inputs, with at least two shown.

Version:

SIMPL+ Version 3.01 for fixed arrays and minimum sizes.

SIMPL+ Version 2.00 for DIGITAL_INPUT arrays.

SIMPL+ Version 1.00 for everything else.

DIGITAL_OUTPUT

Name:

DIGITAL_OUTPUT

Syntax:

```
DIGITAL_OUTPUT <var1>[,<var2>...];
DIGITAL_OUTPUT <var[size]>;
DIGITAL_OUTPUT <var[size[,<min>]]>;
```

Description:

Routes a value from the SIMPL+ program to a SIMPL program. If a value different from 0 is placed on a DIGITAL_OUTPUT, the digital signal in the SIMPL program is set high when the control system processes the logic.

Refer to the discussion on arrays on page 46.

NOTE: DIGITAL_OUTPUTs may be jammed with other digital values from a SIMPL program (i.e., from a BUFFER or other jammable digital logic, even other SIMPL+ symbols). When such an output is jammed, the new value is read back into the SIMPL+ symbol and the value of the output is altered.

NOTE: <min> is the number of outputs shown at a minimum in SIMPL Windows. The Default is 1. The user can expand the minimum up to the full size. Only the last array of a type can have <min>. Refer to Arrays on page 31, and Declarations on page 45.

NOTE: You should use isSignalDefined to test whether the output is connected to an actual signal in the SIMPL Windows program before assigning a value to it. If you assign a value and there is no signal, a message is placed in the system error log.

In X-Generation Control Systems, if a new value is assigned to the DIGITAL_OUTPUT from the SIMPL+ program, the value read back from it within the SIMPL+ program will have the original state until the logic is serviced. For example, if a DIGITAL_OUTPUT has a value of 0, and the value 1 is written to it, the value read back will be 0 until the system processes the rest of the logic attached to that SIMPL+ symbol. This is unlike an ANALOG_OUTPUT. If every change of a DIGITAL_OUTPUT is required to be seen by the logic, a PROCESSLOGIC statement is required after the assignment to the DIGITAL_OUTPUT.

In the 2-Series Control Systems, the logic process sees ALL values that are assigned to the DIGITAL_OUTPUT. No PROCESSLOGIC is required. As an example, if the following code is used in the 2-Series Control Systems:

```
DIGITAL_OUTPUT State1;
State1=1;
State1=0;
```

The logic will end up seeing a short pulse.

For an array of DIGITAL_OUTPUTs, the maximum value of SIZE is 65535. Valid indices are 1 through the specified size.

Example:

```
DIGITAL_OUTPUT State1, State2;
```

Signifies that two digital signals are to be sent to a SIMPL program from this SIMPL+ program.

NOTE: For example, if State1 is jammed high via a BUFFER from outside the SIMPL+ program, the value of State1 becomes 1 and should be handled accordingly in the SIMPL+ code.

```
DIGITAL_OUTPUT state_bits[3];
```

Signifies that up to three digital signals are to be sent to a SIMPL program from this SIMPL+ program. The names are referred to as state_bits[1] through state_bits[3]. The same jamming rules apply as in the previous example.

```
DIGITAL_OUTPUT state_bits[3,3];
```

Same as above except all three are always shown on the symbol.

Version:

SIMPL+ Version 3.01 - Fixed arrays and minimum sizes.

SIMPL+ Version 3.00 - can no longer be passed to functions by reference. (2-Series Control Systems only)

SIMPL+ Version 2.00 for DIGITAL_OUTPUT arrays.

SIMPL+ Version 1.00 for everything else.

INTEGER

Name:

INTEGER

Syntax:

```
INTEGER <var1>[,<var2>...];
INTEGER <var1>[size] [,<var2>[size]…];
INTEGER <var1>[rows1][columns1] [,<var2>[rows2][columns2]…];
```

Description:

The first form declares an integer value that is local to this SIMPL+ program. INTEGER values are 16-bit quantities and are treated the same as ANALOG_INPUT values and range from 0-65535.

The second form declares a one-dimensional array of INTEGER values.

The third form declares a two-dimensional array of INTEGER values. A two-dimensional array can be thought of as a table or matrix.

The values for SIZE, ROWS, and COLUMNS may be up to 65535.

An INTEGER array element may be used anywhere an INTEGER is legal. Array elements are referenced by using the name followed by [element] for one-dimensional arrays or [element1][element2] for two-dimensional arrays. The element number may range from 0 to the element size. For example, if an array is declared as NUM[2], then legal elements are NUM[0], NUM[1], and NUM[2]. The bracket notation is often called an array subscript.

NOTE: (X-Gen) The values of INTEGERs declared outside of functions are non-volatile. If the system is powered down and up, the variables will take the previous values. If programs are changed and uploaded, the values are not preserved.

NOTE: (2-Series) INTEGERs can be volatile or non-volatile. The default is defined using the compiler directives #DEFAULT_NONVOLATILE or #DEFAULT_VOLATILE or overridden using the nonvolatile or volatile keywords.

NOTE: If no RETURN statement is encountered, the function automatically returns a 0.

Example:

```
INTEGER temp_level;
```

Specifies one locally declared INTEGER in this SIMPL+ program

```
INTEGER CommandBytes[2];
```

Specifies an array of three INTEGERS that can be referenced under the name CommandBytes. In pictorial form, it appears as:

CommandBytes[0]	CommandBytes[1]	CommandBytes[2]

```
INTEGER Matrix[4][3];
```

Specifies a two-dimensional array of integers five rows deep by four columns wide.

In pictorial form, it appears as:

Matrix[0][0]	Matrix[0][1]	Matrix[0][2]	Matrix[0][3]
Matrix[1][0]	Matrix[1][1]	Matrix[1][2]	Matrix[1][3]
Matrix[2][0]	Matrix[2][1]	Matrix[2][2]	Matrix[2][3]
Matrix[3][0]	Matrix[3][1]	Matrix[3][2]	Matrix[3][3]
Matrix[4][0]	Matrix[4][1]	Matrix[4][2]	Matrix[4][3]

NOTE: The subscripts of an array may be an expression, i.e.:

```
INTEGER location[5], room;
               room = 2;
               location[room] = 10;
```

Version:

SIMPL+ Version 1.00

SIMPL+ Version 2.00 allowed INTEGER to be declared inside of functions.

LONG_INTEGER

Name:

LONG_INTEGER

Syntax:

```
LONG_INTEGER <var1>[,<var2>...];
LONG_INTEGER <var1>[size] [,<var2>[size]…];
LONG_INTEGER <var1>[rows1][columns1]
[,<var2>[rows2][columns2]…];
```

Description:

The first form declares a long value that is local to this SIMPL+ program. LONG_INTEGER values are 32-bit quantities ranging from 0-4294967296.

The second form declares a one-dimensional array of LONG_INTEGER values.

The third form declares a two-dimensional array of LONG_INTEGER values. A two-dimensional array can be thought of as a table or matrix.

The values for SIZE, ROWS, and COLUMNS may be up to 65535.

A LONG_INTEGER array element may be used anywhere a LONG_INTEGER is legal. Array elements are referenced by using the name followed by [element] for one-dimensional arrays or [element1][element2] for two-dimensional arrays. The element number may range from 0 to the element size. For example, if an array is declared as NUM[2], then legal elements are NUM[0], NUM[1], and NUM[2]. The bracket notation is often called an array subscript.

NOTE: (2-Series) LONG_INTEGERs can be volatile or non-volatile. The default is defined using the compiler directives #DEFAULT_NONVOLATILE or #DEFAULT_VOLATILE or overridden using the nonvolatile or volatile keywords.

Example:

```
LONG_INTEGER temp_level;
```

Specifies one locally declared LONG_INTEGER in this SIMPL+ program

```
LONG_INTEGER CommandBytes[2];
```

Specifies an array of three LONG_INTEGERs that can be referenced under the name CommandBytes. In pictorial form, it appears as:

CommandBytes[0]	CommandBytes[1]	CommandBytes[2]

```
LONG_INTEGER Matrix[4][3];
```

Specifies a two-dimensional array of LONG_INTEGERs five rows deep by four columns wide.

In pictorial form, it appears as:

Matrix[0][0]	Matrix[0][1]	Matrix[0][2]	Matrix[0][3]
Matrix[1][0]	Matrix[1][1]	Matrix[1][2]	Matrix[1][3]
Matrix[2][0]	Matrix[2][1]	Matrix[2][2]	Matrix[2][3]
Matrix[3][0]	Matrix[3][1]	Matrix[3][2]	Matrix[3][3]
Matrix[4][0]	Matrix[4][1]	Matrix[4][2]	Matrix[4][3]

NOTE: The subscripts of an array may be an expression, i.e.:

```
LONG_INTEGER location[5], room;
                 room = 2;
                 location[room] = 10;
```

Version:

SIMPL+ Version 3.00.01

Control System

2-Series Only

SIGNED_INTEGER

Name:

SIGNED_INTEGER

Syntax:

```
SIGNED_INTEGER <var1>[,<var2>...];
SIGNED_INTEGER <var1>[size] [,<var2>[size]…];
SIGNED_INTEGER <var1>[rows1][columns1]
[,<var2>[rows2][columns2]…];
```

Description:

The first form declares an integer value that is local to this SIMPL+ program. SIGNED_INTEGER values are 32-bit quantities ranging from -32678 to 32767.

The second form declares a one-dimensional array of SIGNED_INTEGER values.

The third form declares a two-dimensional array of SIGNED_INTEGER values. A two-dimensional array can be thought of as a table or matrix.

The values for SIZE, ROWS, and COLUMNS may be up to 65535.

A SIGNED_INTEGER array element may be used anywhere an SIGNED_INTEGER is legal. Array elements are referenced by using the name followed by [element] for one-dimensional arrays or [element1][element2] for two-dimensional arrays. The element number may range from 0 to the element size. For example, if an array is declared as NUM[2], then legal elements are NUM[0], NUM[1], and NUM[2]. The bracket notation is often called an array subscript.

NOTE: (2-Series) SIGNED_INTEGERs can be volatile or non-volatile. The default is defined using the compiler directives #DEFAULT_NONVOLATILE or #DEFAULT_VOLATILE or overridden using the nonvolatile or volatile keywords.

Example:

```
SIGNED_INTEGER temp_level;
```

Specifies one locally declared SIGNED_INTEGER in this SIMPL+ program

```
SIGNED_INTEGER CommandBytes[2];
```

Specifies an array of three SIGNED_INTEGERS that can be referenced under the name CommandBytes. In pictorial form, it appears as:

CommandBytes[0]	CommandBytes[1]	CommandBytes[2]

```
SIGNED_INTEGER Matrix[4][3];
```

Specifies a two-dimensional array of integers five rows deep by four columns wide. In pictorial form, it appears as:

Matrix[0][0]	Matrix[0][1]	Matrix[0][2]	Matrix[0][3]
Matrix[1][0]	Matrix[1][1]	Matrix[1][2]	Matrix[1][3]
Matrix[2][0]	Matrix[2][1]	Matrix[2][2]	Matrix[2][3]
Matrix[3][0]	Matrix[3][1]	Matrix[3][2]	Matrix[3][3]
Matrix[4][0]	Matrix[4][1]	Matrix[4][2]	Matrix[4][3]

NOTE: The subscripts of an array may be an expression, i.e.:

```
SIGNED_INTEGER location[5], room;
               room = 2;
               location[room] = 10;
```

Version:

SIMPL+ Version 3.00.06

Control System:

2-Series Only

SIGNED_LONG_INTEGER

Name:

SIGNED_LONG_INTEGER

Syntax:

```
SIGNED_LONG_INTEGER <var1>[,<var2>...];
SIGNED_LONG_INTEGER <var1>[size] [,<var2>[size]…];
SIGNED_LONG_INTEGER <var1>[rows1][columns1]
[,<var2>[rows2][columns2]…];
```

Description:

The first form declares a long value that is local to this SIMPL+ program. SIGNED_LONG_INTEGER values are 32-bit quantities ranging from -2,147,483,647 to 2,147,483,647.

The second form declares a one-dimensional array of SIGNED_LONG_INTEGER values.

The third form declares a two-dimensional array of SIGNED_LONG_INTEGER values. A two-dimensional array can be thought of as a table or matrix.

The values for SIZE, ROWS, and COLUMNS may be up to 65535.

A SIGNED_LONG_INTEGER array element may be used anywhere a SIGNED_LONG_INTEGER is legal. Array elements are referenced by using the name followed by [element] for one-dimensional arrays or [element1][element2] for two-dimensional arrays. The element number may range from 0 to the element size. For example, if an array is declared as NUM[2], then legal elements are NUM[0], NUM[1], and NUM[2]. The bracket notation is often called an array subscript.

NOTE: (2-Series) SIGNED_LONG_INTEGERs can be volatile or non-volatile. The default is defined using the compiler directives #DEFAULT_NONVOLATILE or #DEFAULT_VOLATILE or overridden using the nonvolatile or volatile keywords.

Example:

```
SIGNED_LONG_INTEGER temp_level;
```

Specifies one locally declared SIGNED_LONG_INTEGER in this SIMPL+ program

```
SIGNED_LONG_INTEGER CommandBytes[2];
```

Specifies an array of three SIGNED_LONG_INTEGERs that can be referenced under the name CommandBytes. In pictorial form, it appears as:

CommandBytes[0]	CommandBytes[1]	CommandBytes[2]

```
SIGNED_LONG_INTEGER Matrix[4][3];
```

Specifies a two-dimensional array of SIGNED_LONG_INTEGERs five rows deep by four columns wide.

In pictorial form, it appears as:

Matrix[0][0]	Matrix[0][1]	Matrix[0][2]	Matrix[0][3]
Matrix[1][0]	Matrix[1][1]	Matrix[1][2]	Matrix[1][3]
Matrix[2][0]	Matrix[2][1]	Matrix[2][2]	Matrix[2][3]
Matrix[3][0]	Matrix[3][1]	Matrix[3][2]	Matrix[3][3]
Matrix[4][0]	Matrix[4][1]	Matrix[4][2]	Matrix[4][3]

NOTE: The subscripts of an array may be an expression, i.e.:

```
SIGNED_LONG_INTEGER location[5], room;
                room = 2;
                location[room] = 10;
```

Version:

SIMPL+ Version 3.00.06

Control System

2-Series Only

STRING

Name:

STRING

Syntax:

```
STRING <var1[size1]>[,<var2[size2]>...];
STRING <var1[num_elements1][num_characters1]>[,
<var2[num_elements2][num_characters2]>...];
```

Description:

Declares a string that is local to this SIMPL+ program. Strings are of arbitrary length, so a maximum size must be specified. When a STRING variable has new data assigned to it, the old data is lost.

NOTE: Strings in Version 3.00 for the 2-Series Control Systems may not be passed by value to a function. They must be passed by reference.

NOTE: If no Return Value is specified within an String_Function, then an empty string (0) will be returned by default.

When used in its second form, a one-dimensional array of strings is allocated. The array has num_elements+1 elements, and num_characters per element allocated. The legal indices for referencing the strings are 0 through num_elements.

The value of SIZE and NUM_CHARACTER may be up to 255 in SIMPL+ Version 1.00. In SIMPL+ Version 2.00 and later, they may be up to 65535. The value of NUM_ELEMENTS may be up to 65535.

NOTE: (X-Gen) The values of STRINGs declared are non-volatile. If the system is powered down and up, the variables will take on their previous values. If programs are changed and uploaded, the values are not preserved.

NOTE: (2-Series) STRINGs can be volatile or non-volatile. The default is defined using the compiler directives #DEFAULT_NONVOLATILE or #DEFAULT_VOLATILE or overridden using the nonvolatile or volatile keywords.

Example:

```
STRING temp$[10];
```

Signifies that one local STRING is declared in this SIMPL+ program.

```
STRING temp$[2][10];
```

Signifies that three strings of 10 characters long have been allocated.

To assign values, the following would be legal:

```
temp$[0]=”Val1”;
temp$[1]=”Val2”;
temp$[2]=”Val3”;
```

Version:

SIMPL+ Version 2.00 for SIZE and NUM_CHARACTER up to 65535.

SIMPL+ Version 1.00 for everything else.

STRING_INPUT

Name:

STRING_INPUT

Syntax:

```
STRING_INPUT <var1[max_size1]>[,<var2[max_size2]>...];
STRING_INPUT <var[size][max_size]>;
STRING_INPUT <var[size[,<min>]][max_size]>;
```

Description:

Routes serial inputs from the outside SIMPL program into a SIMPL+ program under the specified variable names. Strings are of arbitrary length, so a maximum size must be specified. Upon receiving new data, the value is cleared and the new string is put in. Strings received greater than the specified size are truncated to the size in the declaration. String inputs may be written to, so their data space may be used as a storage spot for doing something such as parsing through a string without declaring temporary storage. Refer to the discussion on arrays on page 46.

NOTE: STRING_INPUT variables may not be passed to functions in Version 3.00 for the 2-Series Control Systems. If you need to pass a STRING_INPUT variable to a function, assign it to a locally declared variable and pass that variable to the function.

NOTE: <min> is the number of inputs shown at a minimum in SIMPL Windows. The Default is 1. The user can expand the minimum up to the full size. Only the last array of a type can have <min>. Refer to Arrays on page 31, and Declarations on page 45

The value of SIZE and NUM_CHARACTER may be up to 255 in SIMPL+ Version 1.00. In SIMPL+ Version 2.00 and later, they may be up to 65535. For an array of STRING_INPUTs, the maximum value of SIZE is 65535.

Example:

```
STRING_INPUT FirstName[100], SecondName[25];
```

Signifies that two serial inputs are coming into the SIMPL+ program from the SIMPL Program. The first one may only be a maximum of 100 characters, the second may only be a maximum of 25 characters. If an input is longer than the specified length, everything after the specified limit is lost.

```
STRING_INPUT DataBaseNames[9][100];
```

Signifies that 9 serial inputs are coming into the SIMPL+ program from the SIMPL program. Each name has a 100 character limit. The names are referenced as DataBaseNames[1] through DataBaseNames[9].

```
STRING_INPUT Database Names [9,3][100];
```

Same as above except at least three are shown at all times.

Version:

SIMPL+ Version 3.01 for fixed arrays and minimum sizes.
SIMPL+ Version 2.00 for STRING_INPUT arrays and SIZE, NUM_CHARACTER to 65535.
SIMPL+ Version 1.00 for everything else.

STRING_OUTPUT

Name:

STRING_OUTPUT

Syntax:

```
STRING_OUTPUT <var1>[,<var2>...];
STRING_OUTPUT <var[size]>;
STRING_OUTPUT <var[size[,<min>]][size]>;
```

Description:

Routes serial strings from the SIMPL+ program to the SIMPL program. A string length is not required as the output string buffer management is performed by the operating system. Refer to the discussion on arrays on page 46.

NOTE: These outputs may be jammed with other serial string signals in the SIMPL program, although the value does not propagate back into the SIMPL+ symbol.

NOTE: The maximum string length for a STRING_OUTPUT is 255 characters. Assigning a string with a length of more than 255 will result in a loss of data.

NOTE: You should use isSignalDefined to test whether the output is connected to an actual signal in the SIMPL Windows program before assigning a value to it. If you assign a value and there is no signal, a message is placed in the system error log.

NOTE: <min> is the number of outputs shown at a minimum in SIMPL Windows. The Default is 1. The user can expand the minimum up to the full size. Only the last array of a type can have <min>. Refer to Arrays on page 31, and Declarations on page 45.

The value of a STRING_OUTPUT cannot be read. If knowledge of the value of the STRING_OUTPUT is required, the value to be written to the STRING_OUTPUT can also be written to a STRING for local storage.

In X-Generation Control Systems, if several values are issued to a STRING_OUTPUT, the logic will only see the last value written to the STRING_OUTPUT when the SIMPL+ program task switches away. If all values are required to be seen by the logic, a PROCESSLOGIC statement is required after writing to the STRING_OUTPUT.

In the 2-Series Control Systems, all values written to a STRING_OUTPUT are maintained. The logic will see each value of the STRING_OUTPUT. No PROCESSLOGIC is required.

For an array of STRING_OUTPUTs, the maximum value of SIZE is 65535. Valid indices are 1 through the specified size.

Example:

```
STRING_OUTPUT TheName$;
```

Signifies one string called TheName$ that is generated by the SIMPL+ program and sent to the SIMPL program.

```
STRING_OUTPUT SortedNames$[5];
```

Specifies five strings that are generated by the SIMPL+ program and sent to the SIMPL program. The names are referred to as SortedNames[1] through SortedNames[5].

```
STRING_OUTPUT SortedNames$[5,5];
```

Same as above except all five are always shown.

Version:

SIMPL+ Version 3.01 - Fixed size arrays and minimum sizes.

SIMPL+ Version 3.00 - can no longer be passed to functions by reference. (2-Series Control Systems only)

SIMPL+ Version 2.00 for STRING_OUTPUT arrays.

SIMPL+ Version 1.00 for everything else.

STRUCTURES

A structure is a collection of one or more variables grouped together under a single name. These variables, called structure fields or members, may consist of both integer and string datatypes. Structures help organize related data because they allow variables to be grouped together as a unit instead of as separate entities.

Structure datatypes can only be defined globally. Variables of a defined structure datatype may be declared both globally and locally and passed as function arguments. Structures are always passed to functions by reference. INTEGER, LONG_INTEGER, SIGNED_INTEGER, SIGNED_LONG_INTEGER and STRING are the only SIMPL+ datatypes allowed to be used as structure member fields. INTEGER and LONG_INTEGER can include 1 and 2 dimensional arrays. String arrays are not permitted.

The syntax for defining a structure is as follows:

```
STRUCTURE struct_name
{
type member1;
type member2;
.
.
.
type memberN;
};
```

The keyword, STRUCTURE, tells the compiler that a new datatype is being defined. Each type is one of the SIMPL+ datatypes, INTEGER, LONG_INTEGER, SIGNED_INTEGER, SIGNED_LONG_INTEGER or STRING. Struct_name is the name for the structure that will be used as the new datatype.

Declaring a variable of a structure datatype is as follows:

```
struct_name var_name;
```

An example of a structure would be an entry in a phone book. The phone book contains many entries, all containing the same three pieces of information: the person's name, address and phone number. The structure would be defined as follows:

```
STRUCTURE PhoneBookEntry
{
STRING Name[50];
STRING Address[100];
STRING PhoneNumber[20];
};
PhoneBookEntry OneEntry;
PhoneBookEntry Entry[500];
```

In this example, the name, PhoneBookEntry, is the datatype defined that will encapsulate the structure fields, Name, Address and PhoneNumber. Two variables are then defined to be of this datatype. The variable, OneEntry, is a variable that contains one instance of the datatype, PhoneBookEntry.

The variable, Entry, is then defined to be an array of the datatype, PhoneBookEntry consisting of 501 individual instances, namely Entry[0] to Entry[500].

To access a structure's field, the structure's declared variable name is used, followed by a period (also known as the 'dot' or 'dot operator'), then followed by a structure member variable name.

From the example above, accessing the Name field from the declared variable would be written as follows:

```
OneEntry.Name
```

or

```
Entry[5].Name
```

Using this in a SIMPL+ statement might look as follows:

```
If ( OneEntry.Name = "David" )
Return;
If ( Entry[5].Name = "David" )
Return;
```

Passing structures as function arguments is as follows:

```
FUNCTION myFunction ( PhoneBookEntry argOneEntry,
PhoneBookEntry argEntry[] )
{
if ( argOneEntry.Name = "David" )
return;
if ( argEntry[5].Name = "David" )
return;
}
```

Version:

SIMPL+ Version 3.00.02

Control System

2-Series Only

Declaration Modifiers

Volatile

Name:

Volatile

Syntax:

```
Volatile
```

NOTE: This is not a declaration but a declaration modifier. It works only in conjunction with another declaration keyword.

Description:

Global integer and string program variables will not retain their value if hardware power is lost.

Example:

```
Volatile integer n;
Volatile string s[100];
```

Version:

SIMPL+ Version 3.00

Control System

2-Series Only . The X-generation compiler will give an error message saying that all variables are non-volatile.

Nonvolatile

Name:

Nonvolatile

Syntax:

```
Nonvolatile
```

NOTE: This is not a declaration but a declaration modifier. It works only in conjunction with another declaration keyword.

Description:

Global integer and string program variables will retain their value if hardware power is lost.

Example:

```
Nonvolatile integer n;
Nonvolatile string s[100];
```

Version:

SIMPL+ Version 3.00

Control System:

2-series only. The X-generation processors will give a message that says all variables are non-volatile.

E-mail Functions

Important SendMail Considerations

1. In the SIMPL+ function call to “Send Mail”, the parameters “Mailserv”, “To” and “From” fields are MANDATORY, whereas “cc”, “subject” and “message” are not.
2. Only the “SMTP AUTH” authentication type with “LOGIN” authentication scheme is supported for now.
3. Questions for the ISP/e-mail service provider to determine compatibility with the SEND MAIL feature.
   A. Does the ISP/service provider support NON-WEB clients?
   B. Does the ISP/service provider support “SMTP AUTH” authentication type with “LOGIN” authentication scheme?
   C. For example: the e-mail provider SBC YAHOO supports web as well as non web clients. For non web clients, one of the mail servers to communicate with is SMTPAUTH.FLASH.NET. This mail server supports SMTP AUTH and LOGIN auth scheme.
4. SEND MAIL client queries the mail server to determine the authentication type and scheme and returns an “unsupported” error (error # -9) if the mail-server does not support LOGIN scheme; however if the client is unable to determine information regarding the schemes supported, it will go ahead and try to send out the e-mail to the intended recipients, but the server may refuse to relay it to external destinations. This will return a “failure” code, which is a POSITIVE integer (Refer to E-mail Function Return Error Codes on page 72).
5. For mail servers needing no authentication, the “username” and “password” field are set to an EMPTY STRING (““). Again, as in (4) above there is no guarantee that the mail-server will relay the e-mail to external destinations.
6. In case of an error/failure, the first occurring error/failure code is returned.
7. If the message line exceeds 998 characters without a <CR-LF> sequence, the SEND MAIL module automatically inserts one.
8. The “Mail-server” parameter in the SIMPL+ function call to Send Mail can be an IP address, ex. “132.149.6.220” or a name “mail1.Mycompany name.com”. In case of a name, DNS will be used to resolve the name, and the control system MUST have a DNS server setup.
9. **REMINDER**: Strings in SIMPL can only be 256 characters long. But internal to SIMPL+ they can be concatenated to a total length of 65536 characters, as long as a SIMPL+ BUFFER_INPUT type is used to accumulate the strings.

E-mail Function Return Error Codes

ERROR CODE	#	DESCRIPTION
SMTP_OK	0	Success

SMTP ERRORS (NONRECOVERABLE ERRORS)

ERROR CODE	#	DESCRIPTION
SMTP_ERROR_FATAL	-1	Any non-recoverable error from the e-mail module of the firmware (for example: if “mailserver”, “from” and “to” are empty).
SMTP_ERROR_ILLEGAL_CMD	-2	General internal error.
SMTP_ERROR_CONNECT	-3	Failure to connect to the mailserver.
SMTP_ERROR_SEND	-4	Internal error while actually sending out e-mail.
SMTP_ERROR_RECV	-5	Internal error while actually receiving out e-mail.
SMTP_ERROR_NU_CONNECT	-6	Internal error while processing the send.
SMTP_ERROR_NU_BUFFERS	-7	Lack of memory buffers while processing send or receive mail. Internal error.
SMTP_ERROR_AUTHENTICATION	-8	Authentication failure.
SMTP_ERROR_AUTH_LOGIN_UNSUPPORTED	-9	CLEAR TEXT login scheme is not supported.
SMTP_INV_PARAM	-10	Bad parameters to SendMail. Must supply Server, From, and To.
SMTP_ETHER_NOT_ENABLED	-11	Ethernet not enabled. Cannot send mail.
SMTP_NO_SERVER_ADDRESS	-12	No DNS servers configured. Cannot resolve name.
SMTP_SEND_FAILURE	-13	SendMail failed.

SMTP FAILURES (RECOVERABLE ERRORS)

ERROR CODE	#	DESCRIPTION
SMTP_FAILURE_TO_RCPT_COMMAND	3	There was an error sending e-mail to the “to” recepient.
SMTP_FAILURE_CC_RCPT_COMMAND	4	There was an error sending e-mail to the “CC” recepient.
SMTP_FAILURE_DATA_COMMAND	5	There was an error sending the message body.

SendMail

Name:

SendMail

Syntax:

```
SIGNED_INTEGER SendMail( STRING Server,
STRING UserLogonName,
STRING UserLogonPassword,
STRING From,
STRING To,
STRING CC,
STRING Subject,
STRING Message       )
```

Description:

Send an e-mail message using SMTP protocol.

Parameters:

Server - Required. Specifies address of the mail server. It can either be an IP address in dot-decimal notation (ex: 192.168.16.3) or a name to be resolved with a DNS server (ex: mail.myisp.com). If a name is given, the control system must be configured with a DNS server (ADDDNS console command).   Maximum field length: 40.

UserLogonName - Optional, but if authentication is *not* required, put an empty string in its place. If the mail server requires authentication, UserLogonName indicates the user name of the sender for the mail server.   An empty string indicates that authentication is not required. Only “clear text” authentication is implemented. “Clear text” refers to the authentication method used by the mail server. If the mail server requires a higher level authentication, mail can not be sent to the mail server. Maximum field length: 254.

UserLogonPassword - Optional, but if authentication is *not* required, put an empty string in its place. If the mail server requires authentication, UserLogonPassword indicates the password of the sender for the mail server. An empty string indicates that authentication is not required. Only “clear text” authentication is implemented. “Clear text” refers to the authentication method used by the mail server. If the mail server requires a higher level authentication, mail can not be sent to the mail server. Maximum field length: 254.

From - Required. Specifies the e-mail address of the sender in the a@b.com format. Only one e-mail address is allowed. Aliases or nicknames are not supported. This argument is mandatory. Maximum field length: 242.

To - Required. Specifies the e-mail address of the recipient(s) in the a@b.com format. Multiple recipients may be specified delimited with a “;”. This argument is mandatory. Maximum field length: 65535.

CC - Optional , but put an empty string in its place to indicate that there are no recipients. Specifies the e-mail address of the carbon copy recipient(s) in the a@b.com format. Multiple recipients may be specified delimited with a “;”. Maximum field length: 65535.

Subject - Optional, but use an empty string to indicate that there is no subject. Specifies the subject of the e-mail message. Maximum field length: 989.

Message - Optional, but use an empty string to indicate that there is no message. Specifies the body of the e-mail message. An empty string indicates an empty message. Maximum field length: 65535.

Return Value:

0 if successful. Otherwise, E-mail Return Error Code is returned. Negative return error codes indicate that no part of the e-mail was sent (example: user logon password was incorrect). Positive return error codes indicate a failure (example: one or more recipient e-mail addresses was invalid), but the e-mail was still sent. In the event of more than one failure, the return error code of the first failure is returned.

Example:

```
SIGNED_INTEGER nErr;
nErr = SendMail( “192.168.16.3”,
“UserLogonName”,
“UserLogonPassword”,
“SenderEmailAddress@crestron.com”,
“RecipientEmailAddress@crestron.com”,
“ccEmailAddress@crestron.com”,
“This is the subject”,
“This is the message” );
if ( nErr < 0 )
Print( “Error sending e-mail\n” );
else
Print( “SendMail successful!\n );
```

Version:

SIMPL+ Version 3.01.xx (Pro 2 only)

Events

Events Overview

SIMPL+ is an event driven language. There are four functions which deal with activating events in a given SIMPL+ program; CHANGE, EVENT, PUSH, and RELEASE.

CHANGE

Name:

CHANGE

Syntax:

```
CHANGE <variable_name1> [, <variable_name2> ...]
{
[Local Variable Definitions]
<statements>
}
```

Description:

<variable_name> may be either a DIGITAL_INPUT, ANALOG_INPUT, or STRING_INPUT type. If it is a DIGITAL_INPUT, the statements between { and } will be executed when the input transitions from low to high or high to low. If it is an ANALOG_INPUT or STRING_INPUT, the statements between { and } will be executed whenever the variable changes. Note that for an ANALOG_INPUT or STRING_INPUT, the same value re-issued will also cause the CHANGE to activate.

When using ANALOG_INPUT, BUFFER_INPUT, DIGITAL_INPUT, or STRING_INPUT arrays, only a change in the entire array can be detected, not an individual element. Refer to “GetLastModifiedArrayIndex” on page 93 to determine which element actually changed. Use IsSignalDefined to ensure that you send data only to outputs that exist or take input from signals that exist.

When listing multiple variable names, the names can be put on the same line or broken up into several CHANGE statements for readability.

Refer to “Stacked Events” on page 80.

Example:

```
STRING_INPUT some_data$[100];
ANALOG_OUTPUT level;

CHANGE some_data$
{
    level=48;
}
```

When the STRING_INPUT changes, the ANALOG_OUTPUT level will have the value 48 put into it. If the same data comes in on some_data$, the CHANGE block is executed again.

```
ANALOG_INPUT ThingsToAdd[20];
ANALOG_OUTPUT Sum;
INTEGER I, Total;

CHANGE ThingsToAdd
{
    Total=0;
    FOR(I=0 to 20)
       if (IsSignalDefined (ThingsToAdd[I]))
            Total = Total + ThingsToAdd[I];
    Sum = Total;
}
```

In this example, an array is used to hold elements to add. When any element of the array changes, the sum is recomputed and issued on an analog output variable.

Version:

SIMPL+ Version 3.00 - local variables are allowed within CHANGE statements.

SIMPL+ Version 2.00 for ANALOG_INPUT, BUFFER_INPUT, DIGITAL_INPUT, and STRING_INPUT arrays as <variable_name>.

SIMPL+ Version 1.00 for everything else.

EVENT

Name:

EVENT

Syntax:

```
EVENT
{
    [Local Variable Definitions]
    <statements>
}
```

Description:

Executes the defined <statements> anytime one of the inputs to the SIMPL+ symbol changes. It is similar to having a CHANGE statement listed for every input, and each change is set up to execute a common block of code. Refer to “Stacked Events” on page 80.

Example:

```
ANALOG_INPUT level1, level2, level3;
STRING_INPUT extra$[2][20];
STRING_OUTPUT OUT$;

EVENT
{
OUT$=extra$[0]+extra$[1]+CHR(level1)+CHR(level2)+CHR(level3)
;
}
```

In this example, when the ANALOG_INPUTs level1, level2, level3, or level4 have any change or the STRING_INPUT array extra$ has changed, the STRING_OUTPUT OUT$ will be recomputed and reissued.

Version:

SIMPL+ Version 3.00 - Local variables are allowed within EVENT statements.

SIMPL+ Version 1.00

PUSH

Name:

PUSH

Syntax:

```
PUSH <variable_name1> [, <variable_name2> ...]
{
    [Local Variable Definitions]
    <statements>
}
```

Description:

<variable_name> is a DIGITAL_INPUT type. On the rising edge of <variable_name>, the statements between the opening { and closing } are executed.

When using DIGITAL_INPUT arrays, only a change in the entire array can be detected, not an individual element. Refer to “GetLastModifiedArrayIndex” on page 93 for a method of detecting a change to an individual element.

When listing multiple variable names, the names can be put on the same line or broken up into several PUSH statements for readability. Refer to “Stacked Events” on page 80.

Example:

```
DIGITAL_INPUT trigger;
STRING_OUTPUT output$;

PUSH trigger
{
output$ = “Hello, World!”;
}
```

In this example, when the DIGITAL_INPUT trigger transitions from low to high, the STRING_OUTPUT output$ will have the string “Hello, World!” put into it.

Version:

SIMPL+ Version 3.00 - local variables are allowed within PUSH statements.

SIMPL+ Version 2.00 for DIGITAL_INPUT arrays as <variable_name>.

SIMPL+ Version 1.00 for everything else.

Release

Name:

RELEASE

Syntax:

```
RELEASE <variable_name1> [, <variable_name2> ...]
{
[Local Variable Definitions]
<statements>
}
```

Description:

<variable_name> is a DIGITAL_INPUT type. On the trailing edge of <variable_name>, the statements between the opening { and closing } are executed.

When using DIGITAL_INPUT arrays, only a change in the entire array can be detected, not an individual element. Refer to “GetLastModifiedArrayIndex” on page 93 for a method of detecting a change to an individual element.

When listing multiple variable names, the names can be put on the same line or broken up into several RELEASE statements for readability. Refer to “Stacked Events” on page 80.

Example:

```
DIGITAL_INPUT trigger;
STRING_OUTPUT output$;

RELEASE trigger
{
    output$ = “Hello, World!”;
}
```

In this example, when the DIGITAL_INPUT trigger transitions from high to low, the STRING_OUTPUT output$ will have the string “Hello, World!” put into it.

Version:

SIMPL+ Version 3.00 - local variables are allowed within RELEASE statements.

SIMPL+ Version 2.00 for DIGITAL_INPUT arrays as <variable_name>.

SIMPL+ Version 1.00 for everything else.

Stacked Events

Stacked Events refers to multiple CHANGE, PUSH or RELEASE functions followed by a single block of code (complex statement).

NOTE: Only CHANGE, PUSH, or RELEASE functions are used in stacked events. If necessary, refer to the descriptions of each function for details.

NOTE: An input signal can be used in more than one event function. The order execution is as follows:

The order for a PUSH:

PUSH statements in the order they appear in the source.

CHANGE statements in the order they appear in the source EVENT statement

The order for a RELEASE:

RELEASE statements in the order they appear in the source.

CHANGE statements in the order they appear in the source EVENT statement

A typical event statement may appear as:

```
PUSH var1, var2
{
    // code
}
```

SIMPL+ allows event stacking, which allows a block of code to be called from different CHANGE, PUSH, or RELEASE statements. An example is:

```
STRING_INPUT A$[100];
DIGITAL_INPUT IN1, IN2, IN3, IN4;
ANALOG_INPUT LEVEL;
ANALOG_INPUT PRESETS[5];
PUSH IN1
PUSH IN2
CHANGE IN3, LEVEL, A$, PRESETS
RELEASE IN3, IN4
{
    // code
}
```

This allows one piece of code to execute from many different types of event statements.

Expressions & Statements

An expression consists of operators and operands.

i.e.,

```
5 * 6
```

or

```
(VAL1 + 5) / 30
```

or

```
(26 + BYTE(THESTRING$,1)) MOD Z = 25
```

Statements consist of function calls, expressions, assignments, or other instructions. There are two types of statements, Simple and Complex.

A simple statement ends with a semicolon (;). Examples of simple statements are:

```
X = Z/10; // Simple assignment statement using
// operators.
PRINT(“Hello, World!\n”); // Simple statement using a function
// call.
CHECKSUM = ATOI(Z$) + 5; // Simple assignment statement using
// a function call and operators.
```

A complex statement is a collection of simple statements that start with '{' and end with '}'. An example would be:

```
{ // Start of a complex statement
X = Z/10; // Simple assignment statement
// using operators.
PRINT(“Hello, World!\n”); // Simple statement using a
// function call.
CHECKSUM = ATOI(Z$) + 5;  // Simple assignment statement
// using a function call and // operators.
} // End of a Complex statement
```

Looping Constructs

Looping Constructs Overview

Loops are used to perform a section of code zero or more times in a row in a given SIMPL+ program. The body of the loop can consist of statements, expressions, function calls, or other loops.

DO - UNTIL

Name:

DO - UNTIL

Syntax:

```
DO
[{]
    <statements>
[}] UNTIL (<expression>);
```

Description:

This loop performs a set of <statements> at least one time and will terminate when <expression> evaluates to true. If only one statement is present in the body of the loop, then the { and } characters are not required, but may be used. If more than one statement is present in the loop body, then the { and } characters are mandatory. Note that <expression> is evaluated each time through the loop.

Example:

```
INTEGER X;
X=0;
DO
{
X = X + 1;
PRINT(“X = %d\n”, X);
}
UNTIL (X = 25);
```

In this example, the loop will execute 25 times. The PRINT function will show the value of X after it is incremented to the computer port of the control system.

Version:

SIMPL+ Version 1.00

FOR

Name:

FOR

Syntax:

```
FOR (<variable> = <start_expression> TO <end_expression>
[STEP <step_expression>])
[{]
<statements>
[}]
```

Description:

This loop executes the <statements> while <variable> iterates from the value of <start_expression> to the value of <end_expression>. The variable is incremented by <step_expression> at the end of the loop, if STEP is specified, else it is incremented by 1. The <step_expression> can be negative which will result in the loop counting down. If only one statement is present in the body of the loop, then the { and } characters are not required, but may be used. If more than one statement is present in the loop body, then the { and } characters are mandatory. Note that <start_expression> and <end_expression> are evaluated once before the loop starts and are not re-evaluated during the execution of the loop. If it is defined, <step_expression> is evaluated each pass through the loop, so <step_expression> may be modified during execution of the loop.

In the 2-Series control systems, the <step_expression> cannot change its sign during the execution of the loop. That is, if it is initially a positive number, then it is assumed if it will always count up. If it is negative, it will always count down.

NOTE: If <variable> is set to a value greater than the <end_expression> within the body of the FOR loop, the FOR loop will exit when it reaches the end.

At the end of the loop, the loop index has the value of <end_expression> + 1 (unless the loop index was modified in the body of the loop).

The comparisons are based on signed numbers, the maximum loop size for a step of one would be from 1 to 32767. If larger indices are needed, for example, from 1 to 60000 a DO-UNTIL or WHILE loop could be used.

Example:

```
STRING_INPUT IN$[100];
INTEGER X;
FOR (X = 1 TO LEN(IN$))
{
    PRINT(“Character %d of String %s is %s\n”, X, IN$,
    MID(IN$, X, 1));
}
```

In this example, the loop will iterate through each character of a string and print out the string and its position in the original string.

Version:

SIMPL+ Version 1.00

WHILE

Name:

WHILE

Syntax:

```
WHILE(<expression>)
[{]
    <statements>
[}]
```

Description:

This loop performs a set of <statements> as long as <expression> does not evaluate to zero.

If only one statement is present in the body of the loop, then the { and } characters are not required, but may be used. If more than one statement is present in the loop body, then the { and } characters are mandatory. Note that depending on <expression>, the body of the loop may never be executed. Note that <expression> is evaluated at the beginning of each time through the loop.

Example:

```
INTEGER X;
X=0;
WHILE(X < 25)
{
X = X + 1;
PRINT(“X = %d\n”, X);
}
```

In this example, the loop will execute 25 times. The PRINT function will show the value of X after it is incremented to the computer port of the control system.

Version:

SIMPL+ Version 1.00

Branching & Decision Constructs

BREAK

Name:

BREAK

Syntax:

```
BREAK;
```

Description:

Terminates the innermost DO-UNTIL, FOR, or WHILE loop before the exit condition is met. Execution resumes after the end of the bop.

Example:

```
INTEGER X;
ANALOG_INPUT Y;
X=0;
WHILE(X<25)
{
    IF(Y = 69)
    BREAK;
    X = X + 1;
    PRINT(“X=%d\n”, X);
}
```

In this example, the WHILE loop will terminate if the ANALOG_INPUT Y equals the value of 69. Otherwise, the loop will exit via the normal termination condition.

Version:

SIMPL+ Version 1.00

CSWITCH

Name:

CSWITCH

Syntax:

```
CSWITCH (<expression>)
{
CASE (<unique integer constant>):
[{]
<statements1>
[break;]
[}]

CASE (<unique integer constant >):
[{]
    <statements2>
[break;]
[}]

[DEFAULT:
[{]
    <statements>
[break;]
[}]
}
```

NOTE: In SIMPL+ v3.01.00 and later, the 'break' statement is required to terminate the case statement block that it resides within. If no 'break' statement exists, the program will continuing executing to the next case statement block or default statement block.

NOTE: Many CASE statements may be used in the body of the CSWITCH.

Description:

CSWITCH is a more direct method of writing a complex IF-ELSE-IF statement. In the CSWITCH, if <expression> is equal to a CASE’s constant, then the statement block for that CASE value is executed. This same method would apply to as many CASE statements as are listed in the body of the CSWITCH. Note that if any of the

<statements> blocks are only a single statement, the { and } characters on the CASE may be omitted. If no condition is met in the CASE statements, the DEFAULT case, if specified, is used.

CSWITCH has the restriction that the case statement only contains unique integer constants. CSWITCH differs from SWITCH in that the operating system is able to evaluate and execute the CSWITCH statement faster. Therefore, you should use CSWITCH in place of SWITCH whenever unique constants are being evaluated.

Example:

```
ANALOG_INPUT AIN;
INTEGER X;
CSWITCH( AIN )
{
CASE (2):
{
X = 0;
break; // terminate this case statement block
}
    CASE (3):
{
    X = AIN;
    // continue executing to next case statement block ==>
    case(5)
}
    CASE (5):
{
    X = X + AIN + 1;
    break;
}
    DEFAULT:
{
    PRINT("Unknown command %d!\n", AIN);
    break;
}
}
```

In this example, if the value of AIN is 2, X is set equal to 0. If AIN is 3, X is set AIN + AIN + 1. If AIN is 5, X is set equal to AIN+1. If AIN is any other value, an error message is printed.

Version:

SIMPL+ Version 3.00.05

Control System

2-Series Only

IF - ELSE

Name:

IF - ELSE

Syntax:

```
IF ( <expression>)
[{]
<statements>
[}]
    [ELSE]
[{]
    <statements>
[}]]
```

Since <statements> can be an IF construct, you can string out a series of IF-ELSE-IF statements of the form:

```
IF (<expression>)
[{]
    <statements>
[}]
    [ELSE] IF (<expression>)
[{]
<statements>
[}]]
```

NOTE: A final ELSE may be used to express default handling if none of the previous conditions were met.

```
IF (<expression>)
[{]
    <statements>
[}]
    [ELSE] IF (<expression>)
[{]
    <statements>
[}]
    [ELSE]
[{]
    <statements>
[}]
```

Description:

Executes a piece of code only if its associated <expression> evaluates to true. Many expressions can be tested if the IF-ELSE-IF construct is used. Note that only one <statements> block in an IF-ELSE or IF-ELSE-IF construct is executed. In any section of the construct, if <statements> is only a single statement, then the { and } characters may be omitted.

Example:

```
STRING_INPUT IN$[100];
STRING Y$[100];
INTEGER X;
IF (IN$ = “STRING1”)
{
    X=5;
    Y$ = IN$;
}
    ELSE
{
    X=6;
    Y$ = ““;
}
```

In this example, if IN$ is equal to STRING1, then the first two statements are executed. If IN$ is a different value, then the second groups of statements are evaluated. A more complex IF-ELSE-IF construct appears as:

```
IF (IN$ = “STRING1”)
{
X=5;
Y$ = IN$;
}
ELSE IF (IN$=”STRING2”)
{
X=6;
Y$ = ““;
}
ELSE
{
X = 7;
Y$ = “ZZZ”;
}
```

Version:

SIMPL+ Version 1.00

SWITCH

Name:

SWITCH

Syntax:

```
SWITCH (<expression>)
{
    CASE (<expression1>):
[{]
    <statements1>
[}]
    CASE (<expression2>):
[{]
    <statements2>
[}]
    [DEFAULT:
[{]
    <statements>
[}]
}
```

NOTE: Many CASE statements may be used in the body of the SWITCH.

Description:

SWITCH is a more direct method of writing a complex IF-ELSE-IF statement. In the SWITCH, if <expression> is equal to <expression1>, <statements1> is executed. If <expression> is equal to <expression2>, <statements2> is executed. This same method would apply to as many CASE statements as are listed in the body of the SWITCH. Note that if any of the <statements> blocks are only a single statement, the { and } characters on the CASE may be omitted.

SWITCH has the restriction that the expressions may not be STRING expressions, they can only be INTEGER type expressions. SWITCH may only have up to 32 CASE statements in SIMPL+ Version 1.00. If more are used, a “FULL STACK” error results at the time of uploading the module to the control system. Version 2.00 has no restriction.

When a SWITCH is evaluated, the first matching CASE is used. If another CASE (or more) would have matched, only the first one is used. If no condition is met in the CASE statements, the DEFAULT case is used if specified.

Example:

```
ANALOG_INPUT AIN;
INTEGER X;
SWITCH(AIN)
{
    CASE (2):
{
    X = 0;
}
    CASE (3):
{
    X = AIN;
}
    CASE (5):
{
    X = AIN + 1;
}
    DEFAULT:
    PRINT(“Unknown command %d!\n”, AIN);
}
```

In this example, if the value of AIN is 2, X is set equal to 0. If AIN is 3, X is set equal to AIN. If AIN is 5, X is set equal to AIN+1. If AIN is any other value, an error message is printed.

Version:

SIMPL+ Version 2.00 - removes CASE restriction

SIMPL+ Version 1.00 with 32 CASE statements maximum

Array Operations

Array Operations Overview

Array Operations functions are used to perform generalized operations on arrays, such as getting bounds and setting the elements of an array to a known value in a given SIMPL+ program.

GetLastModifiedArrayIndex

Name:

GetLastModifiedArrayIndex

Syntax:

```
INTEGER GetLastModifiedArrayIndex ();
```

Description:

Determines the specific index number of an input list array that has changed.

ANALOG_INPUT, BUFFER_INPUT, DIGITAL_INPUT, and STRING_INPUT arrays are subject to be used in CHANGE, PUSH, and RELEASE statements, but only the overall array can be specified in the statement, not an individual element. In order to find out what element has been modified (and hence caused the activation of the CHANGE, PUSH, or RELEASE), GETLASTMODIFIEDARRAYINDEX is used.

NOTE: To use GETLASTMODIFIEDARRAYINDEX, only one array may be used in a single CHANGE, PUSH, or RELEASE statement. If more than one element of the array changes at the same time, multiple events are run. For example, if D[10] is a DIGITAL_INPUT array that is subject to a PUSH event, and D[1] and D[2] change at the same time, the PUSH is first run where D[1] changes and GETLASTMODIFIEDARRAYINDEX returns 1, then the PUSH is run again where D[2] changes and GETLASTMODIFIEDARRAYINDEX returns 2.

NOTE: Using GetLastModifiedArrayIndex OUTSIDE of an event (PUSH, RELEASE, CHANGE or EVENT) may return an index to an ambiguous signal if more than one input array is declared within the program. Therefore, do not use this function if more than one input signal array is declared within the program, unless you use it within one of the event statements.

Return Value:

The element of the array that has changed.

Example 1 - Correct Use:

```
DIGITAL_INPUT LIGHT_SCENES[10], MORE_LIGHT_SCENES[10};
DIGITAL_OUTPUT INTERLOCKED_LIGHT_SCENES[10];
INTEGER I;
PUSH LIGHT_SCENES
{
    FOR(I=1 to 10)
    INTERLOCKED_LIGHT_SCENES[I] = 0;
    ProcessLogic();
    INTERLOCKED_LIGHT_SCENES[GetLastModifiedArrayIndex()] =
    1;
}
```

Example 2 - Incorrect Use:

```
DIGITAL_INPUT LIGHT_SCENES[10];
DIGITAL_OUTPUT INTERLOCKED_LIGHT_SCENES[10];
INTEGER I;
PUSH LIGHT_SCENES,MORE_LIGHT_SCENES
{//this PUSH statement will be called twice (once for
LIGHT_SCENES and once for MORE_LIGHT_SCENES)
    FOR(I=1 to 10)
    INTERLOCKED_LIGHT_SCENES[I]=0
    ProcessLogic();
    INTERLOCKED_LIGHT_SCENES[GetLastModifiedArrayIndex()] =
    1;
}
```

In this example, when one input element changes, all the output elements are set to 0 and then the output level corresponding to the changed input level is set to 1. This mimics the functionality of the Interlock symbol in SIMPL.

Version:

SIMPL+ Version 2.00

GetNumArrayCols

Name:

GetNumArrayCols

Syntax:

```
INTEGER GetNumArrayCols(STRING | INTEGER ARRAY_NAME);
```

Description:

Finds the number of columns in a two-dimensional array or the size of the array for a one-dimensional array.

Parameters:

ARRAY_NAME is the array as determined by the size.

Return Value:

For the data types in the table after this paragraph, the return value of GetNumArrayCols is shown.

DATA TYPE	RETURN VALUE
ANALOG INPUT X [size]	Size
ANALOG INPUT X [size]	Size
DIGITAL INPUT X [size]	Size
DIGITAL OUTPUT X [size]	Size
STRING INPUT X [size]	Chars
STRING INPUT X [size] [chars]	Chars
STRING OUTPUT X [size]	Size
STRING X [chars]	Chars
STRING X [size] [chars]	Chars
INTEGER X [size]	Size
INTEGER X [size 1] [size 2]	Size2
SIGNED_INTEGER X [size]	Size
SIGNED_INTEGER X [size1] [size2]	Size2
SIGNED_LONG_INTEGER X [size]	Size
SIGNED_LONG_INTEGER X [size 1] [size 2]	Size2
BUFFER INPUT X [chars]	Chars
BUFFER INPUT X [size] [chars]	Chars

Example:

```
DIGITAL_INPUT TEST;
INTEGER My_Array[10][20];

PUSH TEST
{
    PRINT(“Columns = %d\n”, GetNumArrayCols(My_Array));
}
```

In this example, Columns = 20 will be printed.

Version:

SIMPL+ Version 2.00

GetNumArrayRows

Name:

GetNumArrayRows

Syntax:

```
INTEGER GetNumArrayRows(STRING | INTEGER ARRAY_NAME);
```

Description:

Returns the number of rows for two-dimensional arrays.

One-dimensional arrays return 0.

Parameters:

ARRAY_NAME is the array name as determined by the size.

Return Value:

For the data types in the table after this paragraph, the return value of GetNumArrayRows is shown.

DATA TYPE	RETURN VALUE
INTEGER X[size1][size2]	Size 1
SIGNED_INTEGER X[size1][size2]	Size 1
SIGNED_LONG_INTEGER X[size1][size2]	Size 1
STRING X[chars]	Size
STRING_INPUT X[size][chars]	Size
BUFFER_INPUT X[size][chars]	Size

Example:

```
DIGITAL_INPUT TEST;
INTEGER My_Array[10][20];

PUSH TEST
{
    PRINT(“Rows = %d\n”, GetNumArrayRows(My_Array));
}
```

In this example, Rows = 10 will be printed.

Version:

SIMPL+ Version 2.00

SetArray

Name:

SetArray

Syntax:

```
SetArray (ARRAY_NAME, INTEGER | STRING INIT_VALUE);
```

Description:

Sets every element of ARRAY_NAME to the INIT_VALUE.

Parameters:

ARRAY_NAME is the name of the array to be initialized. It may be any array type.

The INIT_VALUE may be a INTEGER or STRING. The following chart shows the various combinations of ARRAY_NAME types and INIT_VALUE types:

ARRAY_NAME TYPE	INIT_VALUE TYPE	MEANING
INTEGER, SIGNED_INTEGER	INTEGER	Every element of ARRAY_NAME is set to the INTEGER value INIT_VALUE.
INTEGER, SIGNED_INTEGER	STRING	Each integer in ARRAY_NAME is initialized to ATOI (INIT_VALUE).
LONG, SIGNED_LONG_INTEGER	INTEGER	Every element of ARRAT_NAME is set to the LONG value INIT_VALUE.
LONG, SIGNED_LONG_INTEGER	STRING	Each integer in ARRAY_NAME is initialized to ATOI (LONG_NAME)
STRING	INTEGER	Each string in ARRAY_NAME is initialized to CHR (INIT_VALUE).
STRING	STRING	Each string in ARRAY_NAME is set equal to INIT_VALUE. IF INIT_VALUE is longer than the maximum size allowed in the array, it is truncated.

NOTE: When working with DIGITAL_OUPUT arrays, if the INIT_VALUE evaluates to 0, the digital signals are set low. For any non-zero value, the outputs are set high.

Return Value:

None.

Example:

```
DIGITAL_INPUT InitializeArrays;
INTEGER Levels[10];
STRING Names[5][5];

PUSH InitializeArrays
{
    SetArray(Levels, 3);
    SetArray(Levels, “3”);
    SetArray(Names, “xyz”);
    SetArray(Names, 0x41);
}
```

The first line initializes all elements of the integer array Levels to contain the integer 3.

The second line attempts to initialize the elements of the integer array Levels with a string value - an ATOI is done on the “3”, which returns a 3, so that the end result is the same as the first line.

The third line initializes all elements of the elements of the string array Names to contain the string value “xyz”.

The fourth line attempts to initialize the elements of the string array Names with an integer value - a CHR is done on the 0x41, which returns the string “A”, so that the end result has all elements of the string array Names containing the string “A”.

Version:

SIMPL+ Version 2.00

Bit & Byte Functions

Bit & Byte Functions Overview

These functions perform bit and byte masking operations in a given SIMPL+ program.

Bit

Name:

Bit

Syntax:

```
INTEGER Bit(STRING SOURCE, INTEGER SOURCE_BYTE, INTEGER
BIT_IN_BYTE);
```

Description:

Determine the state of a specified bit in a particular byte of a given string.

Parameters:

SOURCE contains a STRING in which a bit of one byte is to be examined. Each character in SOURCE is considered one byte.

SOURCE_BYTE references a character in the SOURCE string. The leftmost character in SOURCE is considered 1.

BIT_IN_BYTE specifies which bit in the SOURCE_BYTE of SOURCE is to be examined. BIT_IN_BYTE starts at position 0 (least significant or rightmost bit of the byte). 7 is the most significant or leftmost bit of the byte.

Return Value:

Returns 0 or 1 for a valid bit position. Illegal bit references will return 65535. It is illegal if SOURCE_BYTE is 0 or greater than the length of the SOURCE string. Note that it is legal to specify a bit beyond 7. This will reference a bit in another byte. In this way, a source string can be used as a set of packed bit flags. The algorithm for determining which bit in which byte is set when BIT_IN_BYTE is greater than 7 is as follows:

```
Actual Byte in SOURCE = (BIT_IN_BYTE / 8) + SOURCE_BYTE
Actual Bit in Actual Byte = (BIT_IN_BYTE MOD 8)
```

Applying this to BIT (“abc”,1,16) will reference bit 0 of byte 3 (the least significant bit of byte “c” in SOURCE).

Example:

This example takes an input string and creates an output string containing the elements of the input string that do not have the most significant bit (bit 7) set.

```
STRING_INPUT SOURCE$[100];
STRING_OUTPUT OUT$;
STRING TEMP$[100];
INTEGER I;

CHANGE SOURCE$
{
FOR(I = 1 to LEN(SOURCE$))
{
IF(BIT(SOURCE$, I, 7) = 0)
{
    MAKESTRING(TEMP$, “%s%s”, TEMP$, MID(SOURCE$, I, 1));
}
}
OUT$ = TEMP$;
}
```

Version:

SIMPL+ Version 1.00

Byte

Name:

Byte

Syntax:

```
INTEGER Byte (STRING SOURCE, INTEGER SOURCE_BYTE);
```

Description:

Returns the integer equivalent of the byte at position SOURCE_BYTE within a SOURCE string.

Parameters:

SOURCE is a STRING of characters. Each character in SOURCE is considered one byte.

SOURCE_BYTE references a character in the SOURCE string. The leftmost character in SOURCE is considered 1.

Return Value:

An integer containing the ASCII numeric value of the byte at position SOURCE_BYTE in the string SOURCE. If SOURCE_BYTE is greater than the length of the SOURCE string or is 0, 65535 is returned.

Example:

This piece of code will examine an input string to make sure that it starts with STX character (02). From there, it will test the second byte and process different command types accordingly.

```
STRING_INPUT IN$[100];
CHANGE IN$
{
    IF(BYTE(IN$,1) = 02)
{
    SWITCH(BYTE(IN$,2))
{
    CASE (65):
{
    // Process Command type 65 (A) here.
}
    CASE (66):
{
    // Process Command type 66 (B) here.
}
}
```

Version:

SIMPL+ Version 1.00

High

Name:

High

Syntax:

```
INTEGER High(INTEGER VALUE);
```

Description:

Returns the upper (most significant) 8-bits of an Integer.

Parameters:

VALUE is an integer containing the value of the most significant byte.

Return Value:

The upper 8-bits of the passed value.

Example:

```
ANALOG_INPUT VALUE;

CHANGE VALUE
{
    PRINT(“The upper byte of %X is %X\n”, VALUE, HIGH(VALUE))
}
```

This will print the input value and the upper 8-bits of the value in hexadecimal. For example, if VALUE is 0x1234, then the output is:

```
The upper byte of 1234 is 12.
```

Version:

SIMPL+ Version 1.00

Low

Name:

Low

Syntax:

```
INTEGER Low(INTEGER VALUE)
```

Description:

Returns the lower (least significant) 8-bits of an Integer.

Parameters:

VALUE is an integer containing the value of the least significant byte.

Return Value:

The lower (least significant) 8-bits of the passed value.

Example:

```
ANALOG_INPUT VALUE;

CHANGE VALUE
{
    PRINT("The lower byte of %X is %X\n", VALUE, LOW(VALUE));
}
```

This will print the input value and the lower 8-bits of the value in hexadecimal. For example, if VALUE is 0x1234, then the output is:

```
The lower byte of 1234 is 34.
```

Version:

SIMPL+ Version 1.00

RotateLeft

Name:

RotateLeft

Syntax:

```
INTEGER RotateLeft( INTEGER X, INTEGER Y );
```

Description:

Rotate X to the left (more significant direction) by Y bits; full 16 bits used. Same as {{ operator. See RotateRight on page 106.

Parameters:

X is the INTEGER to have bits rotated

Y is the amount of bits to rotate

Return Value:

An INTEGER containing the result of the rotated bits.

Example:

```
INTEGER X, Y, result;
result = RotateLeft( X, Y );
```

Version:

SIMPL+ Version 3.01.06

RotateRight

Name:

RotateRight

Syntax:

```
INTEGER RotateRight( INTEGER X, INTEGER Y );
```

Description:

Rotate X to the right by Y bits; full 16 bits used. Same as }} operator. e.g.: Each bit takes the value of the bit that is Y bits more significant than it is. The most significant bit(s) are set from the least significant bits.

Parameters:

X is the INTEGER to have bits rotated

Y is the amount of bits to rotate

Return Value:

An INTEGER containing the result of the rotated bits.

Example:

```
INTEGER X, Y, result;
result = RotateRight( X, Y );
```

If X = 0x1234 and Y is 1 then result is 0x091A

Version:

SIMPL+ Version 3.01.06

RotateLeftLong

Name:

RotateLeftLong

Syntax:

```
LONG_INTEGER RotateLeftLong( LONG_INTEGER X, INTEGER Y
);
```

Description:

Rotate X to the left by Y bits; full 32 bits used.

Parameters:

X is the LONG_INTEGER to have bits rotated

Y is the amount of bits to rotate

Return Value:

A LONG_INTEGER containing the result of the rotated bits.

Example:

```
LONG_INTEGER X, Y, result;
result = RotateleftLong( X, Y );
```

Version:

SIMPL+ Version 3.01.06

RotateRightLong

Name:

RotateRightLong

Syntax:

```
LONG_INTEGER RotateRightLong( LONG_INTEGER X, INTEGER Y
);
```

Description:

Rotate X to the right by Y bits; full 32 bits used.

Parameters:

X is the LONG_INTEGER to have bits rotated

Y is the amount of bits to rotate

Return Value:

A LONG_INTEGER containing the result of the rotated bits.

Example:

```
LONG_INTEGER X, Y, result;
result = RotateRightLong( X, Y );
```

Version:

SIMPL+ Version 3.01.06

Data Conversion Functions

Data Conversion Functions Overview

These functions take one form of data (integer or string) and convert it to the opposite type in a given SIMPL+ program. Usually, these functions are for converting number stored in strings to integers, or for converting numbers stored in integers to strings.

Atoi

Name:

Atoi

Syntax:

```
INTEGER Atoi(STRING SOURCE);
```

Description:

Converts a STRING to an INTEGER value. The conversion looks for the first valid character (0-9), and then reads until it finds the first invalid character. The resulting string of valid characters is then converted. The “-” is ignored, hence the output is an unsigned number [i.e., ATOI(“-1”) would yield 1 as the output]. If the value exceeds 65535, the value is undefined. If no valid value to convert is found, 0 is returned.

Parameters:

SOURCE is a string containing characters that range from 0 to 9 to be converted.

Return Value:

An integer representing the given string value. Example:

```
STRING_INPUT IN$[100];
INTEGER VAL;

CHANGE IN$
{
VAL = ATOI(IN$);
PRINT(“Value of %s after ATOI is %d\n”, IN$, VAL);
}
```

For example, if IN$ is “abc1234xyz”, then VAL will hold the integer 1234. If IN$ is “-50”, then VAL will hold the integer 50.

Version:

SIMPL+ Version 1.00

Atol

Name:

Atol

Syntax:

```
LONG_INTEGER Atol(STRING SOURCE);
```

Description:

Converts a STRING to an LONG_INTEGER value. The conversion looks for the first valid character (0-9), and then reads until it finds the first invalid character. The resulting string of valid characters is then converted. The “-” is ignored, hence the output is an unsigned number [i.e., ATOL(“-1”) would yield 1 as the output]. If no valid value to convert is found, 0 is returned. If the integer exceeds 32 bits, the value returned is undefined.

Parameters:

SOURCE is a string containing characters that range from 0 to 9 to be converted.

Return Value:

An integer representing the given string value. Example:

```
STRING_INPUT IN$[100];
LONG_INTEGER VAL;

CHANGE IN$
{
    VAL = ATOL(IN$);
    PRINT(“Value of %s after ATOL is %ld\n”, IN$, VAL);
}
```

For example, if IN$ is “abc1234xyz”, then VAL will hold the number 1234. If IN$ is “-50”, then VAL will hold the number 50.

Version:

SIMPL+ Version 3.00.02

Control System

2-Series Only

Chr

Name:

Chr

Syntax:

```
STRING Chr(INTEGER CODE);
```

Description:

Takes the integer value specified and returns the corresponding ASCII character as a one-byte string.

Parameters:

CODE contains a number from 0 to 255 to be converted into an ASCII string.

Return Value:

A string representing the code. If CODE is greater than 255, lower 8-bits of CODE are used in the computation.

Example:

```
STRING_OUTPUT Code$;
ANALOG_INPUT VALUE;

CHANGE VALUE
{
    Code$ = CHR(VALUE);
    PRINT(“Code = %s\n”, Code$);
}
```

In this example, if VALUE was equal to 72, the output would be Code = H.

Version:

SIMPL+ Version 1.00

ItoA

Name:

ItoA

Syntax:

```
STRING ItoA(INTEGER CODE);
```

Description:

Takes the value in CODE and creates a string containing the string equivalent of that integer. The output string does not contain leading zeros.

Parameters:

CODE contains a number from 0 to 65535 to be converted into a string. CODE is treated as an unsigned number.

Return Value:

A string representing the code. If CODE is greater than 65535, lower 16-bits of CODE are used in the computation.

Note that the following two statements are equivalent:

```
out$ = itoa(CODE);
makestring(out$, “%d”, CODE);
```

Example:

```
STRING_OUTPUT Code$;
ANALOG_INPUT VALUE;

CHANGE VALUE
{
    Code$ = ITOA(VALUE);
    PRINT(“Code = %s\n”, Code$);
}
```

For example, if VALUE was equal to 25, Code$ would contain the string “25”.

Version:

SIMPL+ Version 1.00

ItoHex

Name:

ItoHex

Syntax:

```
STRING ITOHEX(INTEGER CODE);
```

Description:

Takes the value in CODE and creates a string containing the hexadecimal equivalent. The output string does not contain leading zeros and is expressed in uppercase.

Parameters:

CODE contains a number from 0 to 65535 to be converted into a hexadecimal string. CODE is treated as an unsigned number.

Return Value:

A string representing the code. If CODE is greater than 65535, lower 16-bits of CODE are used in the computation.

Note that the following two statements are equivalent:

```
out$ = itohex(CODE);
makestring(out$, “%X”, CODE);
```

Example:

```
STRING_OUTPUT Code$;
ANALOG_INPUT VALUE;

CHANGE VALUE
{
    Code$ = ITOHEX(VALUE);
    PRINT(“Code = %s\n”, Code$);
}
```

For example, if VALUE contained the integer 90, Code$ would contain the string “5A”.

Version:

SIMPL+ Version 1.00

LtoA

Name:

LtoA

Syntax:

```
STRING LtoA(LONG_INTEGER CODE);
```

Description:

Takes the value in CODE and creates a string containing the string equivalent of that LONG_INTEGER. The output string does not contain leading zeros.

Parameters:

CODE contains a number from 0 to 2147483647 to be converted into a string. CODE is treated as an unsigned number.

Return Value:

A string representing the code.

Note that the following two statements are equivalent:

```
out$ = ltoa(CODE);
makestring(out$, “%ld”, CODE);
```

Example:

```
STRING_OUTPUT Code$;
LONG_INTEGER VALUE;

CHANGE VALUE
{
    Code$ = LTOA(VALUE);
    PRINT(“Code = %s\n”, Code$);
}
```

For example, if VALUE was equal to 25, Code$ would contain the string “25”.

Version:

SIMPL+ Version 3.00.07

Control System

2-Series Only

LtoHex

Name:

LtoHex

Syntax:

```
STRING LTOHEX(LONG_INTEGER CODE);
```

Description:

Takes the value in CODE and creates a string containing the hexadecimal equivalent. The output string does not contain leading zeros and is expressed in uppercase.

Parameters:

CODE contains a number from 0 to 2147483647 to be converted into a hexadecimal string. CODE is treated as an unsigned number.

Return Value:

A string representing the code.

Note that the following two statements are equivalent:

```
out$ = ltohex(CODE);
makestring(out$, “%X”, CODE);
```

Example:

```
STRING_OUTPUT Code$;
LONG_INTEGER VALUE;

CHANGE VALUE
{
    Code$ = LTOHEX(VALUE);
    PRINT(“Code = %s\n”, Code$);
}
```

For example, if VALUE contained the integer 90, Code$ would contain the string “5A”.

Version:

SIMPL+ Version 3.00.07

Control System

2-Series Only

File Functions

File Functions Overview

File Functions perform file handle access from SIMPL+. Because of the overhead involved with maintaining current directory and file positions, there are restrictions on file I/O. Each SIMPL+ thread (main loop or event handler) that requires file operations must first identify itself with the operating system. This is done with the function, StartFileOperations. Before terminating the thread, the function EndFileOperations must be called. Files cannot be opened across threads. In other words, you cannot open a file in one thread (function main say) and then access the file with the returned file handle in another (say an event handler). This is to prevent two events from writing to different parts of a file. This means that you should open, access and then close a file within the same thread. For example, a program might be structured as follows:

```
STRING sBuf[1000];
SIGNED_INTEGER nFileHandle;
CHANGE input
{
    SIGNED_INTEGER nNumRead;
    StartFileOperations();
    nFileHandle = FileOpen ( “\\CF0\\MyFile”, _O_RDONLY );
    if ( nFileHandle >= 0 )
    {
       nNumRead=FileRead( nFileHandle, sBuf, 500 );
       if(nNumRead<0)
          Print (“Read Error\n”);
       FileClose( nFileHandle );
    }
    EndFileOperations();
}
/*******************************************************
Main()
Uncomment and place one-time startup code here
(This code will get called when the system starts up)
*******************************************************/
Function Main()
{
SIGNED_INTEGER nNumWritten;
    StartFileOperations();
    nFileHandle = FileOpen ( “\\CF0\\MyFile”, _O_WRONLY );
    if ( nFileHandle >= 0 )
    {
       sBuf = “Hello World!”;
       nNumWritten=FileWrite( nFileHandle, sBuf, 500 );
```

```
            if(nNumWritten<0) Print (“WriteError”);
            FileClose( nFileHandle );
        }
        EndFileOperations();
    }
```

File Function Return Error Codes

KEYWORD	VALUE	FUNCTION
FILE_BAD_USER	-3000	Calling task is not a file user. Use StartFileOperations() first.
FILE_NO_DISK	-3004	Disk is removed.
FILE_LONGPATH	-3017	Path or directory name too long.
FILE_INVNAME	-3018	Path or filename includes invalid character.
FILE_PEMFILE	-3019	No file descriptors available (Too many files open).
FILE_BADFILE	-3020	Invalid file descriptor.
FILE_ACCES	-3021	Attempt to open a read-only file or special (directory).
FILE_NOSPC	-3022	No space to create file in this disk.
FILE_SHARE	-3023	The access conflicts from multiple tasks to a specific file.
FILE_NOFILE	-3024	File not found.
FILE_EXIST	-3025	Exclusive access requested, but file already exists.
FILE_NVALFP	-3026	Seek to negative file pointer.
FILE_MAXFILE_SIZE	-3027	Over the maximum file size.
FILE_NOEMPTY	-3028	Directory is not empty.
FILE_INVPARM	-3029	Invalid Flag/Mode is specified.
FILE_INVPARCMB	-3030	Invalid Flag/Mode combination.
FILE_NO_MEMORY	-3031	Can't allocate internal buffer.
FILE_NO_BLOCK	-3032	No block buffer available.
FILE_NO_FINODE	-3033	No FINODE buffer available.
FILE_NO_DROBJ	-3034	No DROBJ buffer available.
FILE_IO_ERROR	-3035	Driver I/O function routine returned.
FILE_INTERNAL	-3036	Internal error.

Reading and Writing Data to a File

Reading and writing data to a file that is moved from one kind of a system to another has special programming considerations because it will likely be written on one kind of system, e.g. a PC and read on another, e.g. a Crestron control system, or vice versa. Most programmers are used to writing programs that are both written by a PC and read by a PC.

The best way to write to a file that must be transferred between systems is to write pure ASCII text and use the FileRead/FileWrite routines. If you must write binary data as binary, e.g. structures, integers, strings, arrays, please read and consider the following.

Different kinds of systems store their internal data structures with various padding bytes and lengths, that are not always apparent. For example, a structure declared like this:

```
STRUCTURE
{
    STRING s[5];
    INTEGER I;
}
```

may contain a padding byte between the string and the integer, so the integer can begin on a word boundary. But this padding is system and compiler dependent, as another system or compiler may handle this data perfectly well without a padding byte. Also, some systems store integers with their most significant byte first (the industry term is “big-endian”) or with their least significant bytes first (“little-endian”).

Because compact flash is meant to be transferred among different systems, Crestron has given the programmer two different ways to store data. It can be stored with padding bytes by writing the integers, strings, structures, etc., directly (refer to WriteInteger, WriteString, WriteStructure, WriteIntegerArray, etc) or it can be stored directly as a string of bytes where the programmer controls exactly what is written (refer to FileWrite). There are corresponding functions to read each of these. Data written by one method should be read with the corresponding function. If you must write binary files, Crestron recommends the first way, for system independence. Details are listed in each function.

CheckForDisk

Name:

CheckForDisk

Syntax:

```
INTEGER CheckForDisk()
```

Description:

Tests whether or not a compact flash card is currently installed in the control system.

Parameters:

None.

Return Value:

Returns 1 if a compact flash card is currently installed in the control system. Refer to “WaitForNewDisk()” on page 184.

Example:

(Refer to “File Functions Overview” on page 116)

```
IF ( CheckForDisk () = 1 )
PRINT ( “compact flash card found” );
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

EndFileOperations

Name:

EndFileOperations

Syntax:

```
SIGNED_INTEGER EndFileOperations()
```

Description:

Signifies to the operating system that the current thread has completed its file operations.

Parameters:

None.

Return Value:

Returns 0 if successful and –1 if an error occurred.

Example:

(Refer to "File Functions Overview"on page 116)

```
IF ( StartFileOperations() < 0 );
PRINT ( “Error in starting file ops\n” );
// various file operations
IF ( EndFileOperations() < 0 )
PRINT ( “Error Occurred in ending file ops\n” );
```

NOTE: StartFileOperations is required prior to any operation accessing a file. EndFileOperations is required after finishing all file operations and prior to terminating the thread of execution (e.g., one of the PUSH commands).

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileBOF

Name:

FileBOF

Syntax:

```
SIGNED_INTEGER FileBOF (INTEGER handle)
```

Description:

Tests whether or not the current file pointer is at the beginning of the file.

Parameters:

HANDLE specifies the file handle of the previously opened file (from FileOpen).

Return Value:

Returns 1 if beginning of file or 0 if not end of file. Otherwise, file error code is returned.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle;
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
IF (nFileHandle < 0)
{
    PRINT(“Error Opening File MyFile\n”);
    return;
}
IF ( FileBOF ( nFileHandle )  = 1 )
    PRINT ( “Beginning of file reached\n” );
IF ( FileClose ( nFileHandle ) <> 0 )
    PRINT ( “Error closing file” );
    EndFileOperations();
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileClose

Name:

FileClose

Syntax:

```
SIGNED_INTEGER FileClose (INTEGER handle)
```

Description:

Closes a file opened previously by FileOpen. You MUST close a file that was opened, you won't be able to open it again, or eventually the control system may hang or reboot. A reboot clears all open files. Files must be opened and closed during a single thread of operation. Refer to "StartFileOperations()" on page 183.

Parameters:

HANDLE specifies the file handle of the previously opened file (from FileOpen).

Return Value:

Returns 0 if successful. Otherwise, file error code is returned.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER nFileHandle;
StartFileOperations();
nFileHandle = FileOpen ( "MyFile", _O_RDONLY );
IF (nFileHandle < 0)
{
    PRINT("Error Opening File MyFile\n");
    return;
}
IF ( nFileHandle > 0 )
{
    IF ( FileClose ( nFileHandle ) <> 0 )
    PRINT ( "Error closing file\n" );
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileDate

Name:

FileDate

Syntax:

```
STRING FileDate(FILE_INFO Info, INTEGER FORMAT);
```

Description:

Returns a string corresponding to the current date of the specified file with the specified FORMAT.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

FORMAT is an integer describing the way to format the date for the return. Valid formats are 1 through 4.

FORMAT 1 returns a string in the form MM/DD/YYYY

FORMAT 2 returns a string in the form DD/MM/YYYY

FORMAT 3 returns a string in the form YYYY/MM/DD

FORMAT 4 returns a string in the form MM/DD/YY

In format 4, the year 2000 is shown as 00. Digits 58 - 99 are treated as 1958-1999 and 00-57 are treated as 2000 through 2057.

Return Value:

A STRING corresponding to the current date.

Example:

(Refer to "File Functions Overview"on page 116)

```
STRING TheDate$[100];
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    TheDate$ = FileDate(FileInfo);
    PRINT ( “Date of file = %s\n”, TheDate$ );
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

This would print a string such as “Date of file = 03/25/2003”.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileDay

Name:

FileDay

Syntax:

```
STRING FileDay(FILE_INFO Info);
```

Description:

Returns the day of the week of the file as a STRING.

Parameters:

INFO – structure containing the information about a found file (refer to “FindFirst” on page 149 for description).

Return Value:

The day of the week of the file is returned in a string. Valid returns are Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, or Saturday.

Example:

(Refer to "File Functions Overview"on page 116)

```
STRING TheDay$[100];
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    TheDay$ = FileDay(FileInfo);
    PRINT ( “Day of file = %s\n”, TheDay$ );
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

An example output of this would be “Day of file = Monday”.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileDelete

Name:

FileDelete

Syntax:

```
SIGNED_INTEGER FileDelete (STRING filename)
```

Description:

Deletes the specified file from the file system.

Parameters:

FILENAME specifies the name of the file to delete. Can contain wildcards (*) if a full path is not given.

Return Value:

Returns 0 if successful. Otherwise, file error code is returned.

Example:

(Refer to "File Functions Overview"on page 116)

```
StartFileOperations();
IF ( FileDelete ( “MyFile” ) <> 0 )
    PRINT ( “Error deleting file\n” );
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileEOF

Name:

FileEOF

Syntax:

```
SIGNED_INTEGER FileEOF (INTEGER handle)
```

Description:

Tests whether or not the current file pointer is at the end of the file.

Parameters:

HANDLE specifies the file handle of the previously opened file (from FileOpen).

Return Value:

Returns 1 if end of file or 0 if not end of file. Otherwise, file error code is returned.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER nFileHandle;
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
IF (nFileHandle < 0)
{
    PRINT(“Error Opening File MyFile\n”);
    return;
}
IF ( FileEOF ( nFileHandle ) = 1 )
    PRINT ( “End of file reached\n” );
IF ( FileClose ( nFileHandle ) <> 0 )
    PRINT ( “Error closing file\n” );
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileGetDateNum

Name:

FileGetDateNum

Syntax:

```
SIGNED_INTEGER FileGetDateNum(FILEINFO Info);
```

Description:

Returns an integer corresponding to the day of the month of the file.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

The day of the month as an integer from 1 to 31.

Example:

(Refer to "File Functions Overview"on page 116)

```
INTEGER NumDateOfMonth;
FILE_INFO FileInfo;
INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    NumDateOfMonth = FileGetDateNum(FileInfo);
    PRINT ( “Day of the month of file = %d\n”, NumDateOfMonth);
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

An example output of this would be “Day of the month of file = 25”.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileGetDayOfWeekNum

Name:

FileGetDayOfWeekNum

Syntax:

```
SIGNED_INTEGER FileGetDayOfWeekNum(FILEINFO Info);
```

Description:

Returns an integer corresponding to the day of the week of file.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

The day of the week as an integer from 0 to 6; 0 represents Sunday to 6 representing Saturday.

Example:

(Refer to "File Functions Overview"on page 116)

```
INTEGER NumDayOfWeek;
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    NumDayOfWeek = FileGetDayOfWeekNum(FileInfo);
    PRINT ( “Day of week of file = %d\n”, NumDayOfWeek);
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

An example output of this would be “Day of week of file = 4”.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileGetHourNum

Name:

FileGetHourNum

Syntax:

```
SIGNED_INTEGER FileGetHourNum(FILEINFO Info);
```

Description:

Returns an integer corresponding to the number of hours in the time of the file.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

The number of hours from 0 to 23 (24-hour time format).

Example:

(Refer to "File Functions Overview"on page 116)

```
INTEGER NumHours;
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    NumHours = FileGetHourNum(FileInfo);
    PRINT ( “Hours of file time = %d\n”, NumHours);
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

An example output of this would be “Hours of file time = 22”.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileGetMinutesNum

Name:

FileGetMinutesNum

Syntax:

```
SIGNED_INTEGER FileGetMinutesNum(FILEINFO Info);
```

Description:

Returns an integer corresponding to the number of minutes in the file time.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

The number of minutes from 0 to 59.

Example:

(Refer to "File Functions Overview"on page 116)

```
INTEGER NumMinutes;
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    NumMinutes = FileGetMinutesNum(FileInfo);
    PRINT ( “Minutes of file time = %d\n”, NumMinutes);
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

An example output of this would be “Minutes of file time = 33”.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileGetMonthNum

Name:

FileGetMonthNum

Syntax:

```
SIGNED_INTEGER FileGetMonthNum(FILEINFO Info);
```

Description:

Returns an integer corresponding to the month of the year of file.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149).

Return Value:

The month of the year as an integer from 1 to 12.

Example:

(Refer to "File Functions Overview"on page 116)

```
INTEGER NumMonth;
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    NumMonth = FileGetMonthNum(FileInfo);
    PRINT ( “Month of file date = %d\n”, NumMonth);
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( “Error in closing find operation\n” );
End File Operations()
```

An example output of this would be “Month of file date = 9”.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileGetSecondsNum

Name:

FileGetSecondsNum

Syntax:

```
SIGNED_INTEGER FileGetSecondsNum(FILEINFO Info);
```

Description:

Returns an integer corresponding to the number of seconds in the time of the file.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

The number of seconds from 0 to 59.

Example:

(Refer to "File Functions Overview"on page 116)

```
INTEGER NumSeconds;
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    NumSeconds = FileGetSecondsNum(FileInfo);
    PRINT ( “Seconds of file time = %d\n”, NumSeconds);
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

An example output of this would be “Seconds of file time = 25”.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileGetYearNum

Name:

FileGetYearNum

Syntax:

```
SIGNED_INTEGER FileGetYearNum(FILEINFO Info);
```

Description:

Returns an integer corresponding to the year of the file.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

The year as an integer. The full year is specified. For example, the year 2000 will return the integer 2000.

Example:

(Refer to "File Functions Overview"on page 116)

```
INTEGER NumYear;
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    NumYear = FileGetYearNum(FileInfo);
    PRINT ( “Year of file date = %d\n”, NumYear);
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

An example output from this would be “Year of file date = 2002”.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FILE_INFO Structure

Use this structure to retrieve information about a file.

```
STRUCTURE FILE_INFO
{
STRING Name;              // relative name of the found file
INTEGER iAttributes;      // attributes for the file
INTEGER iTime;            // file time in packed form
INTEGER iDate;            // file date in packed form
LONG_INTEGER lSize;       // size of the file in bytes
};
```

File Attribute Bit Flags - ***These may be Bitwise OR'ed together***

KEYWORD	**ATTRIBUTE**	**Equivalent SIMPL+ Function**
ARDONLY	File is marked read only	IsReadOnly
AHIDDEN	File is hidden	IsHidden
ASYSTEM	File is marked as a system file	IsSystem
AVOLUME	File is a volume label	IsVolume
ADIRENT	File is a directory	IsDirectory
ARCHIVE	File is marked as an archive	-

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

NOTE: For an example of how and where to use the FILE_INFO structure, refer to the example code in "FindFirst" on page 149.

FileLength

Name:

FileLength

Syntax:

```
LONG_INTEGER FileLength (INTEGER handle)
```

Description:

Returns the length of a file.

Parameters:

HANDLE specifies the file handle of the previously opened file (from FileOpen).

Return Value:

Number of bytes if successful. Otherwise, file error code is returned.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER nFileHandle;
    StartFileOperations();
    nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
    IF (nFileHandle < 0)
    {
        PRINT(“Error Opening File MyFile\n”);
        return;
    }
IF ( nFileHandle > 0 )
    PRINT ( “Length of file = %d\n”,
        FileLength ( nFileHandle ) );
IF ( FileClose ( nFileHandle ) <> 0 )
    PRINT ( “Error closing file\n” );
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileMonth

Name:

FileMonth

Syntax:

```
STRING FileMonth(FILEINFO Info);
```

Description:

Returns the month of the file date as a string.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

The current month is returned in a string. Valid returns are January, February, March, April, May, June, July, August, September, October, November, or December.

Example:

(Refer to "File Functions Overview"on page 116)

```
STRING TheMonth$[100];
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    TheMonth$ = FileMONTH(FileInfo);
    PRINT ( “Month of file date = %s\n”, TheMonth$);
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

An example output of this would be “Month of file date = September”.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileOpen

Name:

FileOpen

Syntax:

```
SIGNED_INTEGER FileOpen (STRING filename, INTEGER flags)
```

Description:

Opens a file.

Parameters:

FILENAME specifies the full path name or relative path name (link) of the file to open/create.

FLAGS – File Open Flags. Can be combined using the Bitwise OR operator (|)

NOTE: One of the following flags must be specified: _O_RDONLY, _O_WRONLY, or _O_RDWR

KEYWORD	FUNCTION
_O_TEXT	Unused
_O_BINARY	Unused
_O_APPEND	Writes done at the end of file. Mutually exclusive with _O_TRUNC
_O_CREAT	Creates file. If _O_APPEND is specified, the file will created only if it doesn't already exist.
_O_EXCL	Open succeeds only if file doesn't already exist
_O_TRUNC	Truncates file. Mutually exclusive with _O_APPEND
_O_RDONLY	Open file for reading only
_O_RDWR	Open file for both reading and writing
_O_WRONLY	Open file for writing only

Return Value:

File handle if successful (>= 0). Otherwise, file error code is returned.

NOTE: FileClose() must be called before the executing thread is terminated. Failure to do so will result in the file being left open and locked by the control system. Should this happen, the file will not be able to be opened again until the control system is rebooted.

Examples:

(Refer to "File Functions Overview"on page 116)

Example 1: Open a read only file:

```
SIGNED_INTEGER nFileHandle;
StartFileOperations();
    nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
    IF (nFileHandle < 0)
  {
      PRINT(“Error Opening File MyFile\n”);
  }
EndFileOperations();
```

Example 2: Open an existing file to log data to the end

```
SIGNED_INTEGER nFileHandle;
StartFileOperations();
    nFileHandle = FileOpen ( “MyFile”, _O_WDONLY | _O_APPEND);
    IF (nFileHandle < 0)
  {
      PRINT(“Error Opening File MyFile\n”);
  }
EndFileOperations();
```

Example 3: Truncate an existing file and get rid of previous contents. If it does not exist, create it.

```
SIGNED_INTEGER nFileHandle;
StartFileOperations();
    nFileHandle = FileOpen ( “MyFile”, _O_WDONLY | _O_CREAT |
    _O_TRUNC);
    IF (nFileHandle < 0)
  {
      PRINT(“Error Opening File MyFile\n”);
  }
EndFileOperations();
```

Example 4: Continue adding to the end of an existing log file, or create it if it does not already exist.

```
SIGNED_INTEGER nFileHandle;
StartFileOperations();
    nFileHandle = FileOpen ( “MyFile”, _O_WDONLY | _O_APPEND
    | _O_CREAT);
    IF (nFileHandle < 0)
  {
      PRINT(“Error Opening File MyFile\n”);
  }
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileRead

Name:

FileRead

Syntax:

```
SIGNED_INTEGER FileRead (INTEGER handle, STRING buffer,
INTEGER count )
```

Description:

Reads data from a file as a series of bytes into a buffer, starting at the current file position. Refer to the section entitled "Reading and Writing Data to a File" on page 118 for a discussion of when to use this function and when to use the related functions FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray

To avoid an error being generated to the console, use FileEOF() to test for the end of the file prior to reading.

NOTE: Input and Output variables of any kind are not allowed in the file read functions.

Parameters:

HANDLE specifies the file handle of the previously opened file (from FileOpen).

BUFFER is the destination variable for bytes that are read.

COUNT specifies the number of bytes to read.

Return Value:

Number of bytes read from file. If the return value is negative, it is an error code. Refer to "File Function Error Codes" on page 117.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle;
STRING sBuf [ 100 ];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
IF (nFileHandle >= 0)
{
    WHILE (FileRead(nFileHandle, sBuf, 4096) > 0)
       PRINT ( “Read from file: %s\n”, sBuf );
    IF ( FileClose ( nFileHandle ) <> 0 )
       PRINT ( “Error closing file\n” );
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.01 or higher (Pro 2 only)

Control System:

2-Series Only

FileSeek

Name:

FileSeek

Syntax:

```
SIGNED_INTEGER FileSeek (INTEGER handle, LONG_INTEGER
offset, INTEGER origin )
```

Description:

Positions the current file pointer.

Parameters:

HANDLE specifies the file handle of previously opened file (from FileOpen).
OFFSET specifies the number of bytes to move relative to the origin.
ORIGIN is on of the file seek flags in the following table.

FileSeek Flags:

KEYWORD	FUNCTION
SEEK_SET	Start seeking from beginning of file
SEEK_CUR	Start seeking from current position in file
SEEK_END	Start seeking from end of file

Return Value:

Number of bytes offset from the beginning of file. Otherwise, file error code is returned.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle;
StartFileOperations();
nFileHandle = FileOpen(“MyFile”, _O_RDONLY);
IF (nFileHandle >= 0)
{
    IF (FileSeek( nFileHandle, 0, SEEK_SET)) < 0 )
      PRINT ( “Error seeking file\n” );
    IF ( FileClose ( nFileHandle ) <> 0 )
      PRINT ( “Error closing file\n” );
EndFileOperations();
}
```

Other Examples:

1. Go to beginning of file: FileSeek (nFileHandle, O, SEEK_SET)
2. Go to end of file: FileSeek (nFileHandle, O, SEEK_END)
3. Get current file position: CurrentBytePosition= FileSeek (nFileHandle,O, SEEK_CUR)

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileTime

Name:

FileTime

Syntax:

```
STRING FileTime(FILEINFO Info);
```

Description:

Returns a string containing the current system time.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

The return string contains the time in HH:MM:SS format, in 24-hour time. If a value is not two digits wide, it is padded with leading zeros.

Example:

(Refer to "File Functions Overview"on page 116)

```
STRING TheTime$[100];
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst("*.dat", FileInfo );
WHILE (Found = 0)
{
    TheTime$=TIME();
    PRINT ( "File time = %s\n", TheTime$);
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( "Error in closing find operation\n" );
EndFileOperations();
```

An example output from this would be “File time = 14:25:32”.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FileWrite

Name:

FileWrite

Syntax:

```
SIGNED_INTEGER FileWrite (INTEGER handle, STRING buffer,
INTEGER count )
```

Description:

Writes data from a file as a series of bytes into a buffer, starting at the current file position. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Parameters:

HANDLE specifies the file handle of the previously opened file (from FileOpen).

BUFFER is the variable containing the bytes to be written.

COUNT specifies the number of bytes to write.

Return Value:

Number of bytes written to the file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle;
STRING sBuf [ 4096 ];
StartFileOperations();
sBuf = “Hello World!”;
nFileHandle = FileOpen ( “MyFile”, _O_WRONLY );
IF (nFileHandle >= 0)
{
if( FileWrite(nFileHandle, sBuf, 4096) > 0 )
    PRINT ( “Written to file: %s\n”, sBuf );
IF ( FileClose ( nFileHandle ) <> 0 )
    PRINT ( “Error closing file\n” );
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.01 or higher (Pro 2 only)

Control System:

2-Series Only

FindClose

Name:

FindClose

Syntax:

```
SIGNED_INTEGER FindClose()
```

Description:

Signifies to the operating system that the find operation has ended.

Parameters:

None.

Return Value:

Returns 0 if successful and –1 if an error occurred.

Example:

(Refer to "File Functions Overview"on page 116)

```
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    PRINT ( “%s\n”, FileInfo.Name );
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FindFirst

Name:

FindFirst

Syntax:

```
SIGNED_INTEGER FindFirst(STRING filespec, FILE_INFO
info)
```

Description:

This command searches a directory for file(s) matching the given file specification.
Always followed with a FindClose, refer to page 148.
Requires StartFileOperations(), refer to page 183.

Parameters:

FILESPEC specifies the filename to look for. It can be a full path name or a relative path name with wildcards (the '*' character), refer to page 14.

INFO – FILE_INFO structure containing the information about a found file:

File Attribute Bit Flags: *- May be checked with bitwise and character.*

KEYWORD	ATTRIBUTE
ARDONLY	File is marked read only
AHIDDEN	File is hidden
ASYSTEM	File is marked as a system file
AVOLUME	File is a volume label
ADIRENT	File is a directory
ARCHIVE	File is marked as archived

Return Value:

Returns 0 if a file is found matching the specification and –1 if an error occurred.

Example:

(Refer to "File Functions Overview"on page 116)

```
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
if ((FileInfo.File Attributes&ADIRENT) <>0)
    PRINT (“%s is a directory\n”, FileInfo.Name
Else
    PRINT (“%s is a file\n”,FileInfo.Name
Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

NOTE: FindFirst must be followed by a FindClose.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

FindNext

Name:

FindNext

Syntax:

```
SIGNED_INTEGER FindNext(FILE_INFO info)
```

Description:

This command continues the current directory for file(s) matching the file specification in the "FindFirst" command.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

Returns 0 if a file is found matching the specification and –1 if an error occurred.

Example:

(Refer to "File Functions Overview"on page 116)

```
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst("*.dat", FileInfo );
WHILE (Found = 0)
{
    PRINT ( "%s\n", FileInfo.Name );
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( "Error in closing find operation\n" );
EndFileOperations();
```

NOTE: FindNext must be followed by a FindClose.

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

GetCurrentDirectory

Name:

GetCurrentDirectory

Syntax:

```
STRING GetCurrentDirectory()
```

Description:

Returns the complete path name of the current working directory. Refer to “Relative Path Names” on page 14 for a discussion of setting the current directory.

Parameters:

None.

Return Value:

String containing the current directory. If an error occurs, string length equals 0.

Example:

(Refer to "File Functions Overview"on page 116)

```
PRINT( “The current directory =  %s\n”,
GetCurrentDirectory());
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

IsDirectory

Name:

IsDirectory

Syntax:

```
INTEGER IsDirectory(FILE_INFO info)
```

Description:

This routine returns whether the specified file is a directory, equivalent to checking info;Attributes.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for a description).

Return Value:

Returns 1 if file is a directory and 0 otherwise.

Example:

(Refer to "File Functions Overview"on page 116)

```
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    if (IsDirectory(FileInfo))
       PRINT( “%s is a directory\n”, FileInfo.Name );
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

Version:

SIMPL+ Version 3.0x (Pro 2 only)

Control System:

2-Series Only

IsHidden

Name:

IsHidden

Syntax:

```
INTEGER IsHidden(FILE_INFO info)
```

Description:

This routine returns whether the specified file is hidden. Equivelent to checking attributes in FILE_INFO. Refer to page 135.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

Returns 1 if file is hidden and 0 if otherwise.

Example:

(Refer to "File Functions Overview"on page 116)

```
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    if (IsHidden(FileInfo))
      PRINT( “%s is hidden\n”, FileInfo.FileName );
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

IsReadOnly

Name:

IsReadOnly

Syntax:

```
INTEGER IsReadOnly(FILE_INFO info)
```

Description:

This routine returns whether the specified file is marked as read-only. Equivalent to checking attributes in FILE_INFO. Refer to page 135.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

Returns 1 if file is read-only and 0 if otherwise.

Example:

(Refer to "File Functions Overview"on page 116)

```
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    if (IsReadOnly(FileInfo))
      PRINT( “%s is read-only\n”, FileInfo.Name );
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

IsSystem

Name:

IsSystem

Syntax:

```
INTEGER IsSystem(FILE_INFO info)
```

Description:

This routine returns whether the specified file is a system file. Equivalent to checking attributes in FILE_INFO. Refer to page 135.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

Returns 1 if file is a system file and 0 if otherwise.

Example:

(Refer to "File Functions Overview"on page 116)

```
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst("*.dat", FileInfo );
WHILE (Found = 0)
{
    if (IsSystem(FileInfo))
       PRINT( "%s is a system file\n", FileInfo.Name );
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( "Error in closing find operation\n" );
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

IsVolume

Name:

IsVolume

Syntax:

```
INTEGER IsVolume(FILE_INFO info)
```

Description:

This routine returns whether the specified file is a volume label. Equivalent to checking attributes in FILE_INFO. Refer to page 135.

Parameters:

INFO – structure containing the information about a found file (refer to "FindFirst" on page 149 for description).

Return Value:

Returns 1 if file is a volume label and 0 if otherwise.

Example:

(Refer to "File Functions Overview"on page 116)

```
FILE_INFO FileInfo;
SIGNED_INTEGER Found;
StartFileOperations();
Found = FindFirst(“*.dat”, FileInfo );
WHILE (Found = 0)
{
    if (IsVolume(FileInfo))
       PRINT( “volume label = %s\n”, FileInfo.Name );
    Found = FindNext(FileInfo);
}
IF ( FindClose() < 0 )
    PRINT ( “Error in closing find operation\n” );
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

MakeDirectory

Name:

MakeDirectory

Syntax:

```
SIGNED_INTEGER MakeDirectory(STRING DirName)
```

Description:

Creates a directory with the specified name. The path name can be relative or absolute, refer to page 14. Requires StartFileOperations(), refer to page 183.

Parameters:

DIRNAME – string containing the name of the desired directory.

Return Value:

Returns 0 if successful and –1 if an error occurred.

Example:

(Refer to "File Functions Overview"on page 116)

```
IF( MakeDirectory(“NewDirect”) < 0)
    PRINT(“Error occurred creating directory\n”);
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

ReadInteger

Name:

ReadInteger

Syntax:

```
SIGNED_INTEGER ReadInteger ( INTEGER file_handle,
INTEGER i )
```

Description:

Reads an integer from a file starting at the current file position. Two bytes are read, most significant byte first. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file read functions.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

I is the integer whose value is read.

Return Value:

Number of bytes read from file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
INTEGER i;
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = ReadInteger(nFileHandle, i);
    if (iErrorCode > 0)
       PRINT ( “Read integer from file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

ReadIntegerArray

Name:

ReadIntegerArray

Syntax:

```
SIGNED_INTEGER ReadIntegerArray( INTEGER file_handle,
INTEGER iArray[m][n] )
```

Description:

Reads the array from a file starting at the current file position. Two bytes are read, most significant first containing the row dimension of the array, then two more bytes are read, containing the column dimension of the array. Then each integer is read as a two byte quantity, most significant byte first. The integers are stored in row-major order, e.g. all the elements of row 0 first, then the elements of row 1, etc. Note that there is one more row and one more column than the dimensions that are read, because there is a row 0 and a column 0. Refer to the section entitled "Reading and Writing Data to a File" on page 118 for a discussion of when to use this function and when to use the related functions: FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file read functions.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

IARRAY is the array whose values are read.

Return Value:

Number of bytes read from file. If the return value is negative, it is an error code. An error occurs if the array is not large enough to hold the data.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
INTEGER iArray[10];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = ReadIntegerArray(nFileHandle, iArray);
    if (iErrorCode > 0)
      PRINT ( “Read array from file correctly.\n”);
    else
      PRINT ( “Error code %d\n”, iErrorCode);
  }
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

ReadLongInteger

Name:

ReadLongInteger

Syntax:

```
SIGNED_INTEGER ReadLongInteger ( INTEGER file_handle,
LONG_INTEGER li )
```

Description:

Reads a long integer from a file starting at the current file position. Four bytes are read, most significant byte first and least significant byte last. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file read functions.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

LI is the long integer whose value is read.

Return Value:

Number of bytes read from file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
LONG_INTEGER li;
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = ReadLongInteger(nFileHandle, li);
    if (iErrorCode > 0)
       PRINT ( “Read long integer from file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
    }
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

ReadLongIntegerArray

Name:

ReadLongIntegerArray

Syntax:

```
SIGNED_INTEGER ReadLongIntegerArray ( INTEGER
file_handle,
LONG_INTEGER ilArray[m][n] )
```

Description:

Reads the array from a file starting at the current file position. Two bytes are read, most significant first containing the row dimension of the array, then two more bytes are read, containing the column dimension of the array. Then each long integer is read as a four byte quantity, most significant byte first. The integers are stored in row-major order, e.g. all the elements of row 0 first, then the elements of row 1, etc. Note that there is one more row and one more column than the dimensions that are read, because there is a row 0 and a column 0. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file read functions.

An error occurs if the array is not long enough to hold the data.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

ilArray is the array whose values are read.

Return Value:

Number of bytes read from file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
LONG_INTEGER ilArray[10];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = ReadLongIntegerArray(nFileHandle, ilArray);
    if (iErrorCode > 0)
       PRINT ( “Read array from file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

ReadSignedInteger

Name:

ReadSignedInteger

Syntax:

```
SIGNED_INTEGER ReadSignedInteger ( INTEGER file_handle,
SIGNED_INTEGER si )
```

Description:

Reads a signed integer from a file starting at the current file position. Two bytes are read, most significant first. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file read functions.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

SI is the signed integer whose value is read.

Return Value:

Number of bytes read from file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
SIGNED_INTEGER si;
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = ReadSignedInteger(nFileHandle, si);
    if (iErrorCode > 0)
       PRINT ( “Read signed integer from file correctly\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
  }
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

ReadSignedIntegerArray

Name:

ReadSignedIntegerArray

Syntax:

```
SIGNED_INTEGER ReadSignedIntegerArray ( INTEGER
file_handle,
SIGNED_INTEGER isArray[m][n] )
```

Description:

Reads the array from a file starting at the current file position. Two bytes are read, most significant first containing the row dimension of the array, then two more bytes are read, containing the column dimension of the array. Then each signed integer is read as a two byte quantity, most significant byte first. The integers are stored in row-major order, e.g. all the elements of row 0 first, then the elements of row 1, etc. Note that there is one more row and one more column than the dimensions that are read, because there is a row 0 and a column 0. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file read functions.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

ISARRAY is the array whose values are read.

Return Value:

Number of bytes read from file. If the return value is negative, it is an error code. An error occurs if the array is not large enough to hold the data.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
SIGNED_INTEGER isArray[10][5];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
IF (nFileHandle >= 0)
{
iErrorCode = ReadSignedIntegerArray(nFileHandle, isArray);
if (iErrorCode > 0)
PRINT ( “Read array from file correctly.\n”);
else
PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

ReadSignedLongInteger

Name:

ReadSignedLongInteger

Syntax:

```
SIGNED_INTEGER ReadSignedLongInteger ( INTEGER
file_handle,
SIGNED_LONG_INTEGER sli )
```

Description:

Reads data from a file starting at the current file position. Each element of the structure is read, without any padding bytes, that might actually be there in memory. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file read functions.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

SLI is the signed long integer whose value is read.

Return Value:

Number of bytes read from file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
SIGNED_LONG_INTEGER sli;
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = ReadSignedLongInteger(nFileHandle, sli);
    if (iErrorCode > 0)
       PRINT ( “Read from file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

ReadSignedLongIntegerArray

Name:

ReadSignedLongIntegerArray

Syntax:

```
SIGNED_INTEGER ReadSignedLongIntegerArray ( INTEGER
file_handle,
SIGNED_LONG_INTEGER sliArray[m][n] )
```

Description:

Reads the array from a file starting at the current file position. Two bytes are read, most significant first containing the row dimension of the array, then two more bytes are read, containing the column dimension of the array. Then each signed long integer is read as a four byte quantity, most significant byte first. The integers are stored in row-major order, e.g. all the elements of row 0 first, then the elements of row 1, etc. Note that there is one more row and one more column than the dimensions that are read, because there is a row 0 and a column 0. Refer to the section entitled "Reading and Writing Data to a File" on page 118 for a discussion of when to use this function and when to use the related functions: FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file read functions.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

SLIARRAY is the array whose values are read.

Return Value:

Number of bytes read from file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
SIGNED_LONG_INTEGER sliArray[10][5];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = ReadSignedLongIntegerArray(nFileHandle,
sliArray);
    if (iErrorCode > 0)
       PRINT ( “Read array from file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

ReadString

Name:

ReadString

Syntax:

```
SIGNED_INTEGER ReadString ( INTEGER file_handle, STRING
s )
```

Description:

Reads a string from a file starting at the current file position. Internally, the string is stored as a 2-byte length, most significant byte first, then the actual string bytes. In the case of a string variable, the total number of bytes written is calculated from the size of the string, not the string allocation size. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

S is the string whose value is read.

Return Value:

Number of bytes read from file into the string. If the return value is negative, it is an error code. An error occurs if the string is not large enough to hold the data.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
STRING s[100];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = ReadString( nFileHandle, s);
    if (iErrorCode > 0)
       PRINT ( “Read string from file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

ReadStringArray

Name:

ReadStringArray

Syntax:

```
SIGNED_INTEGER ReadStringArray ( INTEGER file_handle,
STRING s[] )
```

Description:

Reads a string from a file starting at the current file position. Internally, the string is stored with the first 2 bytes indicating the total number of string written, then each string follows as a 2-byte length, most significant byte first, then the actual string bytes. In the case of a string variable, the total number of bytes is the calculated from the size of the string, not the string allocation size. Refer to the section entitled "Reading and Writing Data to a File" on page 118 for a discussion of when to use this function and when to use the related functions: FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

S is the string whose value is read.

Return Value:

Number of bytes read from file into the string. If the return value is negative, it is an error code. An error occurs if the array is not large enough to hold the data.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
STRING s[100][100];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = ReadStringArray( nFileHandle, s);
    if (iErrorCode > 0)
       PRINT ( “Read string from file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
    }
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

ReadStructure

Name:

ReadStructure

Syntax:

```
ReadStructure ( INTEGER nFileHandle, STRUCTURE struct [,
INTEGER nTotalBytesRead] )
```

Description:

Reads data from a file starting at the current file position. Each element of the structure is read, without any padding bytes, that might actually be there in memory. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileRead, ReadInteger, ReadString, ReadStructure, ReadSignedInteger, ReadLongInteger, ReadLongSignedInteger, ReadIntegerArray, ReadSignedIntegerArray, ReadLongIntegerArray, ReadLongSignedIntegerArray, ReadStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

There is no error if the structure does not match the data.

Parameters:

nFileHandle - File handle of the previously opened file (from FileOpen).

struct - Structure variable that will receive data read from file

nTotalBytesRead - optional argument. INTEGER variable that will contain the total number of bytes read from the file into the structure.

Return Value:

None.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, nTotalBytesRead;
STRUCTURE PhoneBookEntry
{
    STRING Name[50];
    STRING Address[100];
    STRING PhoneNumber[20];
};
PhoneBookEntry OneEntry;
StartFileOperations();
nFileHandle = FileOpen ( “MyFile.txt”, _O_RDONLY );
if (nFileHandle >= 0)
{
    ReadStructure( nFileHandle, PhoneBookEntry,
    nTotalBytesRead );
    if( nTotalBytesRead < 0 )
       PRINT ( “Error reading structure.  Error code = %d\n”,
       nTotalBytesRead );
    else
       PRINT ( “Read structure from file correctly. Total
        bytes read = %d\n”, nTotalBytesRead );
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

RemoveDirectory

Name:

RemoveDirectory

Syntax:

```
SIGNED_INTEGER RemoveDirectory(STRING DirName)
```

Description:

Removes the directory with the specified name. The path name can be a relative link or absolute, refer to page page 14. Must be empty. Requires StartFileOperations(), refer to page page 183.

Parameters:

DIRNAME – string containing the name of the desired directory.

Return Value:

Returns 0 if successful and –1 if an error occurred.

Example:

(Refer to "File Functions Overview"on page 116)

```
IF( RemoveDirectory(“NewDirect”) < 0)
PRINT(“Error occurred deleting directory\n”);
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

SetCurrentDirectory

Name:

SetCurrentDirectory

Syntax:

```
SIGNED_INTEGER SetCurrentDirectory(STRING DirName)
```

Description:

Changes the working directory to the specified name. Refer to “Relative Path Names” on page 14.

Parameters:

DIRNAME – string containing the name of the desired directory.

Return Value:

Returns 0 if successful and –1 if an error occurred.

Example:

(Refer to "File Functions Overview"on page 116)

```
IF( SetCurrentDirectory(“NewDirect”) < 0)
PRINT(“Error occurred creating directory\n”);
PRINT(“Directory is now: %s\n”, GetCurrentDirectory());
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

StartFileOperations

Name:

StartFileOperations

Syntax:

```
SIGNED_INTEGER StartFileOperations()
```

Description:

Signifies to the operating system that the current thread is starting its file operations.

Parameters:

None.

Return Value:

Returns 0 if successful and –1 if an error occurred.

Example:

(Refer to "File Functions Overview"on page 116)

```
IF ( StartFileOperations() < 0 )
PRINT ( “Error in starting file ops\n” );
// various file operations
IF ( EndFileOperations() < 0 )
PRINT ( “Error Occurred in ending file ops\n” );
```

NOTE: StartFileOperations is required prior to any operation accessing a file. EndFileOperations is required after finishing all file operations and prior to terminating the thread of execution (e.g., one of the PUSH commands).

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

WaitForNewDisk

Name:

WaitForNewDisk

Syntax:

```
SIGNED_INTEGER WaitForNewDisk()
```

Description:

Waits for a compact flash card to be inserted into the control system. Refer to “CheckForDisk” on page 119.

Parameters:

None.

Return Value:

Returns 0 when a new compact flash card is installed into the control system, <0 if an error occurs.

Example:

(Refer to "File Functions Overview"on page 116)

```
while(1)
{
if ( WaitForNewDisk() < 0 )
break;
// perform operations on the new disk. Read a file, etc.
}
```

Version:

SIMPL+ Version 3.00.02 or higher (Pro 2 only)

Control System:

2-Series Only

WriteInteger

Name:

WriteInteger

Syntax:

```
SIGNED_INTEGER WriteInteger ( INTEGER file_handle,
INTEGER i )
```

Description:

Writes an integer from a file starting at the current file position. Two bytes are written, most significant byte first. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Use ReadInteger to read this.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

I is the integer whose value is written.

Return Value:

Number of bytes written to the file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
INTEGER i;
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_WRONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = WriteInteger(nFileHandle, i);
    if (iErrorCode > 0)
       PRINT ( “Written to file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

WriteIntegerArray

Name:

WriteIntegerArray

Syntax:

```
SIGNED_INTEGER WriteIntegerArray( INTEGER file_handle,
INTEGER iArray[m][n] )
```

Description:

Writes the array from a file starting at the current file position. Two bytes are written, most significant first containing the row dimension of the array, then two more bytes are written, containing the column dimension of the array. Then each integer is written as a two byte quantity, most significant byte first. The integers are stored in row-major order, e.g. all the elements of row 0 first, then the elements of row 1, etc. Note that there is one more row and one more column than the dimensions that are written, because there is a row 0 and a column 0. Refer to the section entitled "Reading and Writing Data to a File" on page 118 for a discussion of when to use this function and when to use the related functions: FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Use ReadLongIntegerArray to read this.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

IARRAY is the array whose values are written.

Return Value:

Number of bytes written to the file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
INTEGER iArray[10];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_WRONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = WriteIntegerArray(nFileHandle, iArray);
    if (iErrorCode > 0)
       PRINT ( “Array written to file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

WriteLongInteger

Name:

WriteLongInteger

Syntax:

```
SIGNED_INTEGER WriteLongInteger ( INTEGER file_handle,
LONG_INTEGER li )
```

Description:

Writes a long integer from a file starting at the current file position. Four bytes are written, most significant byte first. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Use ReadLongInteger to read this.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

LI is the long integer whose value is written.

Return Value:

Number of bytes written to the file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
LONG_INTEGER li;
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_WRONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = WriteLongInteger(nFileHandle, li);
    if (iErrorCode > 0)
       PRINT ( “Written to file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

WriteSignedInteger

Name:

WriteSignedInteger

Syntax:

```
SIGNED_INTEGER WriteSignedInteger ( INTEGER
file_handle,
SIGNED_INTEGER si )
```

Description:

Writes a signed integer from a file starting at the current file position. Two bytes are written, most significant first. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Use ReadSignedInteger to read this.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

SI is the signed integer whose value is written.

Return Value:

Number of bytes written to the file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
SIGNED_INTEGER si;
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_WRONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = WriteSignedInteger(nFileHandle, si);
    if (iErrorCode > 0)
       PRINT ( “Written to file correctly\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

WriteSignedIntegerArray

Name:

WriteSignedIntegerArray

Syntax:

```
SIGNED_INTEGER WriteSignedIntegerArray ( INTEGER
file_handle,
SIGNED_INTEGER isArray[m][n] )
```

Description:

Writes the array from a file starting at the current file position. Two bytes are written, most significant first containing the row dimension of the array, then two more bytes are Write, containing the column dimension of the array. Then each signed integer is written as a two byte quantity, most significant byte first. The integers are stored in row-major order, e.g. all the elements of row 0 first, then the elements of row 1, etc. Note that there is one more row and one more column than the dimensions that are written, because there is a row 0 and a column 0. Refer to the section entitled "Reading and Writing Data to a File" on page 118 for a discussion of when to use this function and when to use the related functions: FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Use ReadSignedIntegerArray to read this.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

ISARRAY is the array whose values are Write.

Return Value:

Number of bytes written to the file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
SIGNED_INTEGER isArray[10][5];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_WRONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = WriteSignedIntegerArray(nFileHandle, isArray);
    if (iErrorCode > 0)
       PRINT ( “Array written to file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

WriteSignedLongInteger

Name:

WriteSignedLongInteger

Syntax:

```
SIGNED_INTEGER WriteSignedLongInteger ( INTEGER
file_handle,
SIGNED_LONG_INTEGER sli )
```

Description:

Writes data from a file starting at the current file position. Each element of the structure is written, without any padding bytes, that might actually be there in memory. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

Use ReadSignedLongInteger to read this.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

SLI is the signed long integer whose value is written.

Return Value:

Number of bytes written to the file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
INTEGER  nFileHandle, iErrorCode;
SIGNED_LONG_INTEGER sli;
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_WRONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = WriteSignedLongInteger(nFileHandle, sli);
    if (iErrorCode > 0)
       PRINT ( “Written to file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.01 or higher (Pro 2 only)

Control System:

2-Series Only

WriteSignedLongIntegerArray

Name:

WriteSignedLongIntegerArray

Syntax:

```
SIGNED_INTEGER WriteSignedLongIntegerArray ( INTEGER
file_handle,
SIGNED_LONG_INTEGER sliArray[m][n] )
```

Description:

Writes the array from a file starting at the current file position. Two bytes are written, most significant first containing the row dimension of the array, then two more bytes are written, containing the column dimension of the array. Then each signed long integer is written as a four byte quantity, most significant byte first. The integers are stored in row-major order, e.g. all the elements of row 0 first, then the elements of row 1, etc. Note that there is one more row and one more column than the dimensions that are written, because there is a row 0 and a column 0. Refer to the section entitled "Reading and Writing Data to a File" on page 118 for a discussion of when to use this function and when to use the related functions: FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Use ReadSignedLongIntegerArray to read this.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

SLIARRAY is the array whose values are written.

Return Value:

Number of bytes written to the file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
SIGNED_LONG_INTEGER sliArray[10][5];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_WRONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = WriteSignedLongIntegerArray(nFileHandle,
sliArray);
    if (iErrorCode > 0)
       PRINT ( “Array written to file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

WriteString

Name:

WriteString

Syntax:

```
SIGNED_INTEGER WriteString ( INTEGER file_handle, STRING
s )
```

Description:

Writes a string to a file starting at the current file position. Internally, the string is stored as a 2-byte length, most significant byte first, then the actual string bytes. In the case of a string variable, the total number of bytes written is the calculated from the size of the string, not the string allocation size. Refer to the section entitled "Reading and Writing Data to a File" on page 118 for a discussion of when to use this function and when to use the related functions: FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Use ReadString to read this.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

S is the string whose value is written.

Return Value:

Number of bytes written to the file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
STRING s[100];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_WRONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = WriteString( nFileHandle, s);
    if (iErrorCode > 0)
       PRINT ( “String written to file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

WriteStringArray

Name:

WriteStringArray

Syntax:

```
SIGNED_INTEGER WriteStringArray ( INTEGER file_handle,
STRING s[] )
```

Description:

Writes a string array to a file starting at the current file position. Internally, the string is stored with the first 2 bytes indicating the total number of strings written, then each string follows as a 2-byte length, most significant byte first, then the actual string bytes. In the case of a string variable, the total number of bytes is calculated from the size of the string, not the string allocation size. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Use ReadStringArray to read this.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

S is the string whose value is written.

Return Value:

Number of bytes written to the file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, iErrorCode;
STRING s[100][100];
StartFileOperations();
nFileHandle = FileOpen ( “MyFile”, _O_WRONLY );
    IF (nFileHandle >= 0)
    {
iErrorCode = WriteStringArray( nFileHandle, s);
    if (iErrorCode > 0)
       PRINT ( “String written to file correctly.\n”);
    else
       PRINT ( “Error code %d\n”, iErrorCode);
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

WriteStructure

Name:

WriteStructure

Syntax:

```
WriteStructure ( INTEGER nFileHandle, STRUCTURE struct
[, INTEGER nTotalBytesWritten] )
```

Description:

Writes data to a file starting at the current file position. Each element of the structure is written, without any padding bytes, that might actually be there in memory. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

NOTE: Input and Output variables of any kind are not allowed in the file reading and writing functions, just internal variables.

Use ReadStructure to read this.

Parameters:

nFileHandle - File handle of the previously opened file (from FileOpen).

struct - Structure variable whose data will be written to the file.

nTotalBytesWritten - optional argument. INTEGER variable that will contain the total number of bytes written to the file from the structure.

Return Value:

None.

Example:

(Refer to "File Functions Overview"on page 116)

```
SIGNED_INTEGER  nFileHandle, nTotalBytesWritten;
STRUCTURE PhoneBookEntry
{
    STRING Name[50];
    STRING Address[100];
    STRING PhoneNumber[20];
};
PhoneBookEntry OneEntry;
StartFileOperations();
nFileHandle = FileOpen ( “MyFile.txt”, _O_WRONLY );
if (nFileHandle >= 0)
{
    WriteStructure( nFileHandle, PhoneBookEntry,
    nTotalBytesWritten );
    if( nTotalBytesWritten < 0 )
       PRINT ( “Error writing structure.  Error code = %d\n”,
nTotalBytesWritten );
    else
       PRINT ( “Structure written to file correctly.  Total
bytes written = %d\n”, nTotalBytesWritten );
}
EndFileOperations();
```

Version:

SIMPL+ Version 3.00.07 or higher (Pro 2 only)

Control System:

2-Series Only

Mathematical Functions

Mathematical Functions Overview

These functions perform general mathematical operations in a given SIMPL+ program by operating on one or more numerical arguments and returning an INTEGER as a result.

Abs

Name:

Abs

Syntax:

```
INTEGER Abs(INTEGER SOURCE);
INTEGER Abs(SIGNED_INTEGER or SOURCE);
```

Description:

Takes the absolute value of SOURCE. If SOURCE is negative, a positive value is returned. If SOURCE is already positive, the same value is returned.

Parameters:

Takes the absolute value of an INTEGER.

Return Value:

An INTEGER corresponding to the absolute value of SOURCE.

Example:

```
DIGITAL_INPUT TRIG;
INTEGER I, K;
I=-5;

CHANGE TRIG
{
K=ABS(I);
PRINT(“Original Value = %d, Absolute Value = %d\n”, I, K);
}
```

The output would be:

```
Original Value = -5, Absolute Value = 5
```

Version:

SIMPL+ Version 1.00

Max

Name:

Max

Syntax:

```
INTEGER Max(INTEGER VAL1, INTEGER VAL2)
```

Description:

Determine the maximum of two values based on an unsigned comparison.

Parameters:

VAL1 and VAL2 are both INTEGER values on which the test is performed.

Return Value:

The maximum of Val1, Val2 after an unsigned comparison is performed. Refer to "Signed vs. Unsigned Arithmetic" on page 21 for a further explanation of how the values are compared.

Example:

```
INTEGER X, Y;

FUNCTION MAIN()
{
X = MAX(65535, 0);
Y = MAX(25, 26);
}
```

X would be 65535, and Y would be 26.

Version:

SIMPL+ Version 1.00

MIN

Name:

Min

Syntax:

```
INTEGER Min(INTEGER VAL1, INTEGER VAL2)
```

Description:

Determine the minimum of two values based on an unsigned comparison.

Parameters:

VAL1 and VAL2 are both INTEGER values on which the test is performed.

Return Value:

The minimum of Val1, Val2 after an unsigned comparison is performed. Refer to “Signed vs. Unsigned Arithmetic” for a further explanation of how the values are compared.

Example:

```
INTEGER X, Y;

FUNCTION MAIN()
{
X = MIN(65535, 0);
Y = MIN(25, 26);
}
```

X would be 0, and Y would be 25.

Version:

SIMPL+ Version 1.00

MulDiv

Name:

MulDiv

Syntax:

```
INTEGER MulDiv(INTEGER VAL1, INTEGER VAL2, INTEGER VAL3)
```

Description:

Computes the result (VAL1 * VAL2)/VAL3.

Parameters:

VAL1, VAL2, and VAL3 are INTEGER values.

Return Value:

A 16-bit integer is returned based on the above equation. The arithmetic operations are performed using unsigned arithmetic. Note that 32-bit math is used internally, so that if VAL1*VAL2 is greater than a 16-bit number, accuracy is maintained. If the final result is greater than 16-bits, the lower 16-bits are returned.

Example:

```
INTEGER X, Y;

FUNCTION MAIN()
{
X = 1970;
Y = 40;
PRINT(“The result of (%d * %d)/25 = %d\n”, X, Y,
MULDIV(X, Y, 25);
}
```

The PRINT statement would show the result as being 3152. In this case, X*Y is greater than a 16-bit number, but accuracy is maintained due to the use of 32-bit arithmetic internally.

Version:

SIMPL+ Version 1.00

SMAX

Name:

SMax

Syntax:

```
INTEGER SMax(INTEGER VAL1, INTEGER VAL2)
```

Description:

Determine the maximum of two values based on a signed comparison.

Parameters:

VAL1 and VAL2 are both INTEGER values on which the test is performed.

Return Value:

The maximum of Val1, Val2 after a signed comparison is performed. Refer to "Signed vs. Unsigned Arithmetic" for a further explanation of how the values are compared.

Example:

```
INTEGER X, Y;
FUNCTION MAIN()
{
X = SMAX(65535, 0);
Y = SMAX(25, 26);
}
```

X would be 0 (65535 interpreted as -1), and Y would be 26.

Version:

SIMPL+ Version 1.00

SMin

Name:

SMin

Syntax:

```
INTEGER SMin(INTEGER VAL1, INTEGER VAL2)
```

Description:

Determine the minimum of two values based on a signed comparison.

Parameters:

VAL1 and VAL2 are both INTEGER values on which the test is performed.

Return Value:

The minimum of Val1, Val2 after a signed comparison is performed. Refer to "Signed vs. Unsigned Arithmetic" on page 21 for a further explanation of how the values are compared.

Example:

```
INTEGER X, Y;

FUNCTION MAIN()
{
X = SMIN(65535, 0);
Y = SMIN(25, 26);
}
```

X would be 65535 (interpreted as -1), and Y would be 25.

Version:

SIMPL+ Version 1.00

Random Number Functions

Random Number Functions Overview

These functions allow a SIMPL+ program to generate a random number.

Random

Name:

Random

Syntax:

```
INTEGER Random(INTEGER LowerBound, INTEGER UpperBound);
```

Description:

Generate a random number. Refer to “Seed” on page 213 and “Rnd” on page 212.

Parameters:

LowerBound is an INTEGER specifying the lower end of the range.

UpperBound is an INTEGER specifying the upper end of the range.

Both LowerBound and UpperBound are treated as unsigned values.

Return Value:

Returns an unsigned number from LowerBound to UpperBound. Both LowerBound and UpperBound are legal values.

Example:

```
INTEGER NUM;

FUNCTION MAIN()
{
NUM = RANDOM(25, 80);
PRINT(“The random number between 25 and 80 is: %d\n”, NUM);
}
```

An example output from this would be:

```
The random number between 25 and 80 is: 42
```

Version:

SIMPL+ Version 1.00

Rnd

Name:

Rnd

Syntax:

```
INTEGER Rnd();
```

Description:

Generate a random number. Refer to “Seed” on page 213 and “Random” on page 211.

Parameters:

None.

Return Value:

An INTEGER from 0 to 65535.

Example:

```
INTEGER NUM;

FUNCTION MAIN()
{
NUM = RND();
PRINT(“The random number is: %d\n”, NUM);
}
```

Version:

SIMPL+ Version 1.00

Seed

Name:

Seed

Syntax:

```
Seed(INTEGER SeedValue);
```

Description:

Provides a seed or origin for the random number generator so that the numbers returned by RND and RANDOM are pseudo-random numbers. SEED is not required for generating random numbers as the random number generator will be seed with a default value.

This default value is issued at control system restart, not program restart. That is, if you don't used the SEED call, you will not get the same value if you restart the program. For any particular value of SEED, the random number generator will generate a predictable series of numbers. Note that specifying the seed value is global to all SIMPL+ programs running inside a control system. The sequence begins again whenever SEED is called.

Parameters:

None.

Return Value:

None.

Example:

```
INTEGER NUM;

FUNCTION MAIN()
{
SEED(25);
NUM = RANDOM(25, 80);
PRINT(“The random number between 25 and 80 is: %d\n”, NUM);
}
```

Version:

SIMPL+ Version 1.00

String Formatting & Printing Functions

String Formatting & Printing Functions Overview

The printing functions are used to take INTEGER and STRING type arguments in a SIMPL+ program, format them in a user specified way, and send the output to either the COMPUTER port of the control system or to another STRING.

MakeString

Name:

MakeString

Syntax:

```
MakeString(STRING DESTINATION, <Static Specification
String> [, <arg1> ...]);
MakeString(0 | 1 | 2, <Static Specification String> [,
<arg1> ...]);
```

Description:

MAKESTRING is a variant of PRINT (Refer to page 216). The output of MAKESTRING goes to the DESTINATION string. It can print simple text strings or complex formatted strings. The second form of MAKESTRING allows different destinations to be selected:

0: Console Port, same as PRINT.

1: CPU (same functionality as SendPacketToCPU function)

2: Cresnet Network (same functionality as SendCresnetPacket function).

NOTE: In the second form, the first argument may not be a variable containing 0, 1, 2. It must be the written as 0, 1, 2. Crestron is discouraging the use of the second form of MAKESTRING in favor of either the PRINT command or alternate methods for activating devices that do not require knowledge of Cresnet packets, which are subject to change.

Parameters:

DESTINATION is a string where the output goes to after it has been formatted and processed. For a further description of formatting, refer to PRINT that begins on page 216.

Return Value:

None.

Example:

```
INTEGER X;
STRING Z[100], OUT[100];

X=10;
Z=”Hello”;

FUNCTION MAIN()
{
// Puts “This is a string” followed by a CRLF onto OUT.
MAKESTRING(OUT, “This is string\n”);

// Puts “The value of X is 10 in decimal, 0A in hex”
// followed by CRLF onto OUT.
MAKESTRING(OUT, “The value of X is %u in decimal, %02X in
hex\n”, X, X);

// Puts “The String value is Hello” onto OUT.
MAKESTRING(OUT, “The String value is %s”, Z);
}
```

Version:

SIMPL+ Version 2.00 for Console, Cresnet, and CPU destinations.

SIMPL+ Version 1.00 for everything else.

Print

Name:

Print

Syntax:

```
PRINT(<Static Specification String> [, <arg1> ...]);
```

Description:

The output of PRINT goes to the CONSOLE port of the control system and can be monitored in the Crestron Viewport. It can print simple text strings or complex formatted strings.

Parameters:

<Static Specification String> is a quoted string that contains text and formatting information. Format specifiers are of the form:

```
%[[Pad]Width]specifier
```

Valid Format Specifiers

s	Specifies a BUFFER_INPUT, STRING, or STRING_INPUT variable. (unprintable characters are printed in the format that Viewport uses)
d	Specifies an ANALOG_INPUT, ANALOG_OUTPUT, or INTEGER to be printed as a signed decimal value.
u	Specifies an ANALOG_INPUT, ANALOG_OUTPUT, or INTEGER to be printed as an unsigned decimal value.
x	Specifies an ANALOG_INPUT, ANALOG_OUTPUT, or INTEGER to be printed as a lowercase hexadecimal number.
X	Specifies an ANALOG_INPUT, ANALOG_OUTPUT, or INTEGER to be printed as an uppercase hexadecimal number.
l	Specifies a long_integer or unsigned_long_integer will follow
%	Prints a % sign (i.e. use %% to print a % sign).
%lD	Specifies a LONG_INTEGER to be printed as a signed decimal value.
%c	Specifies a printable ASCII character to be printed.

The optional Width specifier is a number that states the width of the field as characters. If the value to be printed is less than the Width, it is padded on the left with spaces. Width can be two digits.

The optional Pad specifier works with the Width specifier. If the result of the Width operation results in the need to add spaces, the Pad specifier can be used to pad with different values rather than a space. '0' is the only valid pad value, i.e. %03d pads with leading zeros so 1Z would be printed as 012.

As each % value is found, it pulls the matching <arg> off the list. The first % uses <arg1>, the second % uses <arg2>, etc. If the number of % specifiers does not match the number of arguments, the program will generate a compile error, the compiler also checks to make sure the format specifier matches the type of the variable being used (i.e. if %d is used, the variable being used should be INTEGER type).

NOTE: If no format specifiers are used, then a simple quoted text string is printed.

In the <Static Specification String>, certain values may be printed using “escape sequences”. Escape sequences start with the \ character and have a variable number of characters following. The following table specifies the legal escape sequences:

ESCAPE	MEANING	HEX CONSTANT
\n	Carriage Return + Linefeed	\x0D\0A
\t	Tab	\x09
\b	Backspace	\x08
\r	Carriage Return	\x0D
\f	Form Feed	\x0C
\a	Audible Alert (Bell)	\x07
\\	Backslash	\x5C
\'	Single Quote	\x27
\”	Double Quote	\x22
\xZZ	Hex Constant. Z can range from 0-9, a-f or A-F	\xZZ

Return Value:

None.

Example:

```
INTEGER X;
STRING Z[100];
X=10;
Z=”Hello”;
FUNCTION MAIN()
{
// Outputs “This is a string” followed by a CRLF.
PRINT(“This is a string\n”);
// Outputs “The value of X is 10 in decimal, 0A in hex”
// followed by CRLF.
PRINT(“The value of X is %u in decimal, %02X in hex\n”,
X, X);
// Outputs “The String value is Hello”
PRINT(“The String value is %s”, Z);
}
```

Version:

SIMPL+ Version 1.00

String Concatenation

String concatenation can be performed either using the + operator or by using MAKESTRING or PRINT functions. It is easier to use the + operator in general usage, although the formatting options of the MAKESTRING and PRINT functions give greater flexibility.

The + operator for strings is used the same way as in mathematical expressions. String concatenation may only be used as a standalone statement. The syntax is:

```
<Destination_string> = <String1 > [+ <String2> ...];
```

When string values appear on the right-side of the equal sign, the exact contents are appended to the new string. <String> values may be of type literal (quoted) strings, BUFFER_INPUT, STRING, STRING_INPUT, or any function that returns a string.

Examples:

```
STRING A$[100], B$[100], C$[100];

B$=”Hello”;
C$=”World!”;
I=56;
J=2;

// This will output “Hello562World!”
A$=B$+ITOA(I)+ITOA(J)+”xyz”+C$;
PRINT(“%s”, A$);

// This will output “VHello2World”
A$=CHR(I)+B$+ITOA(J)+C$;
PRINT(“%s”, A$);
```

String Parsing & Manipulation Functions

String Parsing and Manipulation Functions Overview

String parsing and manipulation functions are used where the contents of string variables need to be examined or modified.

ClearBuffer

Name:

ClearBuffer

Syntax:

```
ClearBuffer(STRING BUFFERNAME);
```

Description:

Deletes the contents of the specified buffer. If a LEN is done on the buffer after a CLEARBUFFER, the return value will be 0. This is equilavent to assigning an empty string to the buffer, e.g., `BUFFERNAME="";`

Parameters:

BUFFERNAME specifies the name of the string to empty. BUFFER_INPUT, STRING, and STRING_INPUT sources are legal.

Return Value:

None.

Example:

```
BUFFER_INPUT IN$[100];

CHANGE IN$
{
IF(RIGHT$(IN$,1) = "Z")
CLEARBUFFER(IN$);
// Code to process IN$ goes here.
}
```

In this example, if the last character that comes into the BUFFER_INPUT is “Z”, the buffer is cleared.

Version:

SIMPL+ Version 1.00

Find

Name:

Find

Syntax:

```
INTEGER Find(STRING MATCH_STRING, STRING SOURCE_STRING
[,INTEGER START_POSITION]);
```

Description:

Finds the position in SOURCE_STRING where MATCH_STRING first occurs.

Parameters:

MATCH_STRING is a STRING containing the data to be searched.

SOURCE_STRING is a STRING containing the data to be searched.

START_POSITION is an INTEGER which tells FIND at what character in the string to start the search, and is 1 based. If not specified, it defaults to 1.

Return Value:

The index of where MATCH_STRING first occurs (going left to right) in SOURCE_STRING. If a match can not be found, or POSITION exceeds the length of the SOURCE_STRING then 0 is returned. The index is 1 based.

Example:

```
STRING_INPUT IN$[100];
INTEGER START_LOC;
CHANGE IN$
{
START_LOC = FIND(“XYZ”, IN$);
PRINT(“XYZ was found starting at position %d in %s\n”,
START_LOC, IN$);
}
```

If IN$ was set equal to “Hello, World!” then START_LOC would be 0 since “XYZ” can not be found. If IN$ was equal to “CPE1704XYZXYZ”, then START_LOC would be equal to 8.

Version:

SIMPL+ Version 1.00

Gather

Name:

Gather

Syntax:

```
STRING Gather(STRING DELIMITER, STRING SOURCESTRING);
```

Description:

Concatenates the data from SOURCESTRING and issues it on the return string when the specified delimiter has been reached. Note that when GATHER is executed, if SOURCESTRING does not include the DELIMITER, then the equivalent of a PROCESSLOGIC is performed. When the system returns to the GATHER, it will once again check for the proper delimiter. In effect, section of code (a CHANGE statement, for example) is held up at the GATHER until the proper data is received.

Parameters:

The gather function searches the SOURCESTRING for the DELIMITER string.

NOTE: It makes sense only to use GATHER with STRING_INPUT or BUFFER_INPUT types.

Return Value:

The concatenated string which includes the delimiter specified. Example:

```
BUFFER_INPUT COM$[100];
DIGITAL_INPUT trig;
STRING IN$[100];

PUSH trig
{
IN$ = GATHER(“\n”, COM$);
PRINT(“The value of IN$ is %s\n”, IN$);
}
```

In this example, the event is started when TRIG goes high. When data comes into COM$, the GATHER statement is evaluated. The PRINT statement is never reached until the delimiter \n (CRLF) is found. When the delimiter is found, then the string will be printed. Note that the GATHERed string will have the \n on it.

Example:

```
BUFFER_INPUT COM$[100];
DIGITAL_INPUT trig;
STRING IN$[100];

CHANGE COM$
{
IN$ = GATHER(“\n”, COM$);
PRINT(“The value of IN$ is %s\n”, IN$);
}
```

In the 2-Series Control System processors, a GATHER that is waiting for data will use up the next change of the BUFFER_INPUT until the terminating character is encountered. That is, any CHANGE event handler for the BUFFER_INPUT will not be called.

If, in the first event, COM$ contains the string “Hello”, the event will wait in the GATHER. When the COM$ changes again to contain “World!\n”, the event will immediately resume after the GATHER. The CHANGE COM$ event will only be called once in this case. In the X-Generation Control Systems, the CHANGE event would be called both times.

Version:

SIMPL+ Version 2.00

GetC

Name:

GetC

Syntax:

```
INTEGER GetC(BUFFER_INPUT SOURCE);
```

Description:

Returns the value at position 1 of SOURCE string and shifts the rest of the buffer up by one. In this way, values may be picked out of a buffer for processing.

Parameters:

SOURCE is typically from a BUFFER_INPUT statement. It may be defined as a STRING or STRING_INPUT, but since GETC removes characters from SOURCE, the result is destructive to the source string.

Return Value:

An INTEGER containing a single character from position 1 of the buffer.

If there are no characters in the buffer for GETC to retrieve, then the value of 65535 is returned.

Example:

In this example, a buffer input is read until the character “A” is retrieved.

```
BUFFER_INPUT IN$[100];
INTEGER INCHAR;

CHANGE IN$
{
INCHAR = 0;
WHILE(INCHAR <> 'A')
INCHAR = GETC(IN$);

// continue processing.
}
```

Version:

SIMPL+ Version 1.00

Left

Name:

Left

Syntax:

```
STRING Left(STRING SOURCE, INTEGER NUM);
```

Description:

Takes the leftmost NUM characters of SOURCE and returns them in an output string.

Parameters:

SOURCE is a STRING containing the source string.

NUM is an INTEGER that tells LEFT how many characters to use in the computation.

Return Value:

A string representing the leftmost NUM characters of SOURCE. If NUM is greater than the number of characters in SOURCE, then the return is identical to SOURCE.

Example:

```
STRING_INPUT Var$[100];
STRING Temp$[100];

CHANGE Var$
{
Temp$ = LEFT(Var$, 5);
PRINT(“Left most 5 characters of %s are %s\n”, Var$, Temp$);
}
```

In this example, if Var$ is “abcdefghijk”, Temp$ will contain “abcde”.

Version:

SIMPL+ Version 1.00

Len

Name:

Len

Syntax:

```
INTEGER Len(STRING SOURCE);
```

Description:

Returns the length of the actual string, not the declared maximum length.

Parameters:

SOURCE is a string whose length is to be determined.

Return Value:

A value from 0 - 65535, which gives the number of characters in the string. An empty string returns a length of 0.

Example:

```
STRING_INPUT IN$[100];
INTEGER Temp;

CHANGE IN$
{
Temp = LEN(IN$);
PRINT(“The Length of %s is %d\n”, IN$, Temp);
}
```

In this example, if IN$ is equal to “This is a test” then Temp will contain the integer 14.

Version:

SIMPL+ Version 1.00

Lower

Name:

Lower

Syntax:

```
STRING Lower(STRING SOURCE);
```

Description:

Takes a source string and converts characters with the values a-z (lowercase) to A-Z (uppercase).

Parameters:

SOURCE is a string to be converted to lowercase. SOURCE is not modified, unless it is also used as the return value, e.g., `S$=LOWER(S$);`

Return Value:

A STRING containing the lowercase version of SOURCE. Characters that do not fall into the range A-Z are not modified and will stay as specified.

Example:

```
STRING_INPUT IN$[100];
STRING LOWER$[100];

CHANGE IN$
{
LOWER$ = LOWER(IN$);
PRINT(“Lowercase version of %s is %s\n”,IN$, LOWER$);
}
```

In this example, if IN$ contains “This is a Test 123!”, then LOWER$ will contain “this is a test 123!”.

Version:

SIMPL+ Version 1.00

Mid

Name:

Mid

Syntax:

```
STRING Mid(STRING SOURCE, INTEGER START, INTEGER NUM);
```

Description:

Returns a string NUM characters long from SOURCE, starting at position START.

Parameters:

SOURCE is a STRING containing the input string.

START is an INTEGER telling MID at which character position in SOURCE to start. The first character of SOURCE is considered 1.

NUM is an INTEGER telling MID how many characters to use from SOURCE.

Return Value:

A string NUM characters long starting at START.

If START is greater than the length of SOURCE, an empty STRING is returned.

If NUM is greater than the total number of characters that can be retrieved starting from START, only the remaining characters in SOURCE will be pulled. For example, MID("ABCD", 2, 10) would return a STRING containing BCD.

Example:

```
STRING_INPUT Var$[100];
STRING Temp$[100];

CHANGE Var$
{
Temp$ = MID(Var$, 2, 5);
PRINT("String starting at position 2 for 5 characters is
%s\n",Temp$);
}
```

In this example, if Var$ contains "abcdefghijklmnop", then Temp$ will contain "bcdef".

Version:

SIMPL+ Version 1.00

Remove

Name:

Remove

Syntax:

```
STRING Remove(STRING DELIMITER, STRING SOURCESTRING
[, INTEGER POSITION]);
```

Description:

Begins searching a string <source> for the <delimiter> at the specified position, then removes all characters from the beginning of the string <source> up to and including the delimiter. Returns a string containing all of the removed characters.

Parameters:

DELIMITER is a string containing the string to match for.

Search within the string, SOURCESTRING is the string to search within.

POSITION is an optional integer which specifies how many characters into SOURCESTRING to start. It defaults to 1, which is the first character of SOURCESTRING.

Return Value:

If the specified DELIMITER is found, the contents of the source string, up to and including the delimiter are returned. The original source string is modified.

Example:

```
BUFFER_INPUT SOURCE$[50];
STRING OUTPUT$[50];
CHANGE SOURCE$
{
OUTPUT$ = REMOVE(“abc”, SOURCE$);
}
```

In this example, if SOURCE$ were “testabc123”, then OUTPUT$ would be “testabc” and SOURCE$ would contain “123”.

```
BUFFER_INPUT SOURCE$[50];
STRING OUTPUT$[50];
CHANGE SOURCE$
{
OUTPUT$ = REMOVE(“abc”, SOURCE$, 6);
}
```

If SOURCE$ were “testabcabc123”, then OUTPUT$ would be “testabcabc” and SOURCE$ would contain “123”.

Version:

SIMPL+ Version 2.00

REVERSEFIND

Name:

ReverseFind

Syntax:

```
INTEGER ReverseFind(STRING MATCH_STRING, STRING
SOURCE_STRING
[, INTEGER START_POSITION]);
```

Description:

Finds the position in SOURCE_STRING where MATCH_STRING last occurs.

Parameters:

MATCH_STRING is a STRING containing the searched for data.

SOURCE_STRING is a STRING containing the data to be searched.

START_POSITION is an INTEGER which tells REVERSEFIND at what character in the string to start the search, and is 1 based. If it is not specified, it defaults to the end of the string.

Return Value:

The index of where MATCH_STRING last occurs (going right to left) in SOURCE_STRING. If the data can not be found, or POSITION exceeds the length of the SOURCE_STRING then 0 is returned. The index is 1 based.

Example:

```
STRING_INPUT IN$[100];
INTEGER START_LOC;

CHANGE IN$
{
START_LOC = REVERSEFIND(“XYZ”,IN$);
PRINT(“last XYZ occurance was found at position %d in %s\n”,
START_LOC,IN$);
}
```

If IN$ was set equal to “Hello, World!” then START_LOC would be 0 since “XYZ” can not be found. If IN$ was equal to “CPE1704XYZXYZ”, then START_LOC would be equal to 11.

Version:

SIMPL+ Version 1.00

Right

Name:

Right

Syntax:

```
STRING Right(STRING SOURCE, INTEGER NUM);
```

Description:

Takes the rightmost NUM characters of SOURCE and returns them in an output string.

Parameters:

SOURCE is a STRING containing the source string.

NUM is an INTEGER that tells RIGHT how many characters to use in the computation.

Return Value:

A string representing the rightmost NUM characters of SOURCE. If NUM is greater than the number of characters in SOURCE, then the return is identical to SOURCE.

Example:

```
STRING_INPUT Var$[100]
STRING Temp$[100];

CHANGE Var$
{
Temp$ = RIGHT(Var$, 5);
PRINT("Right most 5 characters of %s are %s\n", Var$, Temp$);
}
```

In this example, if Var$ contains “abcdefghijk”, then Temp$ contains “ghijk”.

Version:

SIMPL+ Version 1.00

SetString

Name:

SetString

Syntax:

```
INTEGER SetString(STRING SOURCE, INTEGER POSITION,
STRING DESTINATION);
```

Description:

Overwrites the bytes in DESTINATION with the bytes in SOURCE starting at POSITION in the DESTINATION string.

Parameters:

DESTINATION is a STRING containing the string to be modified.

POSITION is an INTEGER referencing the starting byte to write at in DESTINATION. 1 is the first byte of the string.

SOURCE is a STRING containing the string to use in the operation.

Return Value:

The new length or an error code as defined below:

For the purposes of the explanation, a string has been declared STRING DESTINATION[MAX_LEN]. The string has a current length defined by LEN(DESTINATION).

e.g., If the specified position is beyond the declared length of the destination string:

> If POSITION > MAX_LEN, no operation is performed and -8 is returned.

e.g., If the entire source string can't be inserted without exceeding the length of the destination string:

> If POSITION-1+LEN(SOURCE) > MAX_LEN, the operation is performed, the string is truncated and -4 is returned.

e.g., If the position exceeds the current length of the destination:

> If POSITION > LEN(DESTINATION), the string is padded with spaces and -2 is returned.

e.g., If the source string will make the destination string longer:

> If POSITION-1+LEN(SOURCE) > LEN(DESTINATION), the string will be expanded to fit and -1 will be returned.

NOTE: If more than one condition is met (typically -2 and -1 would be met at the same time), the codes are added together as the return value.

NOTE: The subroutine knows the max length of the destination string.

If the operation meets none of the above conditions, the new length is returned.

The return code may be ignored (as in the following example).

Example:

```
STRING DESTINATION$[100];

DESTINATION$ = :\”Space XXXX To Fill”;

SETSTRING(“ABCD”, 7, DESTINATION$);
```

This would result in DESTINATION containing the string “Space ABCD To Fill”. If the return code were used, it would contain 18 (the string length).

Version:

SIMPL+ Version 1.00

Upper

Name:

Upper

Syntax:

```
STRING Upper(STRING SOURCE);
```

Description:

Takes a source string and converts characters with the values a-z (lowercase) to A-Z (uppercase).

Parameters:

SOURCE is a string to be converted to uppercase. SOURCE is not modified, unless it is also used as the return value, e.g., `S$=UPPER(S$);`

Return Value:

A STRING containing the uppercase version of SOURCE. Characters that do not fall into the range a-z are not modified and will stay as specified.

Example:

```
STRING_INPUT IN$[100];
STRING UPPER$[100];

CHANGE IN$
{
UPPER$ = UPPER(IN$);
PRINT(“Uppercase version of %s is %s\n”,IN$, UPPER$);
}
```

In this example, if IN$ contains “Hello There 123!” then UPPER$ contains “HELLO THERE 123!”.

Version:

SIMPL+ Version 1.00

System Control

System Control Overview

These constructs control system behavior and may change the resultant flow of the given SIMPL+ program.

Delay

Name:

Delay

Syntax:

```
Delay(INTEGER TIME);
```

Description:

Forces a task switch and starts a timer for the hundredths of a second specified by TIME. The system continues with the statements after a delay when the delay time has expired. Refer to “WAIT” on page 265.

Parameters:

TIME is the number of hundredths of a second to delay. For example, 500 specifies a 5-second delay.

Return Value:

None.

Example:

```
// A delay of 525 hundredths of a second or 5.25 seconds
#define_constant MY_DELAY 525

DELAY(MY_DELAY);
```

Version:

SIMPL+ Version 1.00

ProcessLogic

Name:

ProcessLogic

Syntax:

```
ProcessLogic();
```

Description:

Forces a task switch away from the current SIMPL+ module, so that the SIMPL Windows program can process the outputs of the SIMPL+ module. Refer to the discussion on Task Switching on page 25.

Parameters:

None.

Return Value:

None.

Example:

```
INTEGER X;
ANALOG_OUTPUT I;

FOR(X=0 TO 25)
{
I = X;
PROCESSLOGIC();
}
```

In this example, the analog output I is updated every pass through the loop. Logic dependent upon the analog value will refer to the new analog value every pass through the loop.

Version:

SIMPL+ Version 1.00

Pulse

Name:

Pulse

Syntax:

```
Pulse(TIME, DIGITAL_OUTPUT OUT);
```

Description:

Pulses the output high then low for the specified length of time (in hundredths of a second). When the pulse starts, a task switch is performed so other logic can be processed. If the output is already high, the SIMPL Windows logic processor will not see a change and no further actions will be triggered.

Parameters:

TIME is the number of hundredths of a second to pulse. For example, 500 specifies a 5-second delay.

OUT is a DIGITAL_OUTPUT that is to be pulsed.

Return Value:

None.

NOTE: (X-Gen only)Elements of a DIGITAL_OUTPUT array cannot be used within the Pulse function.

Example:

```
// A pulse of 525 hundredths of a second or 5.25 seconds
#define_constant MY_PULSE_TIME 525

DIGITAL_OUTPUT OutputToPulse;

PULSE(MY_PULSE_TIME, OutputToPulse);
```

This will execute immediately and output a pulse of 5.25 seconds to the digital output OutputToPulse.

Version:

SIMPL+ Version 1.00

TerminateEvent

Name:

TerminateEvent

Syntax:

```
TerminateEvent;
```

Description:

Exits a CHANGE, PUSH, or RELEASE event. It may also be used to exit a loop in the main() function if desired. TERMINATEEVENT cannot be used inside of a function.

Example:

```
INTEGER X;
ANALOG_INPUT Y;

CHANGE Y
{
X=0;
WHILE(X<25)
{
IF(Y = 69)
TerminateEvent;
X = X + 1;
PRINT(“X=%d\n”, X);
}
}
```

In this example, the CHANGE event will terminate if the ANALOG_INPUT Y equals the value of 69. Otherwise, the CHANGE will exit after the WHILE loop finishes.

Version:

SIMPL+ Version 2.00 - No longer allowed inside functions, RETURN should be used. Existing code that relies on the event terminating should be revised.

SIMPL+ Version 1.00

System Interfacing

System Interfacing Overview

These functions control the way the SIMPL+ program communicates with Cresnet network devices and the CPU.

GenerateUserNotice

Name:

GenerateUserNotice

Syntax:

```
GenerateUserNotice(<Static Specification String> [, <arg1>
...]);
```

Description:

Places a notification message into the control system's error log

Parameters:

<Static Specification String> is a quoted string that contains text and formatting information. Format specifiers are of the form: %[[Pad]Width]specifier

Refer to "Print" on page 216 for a list and description of valid Format Specifiers.

Return Value:

None.

Example:

```
Function MyFunc()
{
   STRING sNotice;
   sNotice = "Projector";
   GenerateUserNotice( "The %s bulb has a total of %d
   hours", sNotice, 500 );
 }
```

Version:

SIMPL+ Version 3.01.07

Control System:

2-Series Only

GenerateUserWarning

Name:

GenerateUserWarning

Syntax:

```
GenerateUserWarning(<Static Specification String> [, <arg1>
...]);
```

Description:

Places a warning message into the control system's error log

Parameters:

<Static Specification String> is a quoted string that contains text and formatting information. Format specifiers are of the form: %[[Pad]Width]specifier

Refer to "Print" on page 216 for a list and description of valid Format Specifiers.

Return Value:

None.

Example:

```
Function MyFunc()
{
    STRING sWarning;
    sWarning = "Projector";
    GenerateUserWarning( "The %s bulb has a total of %d
    hours", sWarning, 800 );
 }
```

Version:

SIMPL+ Version 3.01.07

Control System:

2-Series Only

GenerateUserError

Name:

GenerateUserError

Syntax:

```
GenerateUserError(<Static Specification String> [, <arg1>
...]);
```

Description:

Places an error message into the control system's error log

Parameters:

<Static Specification String> is a quoted string that contains text and formatting information. Format specifiers are of the form: %[[Pad]Width]specifier

Refer to "Print" on page 216 for a list and description of valid Format Specifiers.

Return Value:

None.

Example:

```
Function MyFunc()
{
    STRING sError;
    sError = "Projector";
    GenerateUserError( "The %s bulb has exceeded %d hours",
    sError, 1000 );
 }
```

Version:

SIMPL+ Version 3.01.07

Control System:

2-Series Only

CheckForNVRAMDisk

Name:

CheckForNVRAMDisk

Syntax:

```
INTEGER CheckForNVRAMDisk()
```

Description:

Tests whether or not an NVRam Disk is currently installed in the control system.

Parameters:

None.

Return Value:

Returns 1 if an NVRam Disk is currently installed in the control system.

Example:

(Refer to “File Functions Overview” on page 116)

```
IF ( CheckForNVRAMDisk() = 1 )
    PRINT ( "NVRAM Disk found" );
```

Version:

SIMPL+ Version 3.01.07 or higher (Pro 2 only)

Control System:

2-Series Only

Time & Date Functions

Time and Date Functions Overview

Time and Date functions in a given SIMPL+ program are used to retrieve information about the current date and time from the system clock. Values can be retrieved as either text strings i.e. “January” or as integer values. Typically, integer values are used if computations need to be performed (i.e. when the date is the 25th, perform a specific action).

Date

Name:

Date

Syntax:

```
STRING Date(INTEGER FORMAT);
```

Description:

Returns a string corresponding to the current date with the specified FORMAT.

Parameters:

FORMAT is an integer describing the way to format the date for the return. Valid formats are 1 through 4.

FORMAT 1 returns a string in the form MM/DD/YYYY
FORMAT 2 returns a string in the form DD/MM/YYYY
FORMAT 3 returns a string in the form YYYY/MM/DD
FORMAT 4 returns a string in the form MM/DD/YY

In format 4, the year 2000 is shown as 00. Digits 58 - 99 are treated as 1958-1999 and 00-57 are treated as 2000 through 2057.

Return Value:

A STRING corresponding to the current date.

Example:

```
STRING TheDate$[100];

FUNCTION MAIN()
{
TheDate$=DATE(1);
PRINT(“The date is %s\n”, TheDate$);
}
```

This would print a string such as “The date is 03/25/2002”.

Version:

SIMPL+ Version 1.00

Day

Name:

Day

Syntax:

```
STRING Day();
```

Description:

Returns the day of the week as a STRING.

Parameters:

None.

Return Value:

The day of the week is returned in a string. Valid returns are Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, or Saturday.

Example:

```
STRING TheDay$[100];

FUNCTION MAIN()
{
TheDay$=DAY();
PRINT(“The day of the week is %s\n”, TheDay$);
}
```

An example output of this would be “The day of the week is Monday”.

Version:

SIMPL+ Version 1.00

GETDATENUM

Name:

GetDateNum

Syntax:

```
INTEGER GetDateNum();
```

Description:

Returns an integer corresponding to the current day of the month.

Parameters:

None.

Return Value:

The day of the month as an integer from 1 to 31.

Example:

```
INTEGER NumDateOfMonth;

FUNCTION MAIN()
{
NumDateOfMonth = GetDateNum();
PRINT("The current day of the month is %d\n", NumDateOfMonth);
}
```

An example output of this would be “The current day of the month is 25”.

Version:

SIMPL+ Version 1.00

GETDAYOFWEEKNUM

Name:

GetDayOfWeekNum

Syntax:

```
INTEGER GetDayOfWeekNum();
```

Description:

Returns an integer corresponding to the current day of the week.

Parameters:

None.

Return Value:

The day of the week as an integer from 0 to 6; 0 represents Sunday to 6 representing Saturday.

Example:

```
INTEGER NumDayOfWeek;

FUNCTION MAIN()
{
NumDayOfWeek = GetDayOfWeekNum();
PRINT(“The current day of the week is %d\n”, NumDayOfWeek);
}
```

An example output of this would be “The current day of the week is 4”.

Version:

SIMPL+ Version 1.00

GETHOURNUM

Name:

GetHourNum

Syntax:

```
INTEGER GetHourNum();
```

Description:

Returns an integer corresponding to the number of hours in the current time.

Parameters:

None.

Return Value:

The number of hours from 0 to 23 (24-hour time format).

Example:

```
INTEGER NumHours;

FUNCTION MAIN()
{
NumHours = GetHourNum();
PRINT(“The Number of hours on the clock is %d\n”, NumHours);
}
```

An example output of this would be “The Number of hours on the clock is 22”.

Version:

SIMPL+ Version 1.00

GETHSECONDS

Name:

GetHSeconds

Syntax:

```
INTEGER GetHSeconds();
```

Description:

Returns an integer corresponding to the number of hundredths of a second based on the system clock. Typically, this function could be used for very fine timing, to determine if a specific amount of time has elapsed.

Parameters:

None.

Return Value:

The number of hundredths of a second based on the system clock.

Example:

```
INTEGER OldTime, NewTime, Loop;

Loop=0;
OldTime=GETHSECONDS();
WHILE(Loop < 10000)
{
Loop = Loop + 1
}
NewTime=GETHSECONDS();

PRINT (“Elapsed Time is %d hundredths of a second.\n”,
Newtime-OldTime);
```

The output of this code would be “Elapsed Time is 400 hundredths of a second.”

NOTE: This is bad programming as it ties up the CPU.

Version:

SIMPL+ Version 1.00

GETMINUTESNUM

Name:

GetMinutesNum

Syntax:

```
INTEGER GetMinutesNum();
```

Description:

Returns an integer corresponding to the number of minutes in the current time.

Parameters:

None.

Return Value:

The number of minutes from 0 to 59.

Example:

```
INTEGER NumMinutes;

FUNCTION MAIN()
{
NumMinutes = GetMinutesNum();
PRINT(“The Number of minutes on the clock is %d\n”,
NumMinutes);
}
```

An example output of this would be “The Number of minutes on the clock is 33”.

Version:

SIMPL+ Version 1.00

GETMONTHNUM

Name:

GetMonthNum

Syntax:

```
INTEGER GetMonthNum();
```

Description:

Returns an integer corresponding to the current month of the year.

Parameters:

None.

Return Value:

The month of the year as an integer from 1 to 12.

Example:

```
INTEGER NumMonth;

FUNCTION MAIN()
{
NumMonth = GetMonthNum();
PRINT(“The current month of the year is %d\n”, NumMonth);
}
```

An example output of this would be “The current month of the year is 9”.

Version:

SIMPL+ Version 1.00

GETSECONDSNUM

Name:

GetSecondsNum

Syntax:

```
INTEGER GetSecondsNum();
```

Description:

Returns an integer corresponding to the number of seconds in the current time.

Parameters:

None.

Return Value:

The number of seconds from 0 to 59.

Example:

```
INTEGER NumSeconds;

FUNCTION MAIN()
{
NumSeconds = GetSecondsNum();
PRINT(“The Number of seconds on the clock is %d\n”,
NumSeconds);
}
```

An example output of this would be “The Number of seconds on the clock is 25”.

Version:

SIMPL+ Version 1.00

GETTICKS

Name:

GetTicks

Syntax:

```
INTEGER GetTicks();
```

Description:

Returns an integer corresponding to the number of system ticks. Each tick is 1/112.5 seconds on an X-generation control system, or 0.01 seconds on a 2-series control system. Typically, this function could be used for very fine timing, to determine if a specific amount of time has elapsed. The use of this function is discouraged, GetHSeconds() should be used instead.

Parameters:

None.

Return Value:

The number of ticks in the clock.

Example:

```
INTEGER OldTime, NewTime, Loop;

Loop=0;
OldTime=GETTICKS();
WHILE(Loop < 10000)
{
Loop = Loop + 1;
}
NewTime=GETTICKS();

PRINT(“Elapsed Time is %d ticks\n”, Newtime-OldTime);
```

An example output from this code fragment would be “Elapsed Time is 7000 ticks”.

NOTE: This is bad programming as it ties up the CPU.

Version:

SIMPL+ Version 1.00

GETYEARNUM

Name:

GetYearNum

Syntax:

```
INTEGER GetYearNum();
```

Description:

Returns an integer corresponding to the current year.

Parameters:

None.

Return Value:

The year as an integer. The full year is specified. For example, the year 2000 will return the integer 2000.

Example:

```
INTEGER NumYear;

FUNCTION MAIN()
{
NumYear = GetYearNum();
PRINT("The current year is %d\n", NumYear);
}
```

An example output from this would be "The current year is 1999".

Version:

SIMPL+ Version 1.00

MONTH

Name:

Month

Syntax:

```
STRING Month();
```

Description:

Returns the current month as a string.

Parameters:

None.

Return Value:

The current month is returned in a string. Valid returns are January, February, March, April, May, June, July, August, September, October, November, or December.

Example:

```
STRING TheMonth$[100];

FUNCTION MAIN()
{
TheMonth$=MONTH();
PRINT("The Month is %s\n", TheMonth$);
}
```

An example output of this would be “The Month is September”.

Version:

SIMPL+ Version 1.00

SETCLOCK

Name:

SetClock

Syntax:

```
SetClock(INTEGER HOURS, INTEGER MINUTES, INTEGER
SECONDS);
```

Description:

Sets the system clock.

Parameters:

HOURS is an integer containing the hour portion of the time to which the clock is set. HOURS is expressed in 24-hour format, which can range from 0 to 23.

MINUTES is an integer containing the minutes portion of the time to which the clock is set. MINUTES range from 0 to 59.

SECONDS is an integer containing the seconds portion of the time to which the clock is set. SECONDS range from 0 to 59.

Return Value:

None.

Example:

```
ANALOG_INPUT Hours, Minutes, Seconds;

CHANGE Hours, Minutes, Seconds
{
SetClock(Hours, Minutes, Seconds);
PRINT("Current Time is: %s\n", Time());
}
```

In this example, the Hours, Minutes, and Seconds are specified from an external SIMPL Program. For example, if Hours = 5, Minutes = 10, Seconds = 25, then the output will be Current Time is: 05:10:25.

Version:

SIMPL+ Version 2.00

SETDATE

Name:

SetDate

Syntax:

```
SetDate(INTEGER MONTH, INTEGER DAY, INTEGER YEAR);
```

Description:

Sets the system date.

Parameters:

MONTH is an integer containing the month to which the date is set. A valid range is 1 through 12, corresponding to January through December.

DAY is an integer containing the day of the month to which the date is set. The range varies from month to month, but always starts at 1.

YEAR is an integer containing the year to which the date is set. The year is four digits, i.e. 1999.

Return Value:

None.

Example:

```
ANALOG_INPUT Month, Day, Year;

CHANGE Month, Day, Year
{
SetDate(Month, Day, Year);
PRINT("Current Date is: %s\n", Date(1));
}
```

In this example, Month, Day, and Year come from a SIMPL Windows program. For example, if Month = 12, Day = 25, Year = 1999, the output from this program will be Current Date = 12/25/1999.

Version:

SIMPL+ Version 2.00

TIME

Name:

Time

Syntax:

```
STRING TIME();
```

Description:

Returns a string containing the current system time.

Parameters:

None.

Return Value:

The return string contains the time in HH:MM:SS format, in 24-hour time. If a value is not two digits wide, it is padded with leading zeros.

Example:

```
STRING TheTime$[100];

FUNCTION MAIN()
{
TheTime$=TIME();
PRINT(“The Time is %s\n”, TheTime$);
}
```

An example output from this would be “The Time is 14:25:32”.

Version:

SIMPL+ Version 1.00

Wait Events

Wait Events Overview

When writing a SIMPL+ program, it is often desirable to have an event that will be processed a predetermined amount of time after it is triggered. The WAIT event allows a block of code to be executed after a specified amount of time. There are related functions which allow WAITS to be paused, resumed, cancelled, or have their times changed. The system supports up to 200 total timed events that may be running at any given time across all SIMPL+ modules.

Timed events include: WAIT, DELAY, and PAUSE statements.

A WAIT statement differs from a DELAY in both timing and order of statement execution. In a WAIT statement, the WAIT block executes only after the specified amount of time, but execution proceeds immediately to the statement following the WAIT block. In a DELAY, all execution is halted until the delay is finished.

CancelAllWait

Name:

CancelAllWait

Syntax:

```
CancelAllWait();
```

Description:

Cancels all WAIT events for the current SIMPL+ program. When an event is cancelled, it is removed from the wait list and will not activate. There is no effect on wait events that have finished running.

Parameters:

None.

Return Value:

None.

Example:

```
DIGITAL_INPUT Trig, KillWaits;

PUSH Trig
{
    WAIT(1000, FirstWait)
  }
    PRINT(“Wait 1 Triggered!\n”);
  {
    WAIT(2000, SecondWait)
  }
    PRINT(“Wait 2 Triggered!\n”);
  }
}
PUSH KillWaits
{
CancelAllWait();
}
```

In this example, when Trig is pushed, a 10-second and 20-second event are scheduled. Whichever wait events are still running when KillWaits is triggered, will be removed from the Wait list and will not get activated.

Version:

SIMPL+ Version 1.00

CancelWait

Name:

CancelWait

Syntax:

```
CancelWait(NAME);
```

Description:

Cancels a specified named WAIT event in the current SIMPL+ program. When an event is cancelled, it is removed from the wait list and will not activate. There is no effect if the wait event has finished running.

Parameters:

NAME is a name of a previously defined and named WAIT event.

Return Value:

None.

Example:

```
DIGITAL_INPUT Trig, KillWaits;

PUSH Trig
{
    WAIT(1000, FirstWait)
    }
    PRINT(“Wait 1 Triggered!\n”);
    {
    WAIT(2000, SecondWait)
    }
    PRINT(“Wait 2 Triggered!\n”);
    }
}
PUSH KillWaits
{
Cancelwait(FirstWait);
}
```

In this example, when Trig is pushed, a 10-second and 20-second event are scheduled. When KillWaits is triggered, if FirstWait is still on the wait list, it will be removed from the wait list and will not get activated. SecondWait will activate at the end of the 20-second wait time.

Version:

SIMPL+ Version 1.00

PauseAllWait

Name:

PauseAllWait

Syntax:

```
PauseAllWait();
```

Description:

Pauses all WAIT events for the current SIMPL+ program. When an event is paused, the timer for it freezes and may later be resumed, retimed, or cancelled. When a wait is resumed, it executes the remaining time from when it was paused until the defined wait time.

Parameters:

None.

Return Value:

None.

Example:

```
DIGITAL_INPUT Trig, PauseWaits;

PUSH Trig
{
    WAIT(1000, FirstWait)
    }
    PRINT(“Wait 1 Triggered!\n”);
    {
    WAIT(2000, SecondWait)
    }
    PRINT(“Wait 2 Triggered!\n”;
    }
}
PUSH PauseWaits
{
PauseAllWaits();
}
```

In this example, when Trig is pushed, a 10-second and 20-second event is scheduled. When PauseWaits is triggered, any of the running WAIT events will be halted, but may later be resumed, cancelled, or retimed.

Version:

SIMPL+ Version 1.00

PauseWait

Name:

PauseWait

Syntax:

```
PauseWait(NAME);
```

Description:

Pauses a specified named WAIT event in the current SIMPL+ program. When an event is paused, the timer for it freezes and may later be resumed, retimed, or cancelled. When a wait is resumed, it executes the remaining time from when it was paused until the defined wait time.

Parameters:

NAME is a name of a previously defined and named WAIT event.

Return Value:

None.

Example:

```
DIGITAL_INPUT Trig, PauseWait;

PUSH Trig
{
    WAIT(1000, FirstWait)
    }
    PRINT(“Wait 1 Triggered!\n”);
    {
    WAIT(2000, SecondWait)
    }
    PRINT(“Wait 2 Triggered!\n”;
    }
}
PUSH PauseWait
{
PauseWait(SecondWait);

}
```

In this example, when Trig is pushed, a 10-second and 20-second event is scheduled. When PauseWait is triggered, the SecondWait event will be paused if it has not already run to completion. It may be later cancelled, resumed, or retimed.

Version:

SIMPL+ Version 1.00

ResumeAllWait

Name:

ResumeAllWait

Syntax:

```
ResumeAllWait();
```

Description:

Resumes all WAIT events for the current SIMPL+ program that had been previously paused. The WAIT will execute when the time from when it was frozen until the specified wait time has elapsed.

Parameters:

None.

Return Value:

None.

Example:

```
DIGITAL_INPUT Trig, PauseWaits, ResumeWaits;

PUSH Trig
{
    WAIT(1000, FirstWait)
    }
    PRINT(“Wait 1 Triggered!\n”);
    {
    WAIT(2000, SecondWait)
    }
    PRINT(“Wait 2 Triggered!\n”;
    }
}
PUSH PauseWaits
{
PauseAllWait();
}
PUSH ResumeWaits
{
ResumeAllWaits();
}
```

In this example, when Trig is pushed, a 10-second and 20-second event is scheduled. When PauseWaits is triggered, any of the running WAIT events will be halted. When ResumeWaits is triggered, the previously paused waits will resume from when they were paused. For example, if FirstWait and SecondWait were paused at 5-second, when ResumeAllWait is called, FirstWait will have 5-seconds more to execute and SecondWait will have 15-seconds more to execute.

Version:

SIMPL+ Version 1.00

ResumeWait

Name:

ResumeWait

Syntax:

```
ResumeWait(NAME);
```

Description:

Resumes the specified named WAIT event in the current SIMPL+ program that has been previously paused. The WAIT will execute from the time when it was paused until the specified wait time has elapsed.

Parameters:

NAME is a name of a previously defined and named WAIT event.

Return Value:

None.

Example:

```
DIGITAL_INPUT Trig, PauseWaits, ResumeWaits;
PUSH Trig
{
    WAIT(1000, FirstWait)
    }
    PRINT("Wait 1 Triggered!\n");
     {
    WAIT(2000, SecondWait)
    }
    PRINT("Wait 2 Triggered!\n";
    }
}
PUSH PauseWaits
{
PauseAllWait();
}
PUSH ResumeWait
{
ResumeWait(FirstWait);
}
```

In this example, when Trig is pushed, a 10-second and 20-second event is scheduled. When PauseWaits is triggered, any of the running WAIT events will be halted. When ResumeWaits is triggered, the FirstWait event that was previously paused will resume from when it was paused. For example, if FirstWait was paused at 5-seconds, when ResumeWait(FirstWait) is called, FirstWait will have 5-seconds more to execute. SecondWait will still be paused.

Version:

SIMPL+ Version 1.00

RetimeWait

Name:

RetimeWait

Syntax:

```
RetimeWait(INTEGER TIME, NAME);
```

Description:

Changes the time for a wait event in progress. When a WAIT is retimed, the WAIT is restarted. For example, if a 5-second wait is 3-second in, and it is retimed to 10-second, a full 10-seconds must elapse before the WAIT triggers.

Parameters:

TIME is an integer that specifies the new wait time in hundredths of a second. If time is set to 0, the event will occur immediately.

NAME is a name of a previously defined WAIT event.

Return Value:

None.

Example:

```
DIGITAL_INPUT Trig, ChangeWaitTime;

PUSH Trig
{
    WAIT(1000, FirstWait)
    }
    PRINT(“Wait 1 Triggered!\n”);
    }
}
PUSH ChangeWaitTime
{
RetimeWait(500, FirstWait);
}
```

In this example, when Trig is pushed, a 10-second event is scheduled. If ChangeWaitTime is activated while FirstWait is still running, the time will be reset to 5-seconds. If FirstWait has expired, no action will be taken.

Version:

SIMPL+ Version 1.00

Wait

Name:

Wait

Syntax:

```
Wait(INTEGER TIME[, NAME])
[{]
<statements>
[}]
```

Description:

NOTE: There is no semicolon after a WAIT statement because it has a clause or block following it.

Adds an event to a list to be executed in TIME hundredths of a second. Giving a WAIT a name is optional, but to cancel, pause, resume, or retime a wait, a name must be specified. A currently running WAIT will finish before being entered into the WAIT list again. For example, if in an endless WHILE loop, a second WAIT will only begin after the first finishes.

When the system encounters a WAIT, the event is put into the WAIT scheduler. The SIMPL+ module continues to execute without interruption. At some point, a task switch will occur (either due to event termination or other means, refer to "Task Switching" that begins on page 8). The WAIT schedule is checked by the operating system after a task switch, and if a wait event needs to be serviced, it is run and then terminates. Note that the module may task switch away while inside the WAIT, just like in other events.

AWAIT statement differs from a DELAY in both timing and order of statement execution. In a WAIT statement, the WAIT block executes only after the specified amount of time, but execution proceeds immediately to the statement following the WAIT block. In a DELAY, all execution is halted until the delay is finished.

Parameters:

TIME is an integer, expressed in hundredths of a second. For example, 525 specifies a wait time of 5.25 seconds.

NAME is an optional name given to the WAIT event. It has the same syntax as a variable name. Note that you cannot put two separate WAIT statements in the same SIMPL+ program that have the same NAME (this will cause a compilation error).

NOTE: (2-Series Only) The only variable types that are allowed to be used within a Wait Statement block are global variables and variables declared locally within the Wait Statement's block. Local variables declared within the function containing the Wait Statement are not allowed.

Example:

```
INTEGER WaitTime;
DIGITAL_INPUT StopVCR;
ANALOG_INPUT SysWait;
STRING_OUTPUT VCR$;

PUSH StopVCR
{
WAIT (SysWait, VCR_Stop)
{
VCR$ = “\x02STOP\x03”;
}
}
    FUNCTION MyFunc()
    {
       while ( 1 )
    {
          // statements (will keep executing during the wait
          statement)
          Wait( 500 )
          {
          // statements (execute once for each wait statement
          occurence)
          }
          // statements (will keep executing during the wait
          statement)
       }
 }
```

In this example, a VCR is triggered to go into STOP, but the STOP command is delayed based upon a time specified by an analog input to the SIMPL+ program.

Version:

SIMPL+ Version 3.00 - local variables are allowed within WAIT statements.

SIMPL+ Version 1.00

User Defined Functions

User Defined Functions Overview

A SIMPL+ program may have functions that are defined by users. Typically, a function is defined to modularize code to make repetitive tasks easier to perform or make code easier to read.

Function Definition

```
<function_type> <function_name> ([argument list])
{
<statements>
[RETURN <expression>;]
}
```

The following table demonstrates what <function_type> may be, what it means, and what data may be returned to the caller. Note that in all cases, a RETURN is not required, as the system defaults it to a shown specified value.

<function type>	**MEANING**	**RETURN VALUE**
FUNCTION	Returns no data to the caller.	No RETURN
INTEGER_FUNCTION	Returns an integer value to the caller.	INTEGER expression (default 0)
LONG_INTEGER_FUNCTION	Returns a long integer value to the caller.	LONG_INTEGER expression (default 0)
SIGNED_INTEGER_FUNCTION	Returns a signed integer value to the caller.	SIGNED_INTEGER expression (default 0)
SIGNED_LONG_INTEGER_FUNCTION	Returns a signed long integer value to the caller.	SIGNED_LONG_INTEGER expression (default 0)
STRING_FUNCTION	Returns a string value to the caller.	STRING expression (default "")

Function Parameters

NOTE: Passing STRINGs with BYVAL and BYREF is not allowed in the 2-Series Control System. All STRINGs are passed by referenced in the 2-Series Control System.

NOTE: Passing I/O datatype variables (DIGITAL_INPUT, ANALOG_INPUT and STRING_INPUT) is not allowed in the 2-Series Control System.

Functions may contain a list of parameters that are passed by the caller. Typically, data is passed to a function in order to make the code readable, maintainable, and less prone to bugs. SIMPL+ Version 1.00 did not allow data to be passed to functions. The only way to get data into functions was to declare global variables and have the functions reference the global variables.

The function argument list contains a comma separated list of arguments. The arguments are of the form:

```
[ByRef | ByVal] <INTEGER | LONG_INTEGER | SIGNED_INTEGER
| SIGNED_LONG_INTEGER | STRUCTURE> <variable_name>
```

ByRef and ByVal are keywords telling the system the read/write permissions and local behavior for the variable in the function. They are discussed in the next section.

INTEGER, STRING or STRUCTURE tells the function the<variable_name> type. BUFFER_INPUT, STRING_INPUT, STRING_OUTPUT, and STRING declarations are passed into a function using the STRING type. All other types are passed using the INTEGER type.

The function refers to the passed variable as <variable_name>. Note that for a one-dimensional array, the syntax is <variable_name>[] and for a two-dimensional array the syntax is <variable_name>[][].

ByRef, ByVal, ReadOnlyByRef

NOTE: Passing STRINGs with BYVAL an BYREF is not allowed in the 2-Series Control System. All STRINGs are passed by referenced in the 2-Series Control System.

NOTE: Passing I/O datatype variables (DIGITAL_INPUT, ANALOG_INPUT and STRING_INPUT) is not allowed in the 2-Series Control System.

Keyword Meanings

KEYWORD	MEANING
ByRef	Changes made to the variable that is passed to the function actually change the contents of the source variable. Note that any change made to the source variable will be reflected in the function. For example, if an INTEGER is passed ByRef and its state changes, the function will know about the change. It is typically more efficient to pass a variable by reference because space is not taken up by making local copies as with ByVal. Also referred to as “Pass by Reference”.
ByVal	The variable that is passed to the function has a local copy made of it. Changes made to the variable in the function are made on a local copy. The local copy is destroyed when the function terminates. The contents of this variable are a “snapshot” of the contents of the variable that was passed. Unlike Pass by Reference, changes made to the original variable that was passed to the function are not recognized in the function. When an expression is passed, it may only be passed by value since there is no source variable that the ByRef keyword may potentially modify. Also referred to as “Pass by Value”.
ReadOnlyByRef	This performs a Pass by Reference, identical to ByRef, but the compiler catches operations that write to the variable that has been passed. This would be typically be used if a DIGITAL_INPUT or other input type has been passed and which cannot be written. It is also used as a tool to catch unintentional writes to variables that have been passed.

If not specified in the function declaration, variables will be passed by reference if applicable. If the variable cannot be passed by reference (such as an element of an array), it will be passed by value. Any expression will always be passed by value.

The following table shows legal access methods for the basic data types when passed to a function.

R: Read access allowed. **W**: Write access allowed.

(E1): Generates a RunTime Error, not allowed to be write to INPUT values. The ReadOnlyByRef generates a compile error instead of a RunTime Error.

VARIABLE TYPE	**ByVal [LOCAL COPY]**	**ByRef [SOURCE]**	**ReadOnlyByRef [SOURCE]**
ANALOG_INPUT	R, W	R, (E1)	R
ANALOG_INPUT array	-	R, (E1)	R
ANALOG_INPUT array element	R,W	-	-
ANALOG_OUTPUT	R,W	-	R
ANALOG_OUTPUT array	-	-	R
ANALOG_OUTPUT array element	R, W	-	-
BUFFER_INPUT	R, W	R	R
BUFFER_INPUT array	-	R	R
BUFFER_INPUT array element	R, W	-	-
DIGITAL_INPUT	R, W	R, (E1)	R
DIGITAL_INPUT array	-	R, (E1)	R
DIGITAL_INPUT array element	R, W	-	-
DIGITAL_OUTPUT	R, W	-	R
DIGITAL_OUTPUT array	-	-	R
DIGITAL_OUTPUT array element	R, W	-	-
INTEGER	R, W	R, W	R
INTEGER array	-	R, W	R
INTEGER array element	R, W	-	-
LONG_INTEGER	R, W	R, W	R
LONG_INTEGER array	-	R, W	R
LONG_INTEGER array element	R, W	-	-
SIGNED_INTEGER	R, W	R, W	R
SIGNED_INTEGER array	-	R, W	R
SIGNED_INTEGER array element	R, W	-	-
SIGNED_LONG_INTEGER	R, W	R, W	R
SIGNED_LONG_INTEGER array	-	R, W	R
SIGNED_LONG_INTEGER array element	R, W	-	-
STRING	R, W	R, W	R
STRING array	-	R, W	R
STRING array element	R, W	-	-
STRING_INPUT	R, W	R	R
STRING_INPUT array	-	R	R
STRING_INPUT array element	R, W	-	-
STRING_OUTPUT	-	-	-

VARIABLE TYPE	ByVal [LOCAL COPY]	ByRef [SOURCE]	ReadOnlyByRef [SOURCE]
STRING_OUTPUT array	-	-	-
STRING_OUTPUT array element	-	-	-
STRUCTURE	-	R, W	R
STRUCTURE element (INTEGER)	R, W	-	-
STRUCTURE element (LONG_INTEGER)	R, W	-	-
STRUCTURE element (SIGNED_INTEGER)	R, W	-	-
STRUCTURE element (SIGNED_LONG_INTEGER)	R, W	-	-
STRUCTURE element (STRING)	-	-	-

R: Read access allowed. **W**: Write access allowed.

(E1): Generates a RunTime Error, not allowed to be write to INPUT values. The ReadOnlyByRef generates a compile error instead of a RunTime Error.

An example of a function declaration that has no parameters and returns no value would be:

```
FUNCTION PrintText()
{
// Code
}
```

The following is an example of a function declaration that takes an INTEGER and returns a STRING. The INTEGER is passed by value, so it cannot be modified.

NOTE: It is not strictly necessary to use the "ByVal" keyword here. ByVal can be used to make sure that no modifications to the original variable are done by accident within the function.

```
STRING_FUNCTION ComputeDate(ByVal INTEGER TheMonth)
{
STRING Month$[20];
// Code to compute Month$…
RETURN(Month$);
}
```

The following is an example of a function declaration that takes a STRING array and sorts it and an integer that takes the actual number of elements that are contained in the array. It returns an INTEGER error code:

```
INTEGER_FUNCTION SortNameInDatabase(STRING
Name[],INTEGER NumElements)
{
INTEGER Error;
// Code to sort Names[] and setError…
RETURN(Error);
}
```

Returning a Value

NOTE: A zero (0) message is automatically returned if no return statement is encountered.

The syntax for returning a value from integer and string functions is RETURN <expression>;. To return a value from a FUNCTION, PUSH, CHANGE, RELEASE or EVENT, the syntax is RETURN.

Integer functions include INTEGER_FUNCTION, SIGNED_INTEGER_FUNCTION, LONG_INTEGER_FUNCTION and SIGNED_LONG_INTEGER_FUNCTION. String functions include STRING_FUNCTION.

For Integer Functions, any valid integer expression is legal. For example:

```
RETURN (25);
RETURN (Z + MULDIV(A,B,C) + 100);
```

Are legal (assuming Z, A, B, C, are INTEGERs). If no RETURN statement is present in an integer, 0 is returned.

For a string function, any valid string is legal (string expressions are not allowed). For example:

```
STRING str[100];
RETURN "Hello!\n";
RETURN (str);
```

Are legal (assuming Z is an INTEGER and A$ is a STRING). If no RETURN statement is present in a STRING_FUNCTION, an empty string ("") is returned.

In SIMPL Version 3.00, the RETURN statement without arguments can be used in all functions that do not return strings or integers. For example:

```
INTEGER_FUNCTION MyIntegerFn ( )
{
IF (1)
{
RETURN (1);
}
RETURN (0);
}
LONG_INTEGER_FUNCTION MyLongIntFn ( )
{
IF (1)
{
SIGNED_INTEGER_FUNCTION MySignedIntFn ( )
{
IF (1)
{
RETURN (1);
}
RETURN (0);
}
SIGNED_LONG_ INTEGER_FUNCTION MySignedLongIntFn ( )
```

```
{
IF (1)
{
RETURN (1);
}
RETURN (0);
}
STRING_FUNCTION MyStringFn ( )
{
IF (1)
{
RETURN (“abc”);
}
RETURN (“def”);
}
FUNCTION MyFn ( )
{
IF (1)
{
return;
}
}
EVENT
{
if (1)
return;
}
PUSH
{
if (1)
return;
}
RELEASE
{
if (1)
return;
}
CHANGE
{
if (1)
return;
}
```

Calling a Function

When calling a function where the return is being used, the syntax is:

```
<variable> = <function_name>([argument_list]);
```

For example,

```
INTEGER X, ANALOG1, ANALOG2;
STRING Q$[20], B$[50];
X = ComputeChecksum(Analog1, Analog2);
Q$ = DoSomething(Analog1, B$);
```

Are legal.

If the return is not going to be used, or there is no return (in the case of a FUNCTION), the syntax is:

```
<CALL> <function_name>([argument_list]);
```

For example,

```
CALL DoSomethingElse();  (X-Generation or 2-Series)
```

or

```
DoSomethingElse() ;   (2-Series only)
```

NOTE: (X-Generation only) Functions do not support recursion, i.e. the code in the body of a function may not contain a call to that function.

NOTE: The keyword, CALL, is required in the X-Generation control system, and optional in the 2-Series control system. CALL may not be used when calling non-user defined functions. For example:

```
CALL Print( str, “abc” );   // Illegal
```

Function Libraries

NOTE: A function may be placed in the same body of code as the caller. In some cases, the same function needs to be used across several different modules. Although the code could be rewritten in all modules (as was the case with SIMPL+ Version 1.00), SIMPL+ Version 2.00 supports function libraries.

A function library is simply a group of functions in a SIMPL+ file. The file is saved as a SIMPL+ Library File (*.USL), from the Save As dialog in the SIMPL+ editor.

In order to include a function library, the #CRESTRON_LIBRARY or #USER_LIBRARY directives are used. The libraries are searched in the order they are included, in case a function name is used in more than one library. The first function found is used. Refer to #CRESTRON_LIBRARY and #USER_LIBRARY for more information.

Program Structure

When a new SIMPL+ program is created, a template is provided that lists the order in which constructs and statements should be defined. Sections can be uncommented and expanded out to implement the desired code.

A SIMPL+ program layout would consist of, in order:

1. Compiler Directives
2. Input/Output definitions From/To a SIMPL Program
3. Global declarations for the module, including STRING, INTEGER, arrays, structures, etc.
4. FUNCTION declarations
5. PUSH/RELEASE/CHANGE statements
6. FUNCTION MAIN

NOTE: All of these are not mandatory and may be left out as needed.

NOTE: In SIMPL+ Version 3.00, local variables are allowed.

Forward references are not allowed in a SIMPL+ program. This means you cannot CALL a function before it has been defined. This is the reason FUNCTION declarations are placed before other code. If function A calls function B, then function B should be located first in the source file.

FUNCTION MAIN is a special case function. It is not required, but any code present between the { and } is executed at startup. This is typically used for initialization purposes.

Example:

```
FUNCTION MAIN()
{
MyVar=0;
For(I=1 to 10)
    B[I] = I;
}
```

Sometimes function MAIN() contains an endless loop with a DELAY statement that executes periodically while the program runs.

Common Runtime Errors

Common Runtime Errors Overview

The following errors will occur at runtime. In order for these error messages to be seen, the Crestron Viewport must be open and communications with the control system (via Ethernet or the computer port) must be established.

Array out of bounds

An attempt was made to access an element of an array that is outside the declared range of the array. For an array size declaration, the allowable indices are 0 through the declared size. For example, INTEGER X[10][10] would allow access to X[0][0] through X[10][10].

Bad printf format

The MAKESTRING or PRINT functions have encountered an invalid character following the % character. The most common reason for this is when a % is actually required, %% should be used to print a single % character. Refer to MAKESTRING and PRINT for a full list of valid format specifiers.

Full Stack

The SWITCH construct may only have 32 CASE statements in SIMPL+ Version 1.00. If more than 32 are used, this error appears.

Library not found

This occurs when a module tries to call a user-defined function that exists in an external library which was specified with #CRESTRON_LIBRARY or #USER_LIBRARY. During compilation, the compiler builds a file containing the libraries to send to the control system. Typically, this could be caused by a transfer error which would be seen at load time.

Rstack overflow

The Rstack that this message refers to is the Return Stack. When an event is interrupted by some means (via a process_logic statement or an implied task switch from inside a loop), information about that event is put on the Return stack, so that when the event resumes, it knows how to continue. When the event continues, the saved information is removed from the return stack.

If during this interruption the event is called again, and interrupted again, more information is saved on the return stack. The return stack is of limited size and if this keeps occurring, the Return stack will not have enough space to contain more data and this message will be issued.

For a further discussion of how to handle the programming when events are interrupted, refer to “Task Switching” on page 22.

Scheduler is full

Any time-based function such as DELAY, PULSE, or WAIT will schedule an event in SIMPL+. A scheduled event will add one or more entries to the SIMPL+ scheduler. The scheduler currently supports 200 events and is global to the entire SIMPL+ system. If the scheduler is full and another event is added, this message is issued.

NOTE: The message “Skedder Full” is issued from a SIMPL program, not SIMPL+. “Skedder full” is a similar problem, but results if too many time-based events are occurring in a SIMPL program.

String array out of bounds

An attempt was made to access an element of a string array that is outside the declared range of the array. Remember that for an array size declaration, the allowable indices are 0 through the declared size. For example, STRING X$[5][20] declares six strings of 20 bytes each, accessed via X$[0] through X$[5].

Too much ram allocated

Too much RAM was allocated for the data structures. Approximately 60K is available for user data. When compiling a program, it will tell you how much memory is required for one instance of the module. Each instantiation of the module in a SIMPL program takes up that much more space. For example, if a module says it requires 100 bytes after it is compiled, two instances of that module will require 200 bytes. If this message is received, reduce the number of variables. If string or buffers have been declared overly large, this is an easy place to reduce memory requirements.

Example Programs

Example 1: Hello, World!

```
// A digital input from the SIMPL program DIGITAL_INPUT
TRIG;

// Upon the digital signal TRIG going high or low, the Hello,
// World! message is printed.

CHANGE TRIG
{
PRINT(“Hello, World!\n”);
}

// Main is only called once when the system starts up or is
reset.
FUNCTION MAIN()
{
PRINT(“Main Starts!\n”);
}
```

Example 2: 8-Level switch on a Pesa switcher

```
#SYMBOL_NAME “Pesa Switcher - 8 Levels”
#HINT “Creates Pesa CPU-Link H command for Switching”

/
*************************************************************
******
DIGITAL, ANALOG and SERIAL INPUTS and OUTPUTS

*************************************************************
******/
// Digital trigger from the SIMPL program - this sends the
command
// string out.
DIGITAL_INPUT TRIG;
// Analogs for the output and 8 levels of the switcher from the
// SIMPL program.
ANALOG_INPUT

OUTPUT,LEVEL1,LEVEL2,LEVEL3,LEVEL4,LEVEL5,LEVEL6,LEVEL7,LEVE
L8;
// The output string that is to be sent from the SIMPL+ program
to
// the SIMPL program to the switcher.
STRING_OUTPUT COMMAND$;

/
*************************************************************
Global Variables
(Uncomment and declare global variables as needed)

*************************************************************
*/
INTEGER I, COUNT, CKSLOW, CKSHI;
STRING PESABUF[30];

/
*************************************************************
Event Handlers
(Uncomment and declare additional event handlers as needed)

*************************************************************
*/
PUSH TRIG
{
// Format command which stores the switcher command in a
// temporary buffer. A Command looks like
H{out}{l1}{l2}...{l8}
// {2 byte checksum}{CR}{LF} where {out} and {l1}..{l8} are 3
// digit ASCII bytes with leading zeros. An example is
// H001001002003004005006007008{2 bytes checksum}{CR}{LF}
//
makestring(PESABUF,”H%03d%03d%03d%03d%03d%03d%03d%03d%03d”,
// OUTPUT, LEVEL1, LEVEL2, LEVEL3, LEVEL4, LEVEL5, LEVEL6,
// LEVEL7, LEVEL8);

COUNT=0;  // Checksum count initialized to 0.
// Add each byte in the string to the running count.
for(i=1 to len(pesabuf))
COUNT = COUNT + BYTE(PESABUF, I);

// The checksum is computed by taking the COUNT and throwing
// away all but the lower byte. The upper nibble + '0' is the
// high order checksum byte, the lower nibble + '0' is the low
// order checksum byte.

// Compute the low byte of the checksum.
CKSLOW = (COUNT & 0x0F) + '0';
// Compute the high byte of the checksum.
CKSHI = ((COUNT & 0xF0) >> 4) + '0';

// Send the checksum command to the COMMAND$ that gets routed
// to the switcher via the SIMPL program.
makestring(COMMAND$, “%s%s%s”, PESABUF, CHR(CKSLOW),
CHR(CKSHI));
}
```

Example 3: Computing the Number of Days in a Month (Using Functions)

```
#SYMBOL_NAME “Compute Number of Days in a Month”
#ANALOG_INPUT MONTH;
#ANALOG_OUTPUT DAYS;

INTEGER_FUNCTION ComputeDaysInMonth(INTEGER Month)
{
// Note that this computation does NOT take into account leap
// year!
INTEGER Days;

SWITCH (Month)
{
CASE( 2): Days = 28; // February
CASE( 4): Days = 30; // April
CASE( 6): Days = 30; // June
CASE( 9): Days = 30; // September
CASE(11): Days = 30; // November
Default:  Days = 31;  // All others
}
Return(Days);
}

CHANGE MONTH
{
DAYS = ComputeDaysInMonth(MONTH);
}
```

Example 4: Computing the Number of Days in a Month (Using Function Libraries)

The following code would be saved as, in this example, “My Function Library.USL”.

```
INTEGER_FUNCTION ComputeDaysInMonth(INTEGER Month)
{
// Note that this computation does NOT take into account leap
// year!
INTEGER Days;

SWITCH(Month)
{
CASE( 2): Days = 28; // February
CASE( 4): Days = 30; // April
CASE( 6): Days = 30; // June
CASE( 9): Days = 30; // September
CASE(11): Days = 30; // November
Default:  Days = 31;  // All others
}
Return(Days);
}
```

The following code can be saved as any filename:

```
#SYMBOL_NAME “Compute Number of Days in a Month”
#USER_LIBRARY “My Function Library”
#ANALOG_INPUT MONTH;
#ANALOG_OUTPUT DAYS;
CHANGE MONTH
{
DAYS = ComputeDaysInMonth(MONTH);
}
```

File Time and Date Functions Overview

These versions of the Time and Date functions in a given SIMPL+ program are used to retrieve information about the current date and time from the file info structure returned from FINDFIRST/FINDNEXT. Values can be retrieved as text strings i.e. “January” or integer values. Typically, integer values are used if computations need to be performed (i.e. when the date is the 25th, perform a specific action).

WriteLongIntegerArray

Name:

WriteLongIntegerArray

Syntax:

```
SIGNED_INTEGER WriteLongIntegerArray ( INTEGER
file_handle,
LONG_INTEGER ilArray[m][n] )
```

Description:

Writes the array from a file starting at the current file position. Two bytes are written, most significant first containing the row dimension of the array, then two more bytes are written, containing the column dimension of the array. Then each long integer is written as a four byte quantity, most significant byte first. The integers are stored in row-major order, e.g. all the elements of row 0 first, then the elements of row 1, etc. Note that there is one more row and one more column than the dimensions that are written, because there is a row 0 and a column 0. Refer to the section entitled “Reading and Writing Data to a File” on page 118 for a discussion of when to use this function and when to use the related functions: FileWrite, WriteInteger, WriteString, WriteStructure, WriteSignedInteger, WriteLongInteger, WriteLongSignedInteger, WriteIntegerArray, WriteSignedIntegerArray, WriteLongIntegerArray, WriteLongSignedIntegerArray, WriteStringArray.

Parameters:

FILE_HANDLE specifies the file handle of the previously opened file (from FileOpen).

ilArray is the array whose values are Write.

Return Value:

Number of bytes written to the file. If the return value is negative, it is an error code.

Example:

(Refer to "File Functions Overview"on page 116)

```
INTEGER  nFileHandle, iErrorCode;
LONG_INTEGER ilArray[10];
nFileHandle = FileOpen ( “MyFile”, _O_RDONLY );
IF (nFileHandle >= 0)
{
iErrorCode = WriteLongIntegerArray(nFileHandle, ilArray);
if (iErrorCode > 0)
PRINT ( “Array written to file correctly.\n”);
else
PRINT ( “Error code %d\n”, iErrorCode);
}
```

Version:

SIMPL+ Version 3.01 or higher (Pro 2 only)

Control System:

2-Series Only

Compiler Errors and Warnings

Compiler Errors and Warnings Overview

The SIMPL+ program compiler errors and warnings are grouped into several categories, as shown in the following table. Errors are listed in numerical order; page links are provided to detailed descriptions of the errors.

Compiler Errors and Warnings

CATEGORY	NUMBER	MESSAGE TEXT	PAGE
Syntax Errors	1000	'<identifier>' already defined	page 289
	1001	Undefined variable: '<identifier>' Undefined function '<identifier>'	page 290
	1002	Missing '<token>'	page 292
	1003	Incorrect type '<decl_type>', expected type(s): '<decl_type1[,decl_type2][,decl_typen]>' Incorrect type, expected type(s): '<decl_type1[,decl_type2][,decl_typen]>'	page 293
	1004	Unmatched symbol: '<identifier>'	page 293
	1005	Unexpected symbol in compiler directive: '<identifier>'	page 294
	1006	Invalid #DEFINE_CONSTANT value: '<identifier>'	page 294
	1007	Missing array index: '<identifier>'	page 295
	1008	Invalid integer argument or undefined variable: '<identifier>'	page 296
	1009	Missing structure member: '<identifier>' Structure does not contain member: '<identifier>'	page 297
	1010	Symbol Name contains illegal character: ';'	page 298
	1011	Missing return value	page 298
	1012	Unterminated string constant	page 299
	1013	Source code does not evaluate to anything	page 299
Fatal Errors	1100	Statement outside of function scope	page 300
	1101	Abort - Error count exceeded <max_errors>	page 301
Expression Errors	1200	Invalid numeric expression: '<expression>' Invalid string expression Invalid expression: '<expression>'	page 301
	1201	Invalid \\x sequence Invalid \\x sequence: '<expression>'	page 303

CATEGORY	NUMBER	MESSAGE TEXT	PAGE
Declaration Errors	1300	Array size missing Array size invalid	page 304
	1301	Invalid array index	page 305
	1302	Variable name, ‘<identifier>’, exceeds maximum length of <max> characters	page 306
	1303	Declaration type not allowed within structure: '<identifier>' Structure cannot contain String Arrays or Structure variables: Structure definitions not allowed within other structures Local Structure declarations are not allowed	page 307
	1304	Local variables must be declared at top of function	page 308
	1305	Local functions not supported	page 308
	1306	Declaration type can only be used globally: '<identifier>'	page 309
	1307	Variables must be declared before array declarations: '<identifier>'	page 310
	1308	Global variable declaration cannot be declared in library file: '<identifier>' I/O Declaration cannot be declared in library file: '<identifier>'	page 311
	1309	Compiler Directive must be set before all global variable declarations #DEFAULT_NONVOLATILE Compiler Directive already set #DEFAULT_VOLATILE Compiler Directive already set	page 312
	1310	Compiler directive cannot be in function scope	page 313
	1311	Undefined Wait Label: '<identifier>' Missing, invalid or already defined Wait label: '<identifier>'	page 314
	1312	Array boundary exceeded maximum size of ‘num_bytes’ bytes	page 315
	1313	Minimum array size invalid	page 315
	1314	Minimum array size is not allowed for this datatype: '<identifier>' Minimum array size for this datatype has already been declared: '<identifier>	page 316
Assignment Errors	1400	Illegal Assignment	page 317
	1401	Variable cannot be used for assignment: '<identifier>'	page 318
	1402	Variable can only be used for assignment: '<identifier>'	page 318
Function Argument Errors	1500	Argument <arg_num> cannot be passed by reference	page 319
	1501	Argument <arg_num> cannot be passed by value	page 320
	1502	Function contains incomplete number of arguments Function call contains an unmatched number of arguments	page 321
	1503	Input or Output signal expected: '<identifier>'	page 321
	1504	Incomplete number of format string arguments Format string contains an unmatched number of arguments Argument <arg_num> is missing or invalid. Argument <arg_num> is missing or invalid. <decl_type> expected	page 322
	1505	Format string contains invalid format specifier	page 323
	1506	0, 1 or 2 constant expected for argument 1	page 324
	1507	Argument <arg_num>: Missing or invalid array	page 324
	1508	I/O variable cannot be passed to read file functions: '<identifier>'	page 325

CATEGORY	NUMBER	MESSAGE TEXT	PAGE
Construct Errors	1600	'Function Main' cannot contain function parameters 'Function Main' cannot return a value	page 326
	1601	Duplicate CASE Statement Constant expected: '<identifier>'	page 326
	1602	Switch statement contains 'default' without 'case' labels	page 327
	1603	#CATEGORY does not exist: '<categorgy_number>' Defaulting to Category Type, ""32"" (Miscellaneous).	page 328
	1604	'EVENT' already has a body	page 329
	1605	Function can only be contained within an event	page 329
	1606	Statement must be contained within a loop statement	page 330
	1607	GetLastModifiedArrayIndex may return an ambiguous signal index	page 331
	1608	Missing library file name	page 331
File Errors	1700	End of file reached	page 332
	1701	Error writing header file: '<file_name>' Error writing file: '<file_name>' Error writing library file Error writing output file Error creating compiler makefile: '<file_name>' Error opening compiler source makefile: '<file_name>' Error opening source file: '<file_name>'	page 332
	1702	Error extracting library, '<file_name>', from archive: '<archive_file>'	page 332
Complier Warnings	1800	'Return' statement will only terminate current Wait statement's function scope	page 333
	1801	'TerminateEvent' statement will only terminate current Wait statement's function scope	page 333
	1802	#CATEGORY_NAME defined more than once. Using: #CATEGORY_NAME "<number>"	page 334
	1803	Possible data loss: LONG_INTEGER to INTEGER assignment	page 335

Syntax Errors (Compiler Errors 1000 to 1013)

Compiler Error 1000

syntax error: '<identifier>' already defined

The specified identifier was declared more than once. A variable can only be declared once within it's function scope. The same identifier cannot be used for more than one declaration type.

Scope refers to the level at which an Event, user-defined function or statement resides. Having a global scope means that the function or variable can be called or accessed from anywhere within the program. A local scope means that the variable can only be accessed from within the event or function that it resides in.

NOTE: Make sure the identifier has not been declared as another declaration type, user-defined function, or structure definition.

The following are examples of this error:

```
INTEGER i;
INTEGER i;       // error – i is already defined as an INTEGER
STRING i[100];    // error – i is already defined as an INTEGER

STRUCTURE myStruct
{
   INTEGER i;     // ok – i is a member variable of myStruct
}

INTEGER_FUNCTION MyFunc( INTEGER x, INTEGER y )
{
   INTEGER i;     // ok
   INTEGER i;     // error - i is already defined as a local
INTEGER
   INTEGER x;    // error – x is already defined as a function
                 //         parameter, which makes it a local
                  //         variable in this function
}

FUNCTION MyFunc() // error – MyFunc() is already defined
                  //         as an INTEGER_FUNCTION
{
}

FUNCTION AnotherFunc( INTEGER x, INTEGER y  ) // ok – x and y
are
                                     // local to this function
{
}
```

Compiler Error 1001

syntax error: Undefined variable: '<identifier>' *Undefined function '<identifier>'*

The specified identifier was not declared.

All variables and user-defined functions must be declared before they are used. They must be declared either globally or within the same function scope. Variables from one program are not accessible from another program.

Scope refers to the level at which an Event, user-defined function or statement resides. Having a global scope means that the function or variable can be called or accessed from anywhere within the program. A local scope means that the variable can only be accessed from within the event or function that it resides in.

- Make sure the identifier is spelled correctly
- Make sure the identifier has not been declared locally within another function
- When using structures, make sure the proper 'dot' notation is being used when accessing the structure's variables (see example below)

The following are examples of this error:

```
INTEGER i;

STRUCTURE myStruct
{
   INTEGER structMember;
   INTEGER structArrMember[10];
}

myStruct struct;
myStruct structArr[10];

FUNCTION MyFunc( INTEGER x )
{
   INTEGER k;

   i = 1;                              // ok
   k = 3;                              // ok
   x = 4;                              // ok
   struct.structMember = 5;            // ok – proper 'dot'
notation
   struct.structMember[1] = 6;         // ok – proper 'dot'
notation
   structArr[1].structMember = 7;      // ok – proper 'dot'
notation
```

```
   structArr[1].structArrMember[2] = 8; // ok – proper ‘dot’
notation

   j = 2;                  // error – j is not declared
   structMember = 10;      // error – improper ‘dot’ notation
   structMember[1] = 11;   // error – structMember is not an
array

   k = AnotherFunc();      // error – AnotherFunc() was not
                           //         declared previously
}

INTEGER_FUNCTION AnotherFunc()
{
   k = 5;           // error – k is a local variable of MyFunc()
   x = 6;           // error – x is a local variable of MyFunc()

   Call MyFunc();     // ok
   Call MyFunk();     // error – spelling error

   return (1);
}
```

Compiler Error 1002

syntax error: Missing '<token>'

A language element was expected and not found. The compiler expects certain language elements to appear before or after other elements. If any other language element is used, the compiler cannot understand the statement. Examples are missing parenthesis after a function call, missing semicolons after a statement and missing braces when defining functions.

A token is a language element such as a keyword or operator. Anything that is not whitespace (i.e.: spaces, tabs, line feeds and comments) is a token.

Examine the last uncommented non-blank line or statement within the program. If a token was required in a previous statement and was not encountered, the compiler will continue onto the next line and mark the first token of the new statement as the error.

The following are examples of this error:

```
STRUCTURE MyStruct
{
   INTEGER x;
   STRING s[100];
}

MyStruct struct;         // error – missing ‘;’ from preceding
                          //         structure definition

INTEGER_FUNCTION MyFunc( INTEGER ) // error – argument 
variable
                                   //         not specified

   INTEGER x;            // error – ‘{’ missing before INTEGER

   Print “abc”;           // error – missing parenthesis
                          //         should be Print (“abc” );

   // printing…
   Print “def”          // error – error message will occur on
                          //         next statement

   // more printing…
   Print “ghi”;          // error – missing ‘;’from preceding 
Print
                          //         statement

   x = ((1+2) + 3;      // error – unmatched set of parentheses

   x = atoi( “abc”, 1 ); // error – atoi() does not take 2 
arguments

   if ( x = 4 )
      return 5;           // error – should be return (5);

   return (6);            // ok
}
```

Compiler Error 1003

syntax error: Incorrect type '<decl_type>', expected type(s): '<decl_type1[,decl_type2] [,decl_typen]>' Incorrect type, expected type(s): '<decl_type1[,decl_type2][,decl_typen]>'

A specific variable or type was expected and not found. Examples are variables of one type being used in place of another, and incorrect variable types within function arguments.

The following are examples of this error:

```
STRING_FUNCTION MyFunc( INTEGER x )
{
   INTEGER y;

   x = getc( y );   // error – y is not of type STRING
   x = MyFunc( 1 ); // error – x cannot accept the resulting
string
                    //         returned from MyFunc()
}
```

Compiler Error 1004

syntax error: Unmatched symbol: '<identifier>'

Some language constructs are composed of more than one keyword. In these cases, each keyword may require statements before and after it is used.

For example, the Switch statement uses the following keywords, Switch, Case, and Default. If the keyword, Case, is encountered before or outside of switch statement, this error will result.

The following are examples of this error:

```
FUNCTION MyFunc( INTEGER x )
{
   x = 1;

   while ( 1 )
   {
      x = x + 1;

   } until ( x > 5 ); // error – ‘until’ is not part of the
                      //         ‘while’ construct

   else               // error – no preceding ‘if’ statement
   {
      x = 0;
   }
}
```

Compiler Error 1005

syntax error: Unexpected symbol in compiler directive: '<identifier>'

An invalid identifier is following a compiler directive.

The following are examples of this error:

```
#DEFINE_CONSTANT   MyIntConst   100  // ok
#DEFINE_CONSTANT  “MyIntConst”  100  // error – MyIntConst
should not
                                    //       be in quotes – this
                                    //       will be evaluated as
                                       //        a literal string
```

Compiler Error 1006

syntax error: Invalid #DEFINE_CONSTANT value: '<identifier>'

The value for a #DEFINE_CONSTANT compiler directive must be either a literal string or an integer value. Expressions, variables, functions and events cannot be specified as the compiler directive’s value.

The following are examples of this error:

```
INTEGER x;

#DEFINE_CONSTANT  MyIntConst    100       // ok
#DEFINE_CONSTANT  MyStrConst    “abc”     // ok

#DEFINE_CONSTANT  MyExprConst  (1+2)    // error – expressions
are
                                        //        not allowed

#DEFINE_CONSTANT  MyVarConst    x         // error –
substitutions are
                                        //        not allowed

#DEFINE_CONSTANT  MyExprConst   (x+1)     // error – macros
are not
                                          //        supported

#DEFINE_CONSTANT  MyFuncConst   myFunc    // error
#DEFINE_CONSTANT  MyFuncConst   getc      // error
```

Compiler Error 1007

syntax error: Missing array index: '<identifier>'

A variable declared as an array is being used within an expression without the array index being specified. For two-dimensional arrays, both indices must be specified. When passing entire arrays as function arguments, no index is needed.

The following are examples of this error:

```
FUNCTION MyFunc()
{
   INTEGER i, arr[10], arr2[10][20];
   STRING str[100], str2[100][50];

   i = arr[5];                         // ok
   i = arr2[5][10];                    // ok
   arr[5] = arr2[5][10];               // ok
   arr2[5][10] = 5;                    // ok

   i = arr;                        // error – no index specified
   arr = 5;                        // error – no index specified
   i = arr2[5];                        // error – 2nd index not
specified

   str2[5] = “a”;                      // ok
  str[5] = “a”;                  // error – ‘str’ is not an array
}
```

Compiler Error 1008

syntax error: Invalid integer argument or undefined variable: '<identifier>'

The construct being used requires either an integer value or variable passed as a function argument.

- Make sure the variable has been declared

The following are examples of this error:

```
STRUCTURE MyStruct
{
   INTEGER x;
   STRING s[100];
}

MyStruct struct;

Function MyFunc()
{
   INTEGER i;
   STRING s[100];

   for ( i = 1 to 10 )        // ok
   {
     for ( j = 1 to 5 )     // error – ‘j’ has not been declared
      {
        x = j;              // error – should be struct.x = j;
      }

     for ( s = “a” to “z” )  // error – strings are not allowed
      {
      }
   }
}
```

Compiler Error 1009

syntax error: Missing structure member: '<identifier>'
Structure does not contain member: '<identifier>'

Variables contained within structures are required when using structures within an expression or statement. When using structures, the 'dot' notation is required to specify a structure's variable.

The notation is as follows: <structure_name>.<member_variable>
Structure arrays are as follows: <structure_name>[index].<member_variable>

The following are examples of this error:

```
STRUCTURE MyStruct
{
   INTEGER x;
   INTEGER x2[10];
}

Function MyFunc( INTEGER x )
{
   INTEGER i;
   MyStruct struct;
   MyStruct structArr[10];

   i = struct.x;                  // ok
   struct.x = 5;                  // ok
   struct.x2[2] = 5;              // ok
   structArr[1].x2[2] = 5;        // ok

   Call MyFunc( i );              // ok
   Call MyFunc( struct.x );       // ok
   Call MyFunc( structArr[1].x ); // ok
   Call MyFunc( struct.x2[1] );   // ok

   i = struct;             // error – structure variable not
specified
   struct = i;             // error – structure variable not
specified
   Call MyFunc( struct );  // error – structure variable not
specified

   i = struct.z;          // error – structure variable does
not exist
   struct.z = 5;          // error – structure variable does
not exist
}
```

Compiler Error 1010

syntax error: Symbol Name contains illegal character: ';'

The compiler directive, #SYMBOL_NAME, cannot contain a semicolon as part of the symbol name.

The following are examples of this error:

```
#SYMBOL_NAME “MySymbol”             // ok
#SYMBOL_NAME “My Symbol”            // ok
#SYMBOL_NAME “MySymbol;YourSymbol”  // error
```

Compiler Error 1011

syntax error: Missing return value

The Return statement requires a valid value or expression when used inside of functions that return a value (INTEGER_FUNCTION, STRING_FUNCTION, etc.). The Return statement is available for functions that don’t return a value (FUNCTION), but do not allow values to be returned.

The following are examples of this error:

```
FUNCTION MyFunc( INTEGER x )
{
   if ( x=1 )
      return;      // ok – MyFunc() does not return a value

  return (5);    // error – MyFunc is declared as FUNCTION and
                   //         cannot return a value
}

INTEGER_FUNCTION AnotherFunc( INTEGER x )
{
   if ( x=1 )
      return;      // error – MyFunc is declared as an
INTEGER_FUNCTION
                   //         and must return a value
   else if ( x=2 )
      return (5);  // ok

   else if ( x=3 )
      return ();   // error – no value or expression is given

   return (x);     // ok
}
```

Compiler Error 1012

syntax error: Unterminated string constant

A literal string was used and was not contained within quotes. If a quotation character is needed within a literal string, a backslash should be placed before the quotation character (i.e.: \). This will indicate to the compiler that the quotation character is not the terminating quote for the literal string.

The following are examples of this error:

```
FUNCTION MyFunc()
{
   Print( "%s", "abc\"" );  // ok
  Print( "%s", "abc\" );   // error - \" is not a closing quote
}
```

Compiler Error 1013

syntax error: Source code does not evaluate to anything

A statement must perform an action in order to be valid. If no action is specified, the statement will not be useful to the program.

The following are examples of this error:

```
FUNCTION MyFunc()
{
   INTTEGER x;
   STRING str[100];

   x = 5;        // ok
   str = “abc”; // ok

   x;            // error
   str;          // error
}
```

Fatal Errors (Compiler Errors 1100 to 1101)

Compiler Error 1100

fatal error: Statement outside of function scope

User-defined functions, Events, and compiler directives can only be defined at a global level.

Scope refers to the level at which an Event, user-defined function or statement resides. Having a global scope means that the function or variable can be called or accessed from anywhere within the program. A local scope means that the variable can only be accessed from within the event or function that it resides in.

Variables can have either a global or local scope.

The following are examples of this error:

```
INTEGER i;
STRING str[100];

#DEFINE_CONSTANT  myConst  1  // ok
#DEFINE_CONSTANT  myConst  2; // error – semicolon is not
needed

i = 5;         // error – variables can only be used within a
               // function or event
Call MyFunc(); // error – functions can only be called from
               //         another function or event

;             // error – a semicolon is valid statement (which
               //         does nothing), and is not contained
               //         within a function or event

{              // error – braces only signify a group of
               // statements within a function or
               // construct (i.e.: if-else, while, etc)
   INTEGER x;
   INTEGER y;
}

Print( “outside of everything” ); // error – statement is
                                  // not contained within
                                  // a function or event

FUNCTION MyFunc()                 // ok
{
}
```

```
Function Main()     // ok – Function Main gets called
automatically
                    //      at the start of the program
{
   i = 5;           // ok
   str = “”;        // ok

   Call MyFunc();   // ok
}
```

Compiler Error 1101

fatal error: Abort - Error count exceeded <max_errors>

When compiling, if the error count is too large, the compiler will terminate the compile process prematurely. This can not only be a tremendous time saver, but also help reduce the aggravation and stress levels of the programmer.

Expression Error (Compiler Errors 1200 to 1201)

Compiler Error 1200

expression error: Invalid numeric expression: '<expression>'
Invalid string expression
Invalid expression: '<expression>'

Expressions can be calculations, comparisons, or the validity of a value from a string or numeric variable or value. All expressions require that all variables and values within the equation are of the same type. For example, you cannot add or compare an integer and a string together. The result of a comparison (i.e.: abc = def) is always a numeric value and will be treated as a numeric expression.

The following are examples of this error:

```
INTEGER x, y;
STRING str[100];

INTEGER_FUNCTION myFunc( INTEGER i )
{
   x = (1 + 2);                          // ok

   if ( x > y )                          // ok
   {
      if ( i )                           // ok
      {
         if ( str = “abc” )              // ok
         {
            while ( 1 )                  // ok
            {
               x = x + y + myFunc(1);    // ok
               break;
            }
         }
      }
   }

   return (1);
}

INTEGER_FUNCTION AnotherFunc( INTEGER i )
{
  x = (1 + str);              // error – cannot add an integer
                                  //         and string

   if ( x > "abc" )              // error – cannot compare an 
integer
                                  //         and string
   {
      if ( str )                  // error – cannot check the 
validity
                                  //         of a string
      {
        if ( str = MyFunc(1) )   // error – cannot add strings
                              //        and integers together
         {
            while ( str < "abc" ) // ok
            {
             x = (x + );       // error – incomplete expression
               break;
            }
         }
      }
   }

   return (1);
}
```

Compiler Error 1201

expression error: Invalid \\x sequence
Invalid \\x sequence: '<expression>'

A hexadecimal sequence within a literal string contained an invalid format. Characters represented by a hexadecimal number must follow the format: \xXX, where '\x' signifies that a hexadecimal sequence is to follow and XX is the 2 digit hexadecimal value.

The following are examples of this error:

```
Function myFunc()
{
   STRING str[100];

   MakeString( str, “Sending commands \xFF” );         // ok
   MakeString( str, “Sending commands \x41\x1A\xFF” ); // ok

   MakeString( str, “Sending cmd \x4” );  // error – 2 digits 
required
   MakeString( str, “Sending cmd \x” );   // error – hex code 
expected
   MakeString( str, “Sending cmd \xZZ” ); // error – invalid 
hex code
   MakeString( str, “Sending cmd \xZZ” ); // error – invalid 
hex code
}
```

Declaration Errors (Compiler Errors 1300 to 1312)

Compiler Error 1300

declaration error: Array size missing
Array size invalid

STRING, STRING_INPUT and BUFFER_INPUT variables require a valid length. A length is specified by number enclosed within brackets. Arrays for these datatypes are specified by the first set of brackets containing the number of strings and the second set of brackets containing the total length for each string. Two-dimensional arrays are not allowed for these datatypes.

In a function's argument list, since all strings are passed by reference, no array size is necessary. A string array is indicated by an empty set of brackets. See example below.

The following are examples of this error:

```
#DEFINE_CONSTANT   ARR_SIZE   100

STRING str[100];                // ok – str has a length of 100
STRING_INPUT strIn[ARR_SIZE];    // ok – strIn has a length of
100
BUFFER_INPUT bufIn[ARR_SIZE];    // ok – bufIn has a length of
100

STRING strArr[50][100];        // ok – 51 strings of length 100
STRING_INPUT strIn[50][100];     // ok - 51 strings of length
100
BUFFER_INPUT bufIn[50][100];     // ok – 51 strings of length
100

STRING_OUTPUT strOut;            // ok – STRING_OUTPUTs do not
                                  //      require a length
STRING_OUTPUT strOutArr[10];      // ok – array of 10
STRING_OUTPUTs

STRING str2;                     // error – no length specified
STRING_INPUT strIn;              // error – no length specified
BUFFER_INPUT bufIn;              // error – no length specified
STRING_OUTPUT strOutArr[10][20]; // error – 2-D arrays not
supported
STRING str[x];                    // error – variables are not
allowed
STRING str[myFunc()];             // error – function calls are
                                  //         not allowed

FUNCTION myFunc( STRING sArg,       // ok – strings are passed
by
               STRING sArgArr[] ) //      reference. sArg is a
                                 //      string and sArgArr is a
                                      //      string array
{
}

FUNCTION myFunc2( STRING sArg[10],     // error – size is not
allowed
               STRING sArgArr[][] ) // error – 2-D strings not
                                       //        supported
{
}
```

Compiler Error 1301

declaration error: Invalid array index

An index is required when accessing any element of an array. Two dimensional arrays require both indices of the array to be specified. This index must be a valid numeric expression.

All arrays are passed to functions by reference, so specifying an index in this case is not allowed.

The following are examples of this error:

```
INTEGER xArr[10], x2dArr[10][20]; // ok
STRING str[100], strArr[50][100]; // ok
STRING_INPUT strIn[100];          // ok
STRING_OUTPUT strOut;             // ok

STRING str;                     // error – no length specified
STRING_INPUT strIn;             // error – no length specified
BUFFER_INPUT bufIn;             // error – no length specified
STRING_OUTPUT strOutArr[10][20];  // error – 2-D arrays not
supported
STRING str[x];                   // error – variables are not
allowed
STRING str[myFunc()];            // error – function calls are
                                  //         not allowed

INTEGER_FUNCTION MyIntFunc( INTEGER x[], INTEGER xArr[][] )
{
   xArr[1] = 5;                                        // ok
   xArr[1+2] = xArr[3+4];                              // ok
   xArr[1+xArr[2]] = xArr[xArr[3]];                    // ok
   xArr[MyIntFunc(xArr,x2dArr)] = 6;                   // ok

   x2dArr[1][2] = 6;                                   // ok
   x2dArr[xArr[1]][xArr[2]] = x2dArr[xArr[5]][xArr[6]]; // ok

   Call MyFunc( xArr, x2dArr );                        // ok

   xArr = 5;         // error – no index specified
   xArr[] = 0;       // error – no index specified
   xArr[str] = 6;    // error - s is a STRING
   xArr[5][6] = 7;   // error – xArr is not a 2D array
   xArr = xArr;      // error – cannot copy arrays
   xArr = x2dArr[1]; // error – cannot copy arrays
   x2dArr[1] = xArr; // error – cannot copy arrays

   Call MyIntFunc( xArr[5], x2dArr ); // error – cannot pass
index
                                 //        arrays are passed
                                     //         by reference
}

FUNCTION MyStrFunc( STRING s, STRING s[] )   // ok
{
   STRING sLocal[100];

   str = “abc”;                               // ok
   strArr[5] = “def”;                         // ok
   strIn = s;                                 // ok
   strOut = s;                                // ok
   sInArr[5] = “abc”;                         // ok
   sOutArr[5] = “abc”;                        // ok

   Call MyStrFunc( str, strArr );             // ok

   str[1] = “a”;    // error – s is a string, not an array
   sLocal = str[1]; // error – individual characters within
                    //         a string can only be accessed
                    //         with the function, Byte()

}
```

Compiler Error 1302

declaration error: Variable name, ‘<identifier>’, exceeds maximum length of <max> characters

Variable names have a maximum length of 120 characters.

Compiler Error 1303

declaration error: Declaration type not allowed within structure: '<identifier>'
Structure cannot contain String Arrays or Structure variables: '<identifier>'
Structure definitions not allowed within other structures Local Structure declarations are not allowed

Structure datatypes can only be defined globally. Variables of a defined structure datatype may be declared both globally and locally and passed as function arguments. INTEGER, LONG_INTEGER, SIGNED_INTEGER, SIGNED_LONG_INTEGER and STRING are the only SIMPL+ datatypes allowed to be used as structure member fields. INTEGER and LONG_INTEGER can include 1 and 2 dimensional arrays. String arrays are not permitted.

The following are examples of this error:

```
STRUCTURE MyStruct                              // ok
{
   INTEGER i, i1[10], l2[10][20];               // ok
   SIGNED_INTEGER si, si1[10], si2[10][20];     // ok
   LONG_INTEGER l, l1[10], l2[10][20];          // ok
   SIGNED_LONG_INTEGER sl, sl1[10], sl2[10][20]; // ok
   STRING s[100];                               // ok

   STRING sArr[10];    // error – string arrays are not allowed
                       //         within structures

   DIGITAL_INPUT di;   // error – declaration type not allowed
   DIGITAL_OUTPUT do;  // error – declaration type not allowed
   ANALOG_INPUT ai;    // error – declaration type not allowed
   ANALOG_INPUT ao;    // error – declaration type not allowed
   STRING_INPUT si;    // error – declaration type not allowed
   BUFFER_INPUT bi;    // error – declaration type not allowed
   STRING_OUTPUT so;   // error – declaration type not allowed

   STRUCTURE locStruct // error – declaration type not allowed
   {
      INTEGER x;
   }

   MyStruct ptr;       // error – declaration type not allowed
}

FUNCTION MyFunc()
{
   STRUCTURE MyStruct  // error – local structures are not
supported
   {
      INTEGER i, i1[10], l2[10][20];
   }
}
```

Compiler Error 1304

declaration error: Local variables must be declared at top of function

All local variables within a function block must be declared before any statements are encountered. Local variables are not allowed to be declared within a block of statements such as inside an if-else or while loop.

The following are examples of this error:

```
FUNCTION MyFunc( INTEGER arg1, STRING arg2 ) // ok
{
   INTEGER i;                  // ok
   STRING str[100];            // ok

   Print( “Inside MyFunc!” );

   INTEGER j;                  // error

   if ( i > 1 )
   {
     INTEGER k;            // error – if-statement block cannot
                                      contain local variables
   }
}
```

Compiler Error 1305

declaration error: Local functions not supported

A function cannot be defined within another function definition. All function definitions must be defined with a global scope inside the module.

Scope refers to the level at which an Event, user-defined function or statement resides. Having a global scope means that the function or variable can be called or accessed from anywhere within the program. A local scope means that the variable can only be accessed from within the event or function that it resides in.

The following are examples of this error:

```
FUNCTION MyFunc()          // ok – MyFunc is global
{
   FUNCTION MyLocalFunc()  // error – MyLocalFunc is local to
MyFunc
   {
   }
}
```

Compiler Error 1306

declaration error: Declaration type can only be used globally: '<identifier>'

I/O declarations must be defined globally; they cannot be declared as local variables inside of a function or library file.

The following are examples of this error:

```
INTEGER i;              // ok
STRING str[100];        // ok

DIGITAL_INPUT di;       // ok
DIGITAL_OUTPUT do;      // ok
ANALOG_INPUT ai;        // ok
ANALOG_OUTPUT ao;       // ok
STRING_INPUT si[100];   // ok
STRING_OUTPUT so;       // ok
BUFFER_INPUT bi[100];   // ok

FUNCTION MyFunc()
{
   INTEGER i;           // ok – not an I/O declaration
   STRING str[100];     // ok – not an I/O declaration

   DIGITAL_INPUT di;     // error
   DIGITAL_OUTPUT do;    // error
   ANALOG_INPUT ai;      // error
   ANALOG_OUTPUT ao;     // error
   STRING_INPUT si[100]; // error
   STRING_OUTPUT so;     // error
   BUFFER_INPUT bi[100]; // error
}
```

Compiler Error 1307

declaration error: Variables must be declared before array declarations: '<identifier>'

I/O declarations must be declared in a specific order. All arrays of an I/O declaration type (i.e.: DIGITAL_INPUT) must be declared after any variables of the same type.

The following are examples of this error:

```
DIGITAL_INPUT di1, di2;   // ok
DIGITAL_INPUT di3;        // ok

ANALOG_INPUT ai1          // ok

DIGITAL_OUTPUT do1;       // ok

ANALOG_INPUT aiArr1[10];  // ok

DIGITAL_INPUT di4;       // ok – no DIGITAL_INPUT array exists
yet
DIGITAL_INPUT diArr1[10]; // ok

DIGITAL_OUTPUT do2;       // ok

ANALOG_INPUT aiArr2[20];  // ok – multiple arrays are allowed

DIGITAL_INPUT di5;       // error – cannot define after diArr1

ANALOG_INPUT ai2;        // error – cannot define after aiArr2
```

Compiler Error 1308

declaration error: Global variable declaration cannot be declared in library file: '<identifier>'
I/O Declaration cannot be declared in library file: '<identifier>'

I/O declarations and global variables can only be defined in a SIMPL+ module (.usp file). Libraries files (.usl files) are files that only contain functions. Local functions variables, function arguments and function that return values are permitted within library files.

The following are examples of this error:

```
////////////////////////////////////////////////////////////
//////
//  MyLib.usl

INTEGER x;                                    // error – x is global
STRING str[100];                            // error – str is global
DIGITAL_INPUT di;                            // error – di is global

FUNCTION MyFunc()
{
   INTEGER i, j;                        // ok – i and j are local
   STRING str[100];                         // ok – str is local
}

INTEGER_FUNCTION MyIntFunc( INTEGER x ) // ok – x is local
{
   INTEGER i, j;                        // ok – i and j are local
   STRING str[100];                         // ok – str is local

   return (x);
}

STRING_FUNCTION MyStFunc( STRING s )    // ok – s is local
{
   INTEGER i, j;                        // ok – i and j are local
   STRING str[100];                         // ok – str is local

   return (str);
}
```

Compiler Error 1309

declaration error: Compiler Directive must be set before all global variable declarations #DEFAULT_NONVOLATILE Compiler Directive already set #DEFAULT_VOLATILE Compiler Directive already set

The compiler directives, #DEFAULT_VOLATILE and #DEFAULT_NONVOLATILE, must be used before any global variables are encountered within the SIMPL+ module. A module cannot contain more than one of these directives.

The following are examples of this error:

```
////////////////////////////////////////////////////////////
//////
// Example 1

#DEFAULT_VOLATILE     // ok – compiler directive exists before
                      //      all global variables

INTEGER x;
STRING str[100];
DIGITAL_INPUT di;

FUNCTION MyFunc()
{
}

////////////////////////////////////////////////////////////
//////
// Example 2

INTEGER x;
STRING str[100];
DIGITAL_INPUT di;

#DEFAULT_VOLATILE    // error – global variables have already
been
                      //         declared within this module

FUNCTION MyFunc()
{
}

////////////////////////////////////////////////////////////
//////
// Example 3

#DEFAULT_VOLATILE     // ok – compiler directive exists before
```

```
                      //      all global variables
INTEGER x;
STRING str[100];
DIGITAL_INPUT di;

#DEFAULT_NONVOLATILE  // error – #DEFAULT_VOLATILE has already
                      //         been set
INTEGER y;

#DEFAULT_NONVOLATILE  // error – #DEFAULT_VOLATILE has already
                      //         been set
INTEGER z;

FUNCTION MyFunc()
{
}
```

Compiler Error 1310

declaration error: Compiler directive cannot be in function scope

Compiler directives cannot be used locally within functions. They can only be used at a global level and the directive applies to the entire SIMPL+ module.

Scope refers to the level at which an Event, user-defined function or statement resides. Having a global scope means that the function or variable can be called or accessed from anywhere within the program. A local scope means that the variable can only be accessed from within the event or function that it resides in.
The following are examples of this error:

```
#DEFINE_CONSTANT   MyConst   100         // ok – used globally
#USER_LIBRARY “MyUserLib”                // ok – used globally

FUNCTION MyFunc()
{
   #DEFINE_CONSTANT   AnotherConst   100 // error – constants
cannot
                                     //        be used locally

   #USER_LIBRARY “AnotherUserLib”        // error – libraries
cannot
                                  //       be included locally
}
```

Compiler Error 1311

declaration error: Undefined Wait Label: '<identifier>'
Missing, invalid or already defined Wait label: '<identifier>'

Wait Statements can be given a label as an optional argument. This label must be a unique name and more than one wait statement cannot share the same label name. The label name can then be used in the Pause, Cancel and Resume wait functions. All labels must already be declared in within a wait statement before any Pause, Cancel or Resume wait statement can reference it.

The following are examples of this error:

```
FUNCTION MyFunc()
{
   CancelAllWaits();               // ok

   CancelWait( MyWaitLabel );      // error – MyWaitLabel has
                              //        not been declared yet

   Wait( 500 )                   // ok – Label is not required
   {
   }

   Wait( 500, MyWaitLabel )       // ok – MyWaitLabel is unique
   {
   }

   Wait( 500, MyWaitLabel )        // error – MyWaitLabel has
already
                                   //         been used
   {
   }

  CancelWait( AnotherWaitLabel ); // error – AnotherWaitLabel
has
                              //        not been declared yet

  Wait( 500, AnotherWaitLabel )   // ok – AnotherWaitLabel is
unique
   {
   }

   CancelWait( AnotherWaitLabel ); // ok
   PauseWait( MyWaitLabel );       // ok
   ResumeWait( MyFunc );           // error – MyFunc is not a
valid
                                   //         wait label
   ResumeWait( someLabel );        // error – someLabel does
not exist
}
```

Compiler Error 1312

declaration error: Array boundary exceeded maximum size of ‘num_bytes’ bytes

The maximum number of indices for an array is 65535.

The following are examples of this error:

```
FUNCTION MyFunc()
{
   INTEGER int[100], intArr[100][100]; // ok
   STRING str[100], strArr[100][100];  // ok

   INTEGER int[100000];                // error
   INTEGER intArr[100000][100];        // error
   INTEGER intArr[100][100000];        // error

   STRING str[100000];                 // error
   STRING strArr[100000][100];         // error
   STRING strArr[100][100000];         // error
}
```

Compiler Error 1313

declaration error: Minimum array size invalid

The minimum array size cannot exceed the total size of the array. The minimum array size must be between 1 and the total size of the array.

The following are examples of this error:

```
DIGITAL_INPUT digIn1[10];     // ok
DIGITAL_INPUT digIn2[10,5];   // ok – minimum size is 5

ANALOG_INPUT anlgIn3[10,0];   // error – minimum size must be
                              //         greater than 0
STRING_INPUT strIn4[10,20];   // error – minimum size of 20
exceeds
                             //         total array size of 10
```

Compiler Error 1314

declaration error: Minimum array size is not allowed for this datatype: '<identifier>'
Minimum array size for this datatype has already been declared: '<identifier>'

Minimum array sizes are only applicable to Input and Output datatypes (i.e.: DIGITAL_INPUT, ANALOG_OUTPUT, STRING_INPUT, etc.). A variable of another datatype was found trying to define a minimum array size. Only one array for each Input or Output datatype is allowed to be declared with a minimum array size.

The following are examples of this error:

```
DIGITAL_INPUT digIn1[10];    // ok
DIGITAL_INPUT digIn2[10,5];  // ok – minimum size is 5

DIGITAL_INPUT digIn3[20,10]; // error – the DIGITAL_INPUT
array
                       //       variable, digIn2, has already
                       //       been declared with a minimum
                             //         array size

ANALOG_INPUT anlgIn1[10];    // ok
ANALOG_INPUT anlgIn2[10,5];  // ok – no other ANALOG_INPUT has
been
                             //      declared with a minimum
array size

INTEGER x[10];               // ok

INTEGER y[10,5];          // error – INTEGER is not an Input or
                             //         Output datatype
```

Assignment Errors (Compiler Errors 1400 to 1402)

Compiler Error 1400

assignment error: Illegal Assignment

Assignments in SIMPL+ require that the value being assigned to a variable must equate to the same type as that variable. Integer variables can only be assigned integer values and string variables can only be assigned a string value.

- If a complex assignment is being made, make sure that all parenthesis are matched. In other words, all opening parenthesis must have a matching closing parenthesis.
- When comparing 2 strings ('=', '<', '>=', etc.), the resulting value is an integer
- Input variables (DIGITAL_INPUT, ANALOG_INPUT, STRING_INPUT and BUFFER_INPUT) cannot be assigned a value.

The following are examples of this error:

```
INTEGER x, y;
STRING str[100], str2[100];
DIGITAL_INPUT digIn;
DIGITAL_OUTPUT digOut;
ANALOG_OUTPUT anlgOut;

FUNCTION MyFunc()
{
   str = “abc”;                // ok
   str = “abc” + “def”;        // ok
   str = str2;                 // ok

   x = 1;                      // ok
   x = digOut;                 // ok
   x = (str = str2);           // ok
   x = 5 * (1 + (str > str2)); // ok

   digOut = x;                 // ok
   digOut = 5;                 // ok
   digOut = anlgIn;            // ok – both are integer types

   x = str;                  // error – str does not equate to
                               //         an integer
                             //        atoi() should be used

  digIn = 1;              // error - digIn is an input variable

   str = 5;                    // error – 5 is an integer
                         //       MakeString() should be used

   str = str2 = “abc”;         // error = str2 = “abc” is an
equality
                              //        test, not an assignment
}
```

Compiler Error 1401

assignment error: Variable cannot be used for assignment: '<identifier>'

Function arguments that have been declared as ReadOnlyByRef can only have their values read; values cannot be assigned to them.

The following are examples of this error:

```
FUNCTION MyFunc( INTEGER x, ReadOnlyByRef INTEGER y )
{
   x = 5; // ok
   x = y; // ok – the value of y can be read
   y = 6; // error – y is read-only
}
```

Compiler Error 1402

assignment error: Variable can only be used for assignment: '<identifier>'

STRING_OUTPUT variables can only have their values read. Once assigned a value, that value is immediately acted upon by the control system, and the value is assumed to be unknown thereafter.

The following are examples of this error:

```
STRING_OUTPUT sOut;
STRING str[100];

FUNCTION MyFunc()
{
   str = “abc”;                 // ok
  sOut = str;              // ok – sOut can be assigned a value
  sOut = “abc”;            // ok – sOut can be assigned a value

  str = sOut;              // error – the value of sOut is lost

   Print( “str = %s”, str );   // ok – STRINGs can be read and
written
   Print( “sOut = %s”, sOut ); // error – the value of sOut is
unknown
}
```

Function Argument Errors (Compiler Errors 1500 to 1508)

Compiler Error 1500

function argument error: Argument <arg_num> cannot be passed by reference

A variable was being passed that can either only have a value assigned to it, or it's value be copied into another variable or used within an expression. An example of this is trying to pass a STRING_INPUT variable as a function argument; the STRING_INPUT must first be copied into a STRING variable and then passed.

Pass by Reference – The function will act directly on the variable that was passed as the argument. Any changes to the variable within the called function the will be reflected within the calling function.

Pass by Value – The function creates a local copy of the source variable. Any changes to this local copy are not reflected in the source variable that was originally passed to the function. The source variable will still retains its original value from before the function was called..

The following are examples of this error:

```
INTEGER i;
STRING str[100];
STRING_INPUT strIn[100];
STRING_OUTPUT strOut;
DIGITAL_INPUT di;

FUNCTION MyFunc( STRING s )
{
   str = strIn;
  Call MyFunc( str );     // ok – the previous statement copied
                             //      'strIn' into 'str'

   Call MyFunc( "abc" );    // ok

   Call MyFunc( strIn );    // error – strIn is a STRING_INPUT
and
                       //       cannot be passed by reference

  Call MyFunc( strOut );   // error – strIn is a STRING_OUTPUT
and
                       //       cannot be passed by reference
}

FUNCTION MyFunc2( ByRef STRING s ) // error – STRINGs cannot be
                                  //        passed by reference
{
   Call MyFunc2( str );           // error – STRINGs cannot be
                                  //        passed by reference
}

FUNCTION AnotherFunc( ByRef INTEGER x )
{
   Call AnotherFunc( 1 );   // ok

  Call AnotherFunc( di );  // error – di is a DIGITAL_INPUT and
                             //         cannot be passed
                             //         by reference

}
```

Compiler Error 1501

function argument error: Argument <arg_num> cannot be passed by value

In SIMPL+, arrays can only be passed by reference. The keyword, ByVal, cannot be used within a function's argument list in conjunction with arrays. A copy of an individual element within an array must first be copied into an INTEGER or STRING variable and then that variable can be passed.

Pass by Reference – The function will act directly on the variable that was passed as the argument. Any changes to the variable within the called function the will be reflected within the calling function.

Pass by Value – The function creates a local copy of the source variable. Any changes to this local copy are not reflected in the source variable that was originally passed to the function. The source variable will still retains it's original value from before the function was called.

The following are examples of this error:

```
FUNCTION MyFunc( ByVal INTEGER intArr[] )   // error – arrays
cannot be
                                     //       passed by value
{
}

FUNCTION MyFunc( ByVal STRING strArr[] )    // error – arrays
cannot be
                                     //       passed by value
{
}

FUNCTION MyFunc( ByVal INTEGER intArr[][] ) // error – arrays
cannot be
                                     //       passed by value
{
}

FUNCTION MyFunc( ByVal STRING strArr[][] )  // error – arrays
cannot be
                                     //       passed by value
{
}
```

Compiler Error 1502

function argument error: Function contains incomplete number of arguments
Function call contains an unmatched number of arguments

When calling a functions that contain parameter lists, the number of arguments passed to the function must match the number of parameters defined for that function.

The following are examples of this error:

```
FUNCTION MyFunc( INTEGER x, STRING str )
{
   Call MyFunc( 1, “abc” ); // ok

   Call MyFunc();          // error – 2 arguments are expected
   Call MyFunc( 1 );        // error – argument 2 is missing
}
```

Compiler Error 1503

function argument error: Input or Output signal expected: '<identifier>'

The expected identifier must be of an Input or Output signal datatype (i.e.: DIGITAL_INPUT, ANALOG_OUTPUT, STRING_INPUT, etc.).

The following are examples of this error:

```
DIGITAL_INPUT digIn, digInArr[10];
DIGITAL_INPUT digIn;
ANALOG_INPUT anlgIn;
ANALOG_OUTPUT anlgOut;
STRING_INPUT strIn[100];
STRING_OUTPUT strOut;
BUFFER_INPUT buffIn[100];

INTEGER i;
STRING str[100];

FUNCTION MyFunc()
{
   i = IsSignalDefined( digIn );       // ok
   i = IsSignalDefined( digInArr[5] ); // ok
   i = IsSignalDefined( digOut );      // ok
   i = IsSignalDefined( anlgIn );      // ok
   i = IsSignalDefined( anlgOut );     // ok
   i = IsSignalDefined( strIn );       // ok
   i = IsSignalDefined( strOut );      // ok
   i = IsSignalDefined( buffIn );      // ok

   digOut = IsSignalDefined( i );      // error – ‘i’ is not
an Input
```

```
                                   //        or Output signal
   i = IsSignalDefined( str );         // error – ‘i’ is not
an Input
                                   //        or Output signal
   digOut = IsSignalDefined( 5 );      // error – ‘5’ is not
an Input
                                   //        or Output signal
}
```

Compiler Error 1504

function argument error: Incomplete number of format string arguments
Format string contains an unmatched number of arguments
Argument <arg_num> is missing or invalid.
Format Specifier expected
Argument <arg_num> is missing or invalid. <decl_type> expected

Format lists contain format specifiers that tell the compiler to replace a given specifier with the value or result given in the argument list that follows. A format list is analogous to a function parameter list in that the format specifier tells the compiler what type of argument to expect. For each format specifier, their must be a corresponding value or result in the argument list that follows. This value or result must also be of the same datatype.

Format strings contain specifications that determine the output format for the arguments. The format argument consists of ordinary characters, escape sequences, and (if arguments follow format) format specifications Format Specifications always begin with a percent sign (%) and are read left to right. When the first format specification is encountered (if any), it converts the value of the first argument after format and outputs it accordingly. The second format specification causes the second argument to be converted and output, and so on.
The following are examples of this error:

```
FUNCTION MyFunc()
{
   INTEGER x, intArr[100];
   STRING str[100], strArr[100][100];

   Print( “Hello World” );                                 // ok
   Print( “My name is %s.  My age is %d”, “David”, 33 );      /
/ ok
  Print( “My name is %s.  My age is %d”, str, x );         // ok

  MakeString( str, “Hello World” );                        // ok
   MakeString( str, “My name is %s.  My age is %d”, str, x );
// ok

   Print( “My name is %s.  My age is %d”, “David” );     //
error –
                       // %d format specifier does not have a
```

```
                        // corresponding value

   Print( “My name is %s.  My age is %d”, 33, “David” ); //
error -
                        // both format specifiers contain
corresponding
                        // values of different datatypes

   SetArray( strArr, 1 );     // ok
   SetArray( strArr, “abc” ); // ok
   SetArray( intArr, 0 );     // ok

   SetArray( “abc”, 1 );      // error – “abc” is not an array
   SetArray( 1, “abc” );      // error – 1 is not an array
}
```

Compiler Error 1505

function argument error: Format string contains invalid format specifier

An invalid format specifier was used within a format string.

Format strings contain specifications that determine the output format for the arguments. The format argument consists of ordinary characters, escape sequences, and (if arguments follow format) format specifications Format Specifications always begin with a percent sign (%) and are read left to right. When the first format specification is encountered (if any), it converts the value of the first argument after format and outputs it accordingly. The second format specification causes the second argument to be converted and output, and so on.
The following are examples of this error:

```
FUNCTION MyFunc()
{
  Print( “Hello World” );                               // ok
  Print( “My name is %s.  My age is %d”, “David”, 33 );     /
/ ok

   Print( “My name is %xs”, “David” ); // error - %xs is an
invalid
                                  //        format specifier
}
```

Compiler Error 1506

function argument error: 0, 1 or 2 constant expected for argument 1

The function, MakeString, can contain a 0, 1, 2 as the first argument. This tells the control system to output the resulting string to a specific destination. An integer value other than 0, 1 or 2 was encountered as the first argument of MakeString().

The different destinations are as follows:
0: Computer Port, same as PRINT.
1: CPU (same functionality as the SendPacketToCPU function)
2: Cresnet Network (same functionality as the SendCresnetPacket function).

The following are examples of this error:

```
FUNCTION MyFunc( INTEGER x, STRING str )
{
   Call MyFunc( 1, “abc” ); // ok

   Call MyFunc();          // error – 2 arguments are expected
   Call MyFunc( 1 );        // error – argument 2 is missing
}
```

Compiler Error 1507

function argument error: Argument <arg_num>: Missing or invalid array

An integer or string variable array was expected and was not encountered.

The following are examples of this error:

```
FUNCTION MyFunc( INTEGER x[], STRING str[] )
{
   INTEGER i;
   STRING strArr[100][100];

   SetArray( x, 1 );         // ok
   Call MyFunc( x, StrArr ); // ok

   SetArray( i, 1 );         // error – i is not an array
   Call MyFunc( 1, “abc” );  // error – 1 is not an array
}
```

Compiler Error 1508

function argument error: I/O variable cannot be passed to read file functions: '<identifier>'

Read file functions (ReadInteger, ReadString, etc.) cannot contain Input or Output variables for the function's resulting read buffer.

The following are examples of this error:

```
DIGITAL_OUTPUT digOut;
STRING_OUTPUT strOut;

FUNCTION MyFunc( SIGNED_INTEGER nHandle )
{
   STRING str[100];
   INTEGER x;

   ReadInteger( nHandle, x );      // ok
   ReadString( nHandle, str );     // ok

   ReadInteger( nHandle, digOut ); // error
   ReadString( nHandle, strOut );  // error
}
```

Construct Errors (Compiler Errors 1600 to 1608)

Compiler Error 1600

construct error: 'Function Main' cannot contain function parameters 'Function Main' cannot return a value

Function Main is the starting point of a SIMPL+ program. It is automatically called once when the system startup or is reset. Since this function is invoked by a method outside of the SIMPL+ module, no arguments can be included in it's argument list and no value can be returned from it.

The following are examples of this error:

```
Function Main()                     // ok
{
}

INTEGER_FUNCTION Main()             // error – Main() cannot
return
                                    //         a value
{
}

Function Main( INTEGER cmdLineArg ) // error – Main() cannot
contain
                                  //        a parameter list
{
}
```

Compiler Error 1601

construct error: DuplicateCASE Statement Constant expected: '<identifier>'

Unlike the Switch Statement the CSwitch statement's case statements must consist of unique values. Expressions are not permitted within the case statements. Instead, each case statement must contain a unique integer value for the CSwitch's comparison.

The following are examples of this error:

```
FUNCTION MyFunc( INTEGER x )
{
   STRING str[100];

   CSwitch( x )
   {
      case (1):     // ok – 1 has not been used yet
      {
      }

      case (2):     // ok – 2 has not been used yet
```

```
            {
            }

            case (2):     // error – 2 has been previously used
            {
            }

            case (5+6):   // error – expressions are not allowed
            {
            }

            case (x):     // error – variables are not allowed
            {
            }

            case (“abc”): // error – strings are not allowed
            {
            }

            case (str):   // error – strings are not allowed
            {
            }
         }
      }
```

Compiler Error 1602

construct error: Switch statement contains 'default' without 'case' labels

The Switch and CSwitch constructs must contain ‘case’ statements if the ‘default’ statement is to be used. The ‘default’ statement is optional.

The following are examples of this error:

```
FUNCTION MyFunc( INTEGER x )
{
   Switch ( x )
   {
      case (1):  // ok
      {
      }

      default:   // ok
      {
      }
   }

   CSwitch ( x )
   {
      case (1):  // ok
      {
      }

      default:   // ok
```

```
         {
         }
      }

      Switch ( x )
      {
         default:   // error – no Case statement in Switch
         {
         }
      }

      CSwitch ( x )
      {
         default:   // error – no Case statement in Switch
         {
         }
      }
   }
```

Compiler Error 1603

construct error: #CATEGORY does not exist: '<categorgy_number>'. Defaulting to Category Type, ""32"" (Miscellaneous).

The category number for this compiler directive was not found or was not a valid category number within the Symbol Tree Category List within SIMPL windows. The category number must be enclosed in quotation marks.

Selecting Edit | Insert Category from the SIMPL+ menu will display the list of valid category numbers and give the option for this compiler directive to be automatically inserted into the SIMPL+ module.

The following are examples of this error:

```
#CATEGORY “6”                // ok – “6” is the category number
                               //       for Lighting

#CATEGORY “Lighting”          // error – the category number,
“6”,
                           //        should be used instead of
                            //        the category symbol name

#CATEGORY 6                // error – the category number should
                        //       be enclosed in quotation marks

#CATEGORY 99                // error – invalid category number

#DEFINE_CONSTANT MyCategory 6
#CATEGORY MyCategory          // error – cannot substitute
category
                        //       number with #DEFINE_CONSTANTs
```

Compiler Error 1604

construct error: 'EVENT' already has a body

The EVENT statement can only be defined once per SIMPL+ module. A previously defined definition of EVENT was already encountered by the compiler.

The following are examples of this error:

```
EVENT  // ok
{

}

EVENT  // error – EVENT is already defined
{

}
```

Compiler Error 1605

construct error: Function can only be contained within an event

The function, TerminateEvent, can only be used within a PUSH, CHANGE, RELEASE or EVENT statement. The compiler encountered this function outside of one of these event functions.

The following are examples of this error:

```
DIGITAL_INPUT digIn;

EVENT
{
   TerminateEvent;    // ok
}

PUSH digIn
{
   TerminateEvent;    // ok
}

RELEASE digIn
{
   TerminateEvent;    // ok
}

CHANGE digIn
{
   TerminateEvent;    // ok
}

FUNCTION MyFunc()
{
   while (1)
```

```
      {
         TerminateEvent; // error – TerminateEvent is not within
                         //         an event function
      }
   }
```

Compiler Error 1606

construct error: Statement must be contained within a loop statement

The ‘break’ statement can only be used with a loop construct. Valid loop constructs are While loops, Do-While loops and For loops. The compiler encountered this function outside of one of these event functions.

The following are examples of this error:

```
FUNCTION MyFunc()
{
   INTEGER I;

   for ( i = 1 t 10 )
   {
      break;    // ok
   }

   while (1)
   {
      break;    // ok
   }

   do
   {
      break;    // ok

   } until (1);

   if (1)
   {
      break;    // error – break cannot exist within an ‘if’
statement
   }
}

EVENT
{
  break;      // error – TerminateEvent should be used instead
}
```

Compiler Error 1607

construct error: GetLastModifiedArrayIndex may return an ambiguous signal index

If an event function (EVENT, PUSH, CHANGE, RELEASE) is acting on more than one input array signal, the specific array will not be able to be determined based on the index returned from GetLastModifiedArrayIndex(). In order to use GetLastModifiedArrayIndex() for multiple input signal arrays, a separate event function will have to be defined for each array.

The following are examples of this error:

```
DIGITAL_INPUT digIn[10];
ANALOG_INPUT anlgIn[10];

PUSH digIn
{
   INTEGER i;

   i = GetLastModfiedArrayIndex();  // ok – index from digIn
}

PUSH anlgIn
{
   INTEGER i;

   i = GetLastModfiedArrayIndex();  // ok – index from anlgIn
}

CHANGE digIn, anlgIn
{
   INTEGER i;

  i =  GetLastModfiedArrayIndex(); // error – ambiguous result
}
```

Compiler Error 1608

construct error: Missing library file name

A filename was not found following the compiler directive, #USER_LIBRARY or #CRESTRON_LIBRARY. This filename must be enclosed within quotation marks. The file extension (.usl or .csl) should NOT be used when specifying the filename.

The following are examples of this error:

```
#USER_LIBRARY "MyUserLib"     // ok
#CRESTRON_LIBRARY "EvntSched" // ok

#USER_LIBRARY MyUserLib      // error – missing quotation marks
#USER_LIBRARY MyUserLib.usl   // error – missing quotation
marks and
                              //        extension is not allowed
```

File Errors (Compiler Errors 1700 to 1702)

Compiler Error 1700

file error: End of file reached

The compiler reached the end of file before all functions or statements were completed.

- Make sure all functions have matching opening and closing braces.
- Make sure all statement expressions have matching opening and closing parenthesis.

Compiler Error 1701

file error: Error writing header file: '<file_name>'
Error writing file: '<file_name>'
Error writing library file
Error writing output file
Error creating compiler makefile: '<file_name>'
Error opening compiler source makefile: '<file_name>'
Error opening source file: '<file_name>'

The specified file could not be opened or created.

- Make sure there is sufficient disk space for the file to be written.
- If including a User or Crestron Library (#USER_LIBRARY or #CRESTRON_LIBRARY), make sure the library file name is valid, spelled properly and does not contain the file extension (.usl or .csl).
- Make sure the latest version of the Crestron Database is installed.
- Make sure the path for the Crestron Database and User SIMPL+ files have been specified within SIMPL Windows.
- Make sure the file does not exist with a Read-Only attribute.
- Make sure another application (i.e.: another instance of SIMPL+) is not currently running with this file open.

Compiler Error 1702

file error: Error extracting library, '<file_name>', from archive: '<archive_file>'

The specified file was not found within the Crestron Library archive.

- Make sure the library file name is valid, spelled properly and does not contain the file extension (.csl).
- Make sure the latest version of the Crestron Database is installed.
- Make sure the path for the Crestron Database has been specified within SIMPL Windows.

Compiler Warnings (Compiler Errors 1800 to 1803)

Compiler Warning 1800

compiler warning: 'Return' statement will only terminate current Wait statement's function scope

A ‘Return’ statement within a Wait Statement’s block of code will cause the Wait Statement to terminate. It will NOT terminate the current function that the Wait Statement resides within.

Wait Statements are similar to event functions (EVENT, PUSH, CHANGE, RELEASE) in that they execute in their own program thread. The control system can have many threads executing at the same time; each thread runs concurrent with one another.

The following are examples of this warning:

```
FUNCTION MyFunc( INTEGER x )
{
   if ( x == 1 )
   {
      Wait( 500 )
      {
         return;      // warning - this will terminate the
                     //           Wait Statement. It will NOT
                      //           terminate MyFunc()
      }
   }
   else if ( x == 2 )
      return;         // this will terminate MyFunc()

   x = x + 1;
}
```

Compiler Warning 1801

compiler warning: 'TerminateEvent' statement will only terminate current Wait statement's function scope

When Wait Statements are embedded within one another, the TerminateEvent, will only terminate the corresponding Wait Statement of the same scope. It will NOT terminate any Wait Statements that are of a different scope.

Wait Statements are similar to event functions (EVENT, PUSH, CHANGE, RELEASE) in that they execute in their own program thread. The control system can have many threads executing at the same time; each thread runs concurrent with one another.

Scope refers to the level at which an Event, user-defined function or statement resides. Having a global scope means that the function or variable can be called or accessed from anywhere within the program. A local scope means that the variable can only be accessed from within the event or function that it resides in.

The following are examples of this warning:

```
FUNCTION MyFunc( INTEGER x )
{
   Wait( 500, MyLabel1 )
   {
      Wait( 300, MyLabel2 )
      {
       TerminateEvent;    // warning – this will only terminate
                        //         the Wait Statent, MyLabel2.
                         //          MyLabel1 will continue to
                             //           execute
      }
   }
}
```

Compiler Warning 1802

compiler warning: #CATEGORY_NAME defined more than once. Using: #CATEGORY_NAME "<number>"

Only one category name is allowed for each SIMPL+ module. If the compiler directive, #CATEGORY, is found more than once within a SIMPL+ module, the compiler will use the category number from the last occurrence of the compiler directive.

The following are examples of this warning:

```
#CATEGORY “1”
#CATEGORY “2”

FUNCTION MyFunc()
{
}

#CATEGORY “3”         // this is the resulting category number
                        // for this SIMPL+ module

FUNCTION AnotherFunc()
{
}
```

Compiler Warning 1803

compiler warning: Possible data loss: LONG_INTEGER to INTEGER assignment

A LONG_INTEGER result was assigned to an INTEGER variable or passed to a function for an INTEGER parameter. The 32-bit LONG_INTEGER will be truncated to 16-bit value and assigned to the integer, resulting in a loss of data.

- Make sure all the datatypes within an expression are of the same datatype.
- Make sure the parameter of a function being called is of the same datatype as the argument being passed in.
- Make sure the return value of a function matches the destination's datatype.

The following are examples of this warning:

```
LONG_FUNCTION MyFunc( INTEGER x )
{
   INTEGER i;
   LONG_INTEGER j;

   i = i;           // ok – both sides of the assigment are of
                     //      the same datatype
   j = i;            // ok – no loss of data
   j = j;           // ok – both sides of the assigment are of
                     //      the same datatype

   i = j;          // warning – LONG_INTEGER being assigned to
                     //           an INTEGER

   Call MyFunc( i ); // ok
   Call MyFunc( j ); // warning

   i = MyFunc( 5 );  // warning – the integer, i, is being
assigned a
                     //           LONG_INTEGER value
}
```

SIMPL+ Revisions

For the latest revisions to SIMPL Windows, refer to the release notes installed with the program. This can be accessed in the Start Menu, under **Programs | Crestron | SIMPL Windows**

Obsolete Functions

System Interfacing - Cresnet and CPU

GetCIP

Name:

GetCIP

Syntax:

```
INTEGER GetCIP(INTEGER CIPID, INTEGER JOIN_NUMBER,
INTEGER TYPE);
```

Description:

Retrieves the current state of the join number on a particular CIP ID (referred to as IP ID in SIMPL+). Note that the device must be defined in SIMPL Windows and the join number to use must have a signal tied to it for this function to work.

NOTE: CIP is defined as Cresnet (over) Internet Protocol.

Parameters:

CIPID is an INTEGER containing the ID of the CIP device to query.

JOIN_NUMBER is an INTEGER containing the Join number to get the status. For touchpanels, the join number is identical to the press/feedback number. For other devices, contact Crestron customer service.

TYPE is one of several predefined constants:

din:	Digital inputs from device (symbol output list)
ain:	Analog inputs from device (symbol output list)
dout:	Digital outputs to device (symbol input list)
aout:	Analog outputs to device (symbol input list)

NOTE: Access to serial signals is not supported.

Return Value:

An Integer. For Digital values, a non-zero value indicates a logic high and a 0 value indicates a logic low. For analog values, a 16-bit number is returned corresponding to the state of the analog join.

Example:

Assuming a relay card has been defined in Slot 1 and Relay A2 has a signal name tied to it, and a CEN-IO has been defined at CIP ID 03 and cue i1 has a signal tied to it, this SIMPL+ statement will connect the two:

```
SetSlot(1,2,dout) = GetCIP(0x03,18,din);
```

NOTE: In the above example statement, the join number representing cue i1 on the CEN-IO is 18.

NOTE: This is not a permanent connection; it will only set the state when this statement is executed

Version:

SIMPL+ Version 2.00 only. This function is not available in Versions 1.00 or 3.00.

Control System:

X-Generation only

GetCresnet

Name:

GetCresnet

Syntax:

```
INTEGER GetCresnet(INTEGER CRESNET_ID, INTEGER
JOIN_NUMBER, INTEGER TYPE);
```

Description:

Retrieves the current state of the join number on a particular Cresnet Network ID. Note that the device must be defined in SIMPL Windows and the join number to use must have a signal tied to it for this function to work.

Parameters:

CRESNET_ID is an INTEGER containing the ID of the Cresnet Network device to query.

JOIN_NUMBER is an INTEGER containing the Join number to get the status. For touchpanels, the join number is identical to the press/feedback number. For other devices, contact Crestron customer service.

TYPE is one of several predefined constants:

din:	Digital inputs from device (symbol output list)
ain:	Analog inputs from device (symbol output list)
dout:	Digital outputs to device (symbol input list)
aout:	Analog outputs to device (symbol input list)

NOTE: Access to serial signals is not supported.

Return Value:

An Integer. For Digital values, a non-zero value indicates a logic high and a 0 value indicates a logic low. For analog values, a 16-bit number is returned corresponding to the state of the analog join.

Example:

Assuming a relay card has been defined in Slot 1 and Relay A2 has a signal name tied to it, and a touchpanel has been defined at Cresnet ID 07, and press 42 has a signal name tied to it, this SIMPL+ statement will connect the two:

```
SetSlot(1,2,dout) = GetCresnet(0x07,42,din);
```

NOTE: This is not a permanent connection; it will only set the state when this statement is executed.

Version:

SIMPL+ Version 2.00 only. This function is not available in Versions 1.00 or 3.00.

Control System:

X-Generation only

GetSlot

Name:

GetSlot

Syntax:

```
INTEGER GetSlot(INTEGER SLOT_NUMBER, INTEGER
JOIN_NUMBER, INTEGER TYPE);
```

Description:

Retrieves the current state of the join number on a particular card. Note that the device must be defined in SIMPL Windows and the join number to use must have a signal tied to it for this function to work.

Parameters:

SLOT_NUMBER is an INTEGER containing the slot number of the card to query.

JOIN_NUMBER is an INTEGER containing the Join number to get the status.

TYPE is one of several predefined constants:

din:	Digital inputs from device (symbol output list)
ain:	Analog inputs from device (symbol output list)
dout:	Digital outputs to device (symbol input list)
aout:	Analog outputs to device (symbol input list)

NOTE: Access to serial signals is not supported.

Return Value:

An Integer. For Digital values, a non-zero value indicates a logic high and a 0 value indicates a logic low. For analog values, a 16-bit number is returned corresponding to the state of the analog join.

Example:

Assuming a relay card has been defined in Slot 1 and Relay A2 has a signal name tied to it, and a CNXIO-16 has been defined in Slot 2 and cue i1 has a signal tied to it, this SIMPL+ statement will connect the two:

```
SetSlot(1,2,dout) = GetSlot(2,1,din);
```

NOTE: This is not a permanent connection; it will only set the state when this statement is executed.

Version:

SIMPL+ Version 2.00 only. This function is not available in Versions 1.00 or 3.00.

Control System:

X-Generation only

IsSignalDefined

Name:

IsSignalDefined

Syntax:

```
INTEGER IsSignalDefined <input/output signal>;
```

Description:

Retrieves the current SIMPL signal number associated with a particular input or output. This is generally used to determine if a particular input or output on a gate is being used, and generally used with arrayed inputs or outputs. This can be used to build a gate of a predetermined maximum size, and allow the user to add and subtract signals on the input or output of the gate (i.e., the program would be written to iterate through a DIGITAL_INPUT array until IsSignalDefined returns a 0).

Parameters:

Legal output and input signal types are ANALOG_INPUT, ANALOG_OUTPUT, BUFFER_INPUT, DIGITAL_INPUT, DIGITAL_OUTPUT, STRING_INPUT, STRING_OUTPUT.

Return Value:

The particular input or output is tied to an integer giving the signal number. If a signal has been tied to that input or output of the gate, a non-zero value will be returned. If the signal is tied to 0 on the SIMPL gate or the signal is not defined, then 0 will be returned. If the signal is tied to 1, then 1 is returned.

Example:

```
DIGITAL_INPUT INS[20];
INTEGER NumInputs;

FUNCTION MAIN()
{
FOR(NumInputs = 20 to 1 Step -1)
IF(IsSignalDefined(INS[NumInputs]))
Break;
}
```

This example computes how many inputs are used on the gate. It should be noted that it is useful to work backwards from the end of the gate. If the user tied five signals, a 0, and then five more signals, this would yield the correct result that the 11th input was the last one used.

Version:

SIMPL+ Version 2.00

SendCresnetPacket

Name:

SendCresnetPacket

Syntax:

```
SendCresnetPacket(STRING PACKET);
```

Description:

Sends the string specified by PACKET onto the Cresnet network. It duplicates the function of the SIMPL Windows symbol “Network Transmission (Speedkey: NTX).” This function is not used in general programming.

Parameters:

PACKET is a string containing the command to put on the Cresnet network.

Return Value:

None.

Example:

```
SendCresnetPacket(“\xFF\x03\x02”);
```

This example will send a broadcast message to all touchpanels causing them to enter sleep mode. The preferable way to do this is use the SLEEP input of the BROADCAST symbol in SIMPL Windows.

Version:

SIMPL+ Version 2.00

SendPacketToCPU

Name:

SendPacketToCPU

Syntax:

```
SendPacketToCPU(STRING PACKET);
```

Description:

Sends the string specified by PACKET to the Cresnet CPU. This is normally used for sending ESC style commands to the CPU for control. This function duplicates the functionality of the SIMPL Windows symbol “Send Message to CPU (Speedkey: TMSG).” This function is not used in general programming.

Parameters:

PACKET is a string containing the command to send to the CPU.

Return Value:

None.

Example:

```
SendPacketToCPU("\x1BDFF\r");
```

This example will turn the Super Debugger on, which shows all network transitions on the console port of the control system. This command would normally be typed in manually through the Crestron Viewport, since it is for debugging only.

Version:

SIMPL+ Version 2.00

SetCIP

Name:

SetCIP

Syntax:

```
SetCIP(INTEGER CIPID, INTEGER JOIN_NUMBER, INTEGER
TYPE);
```

Description:

Sets the state of the join number on a particular CIP ID. Note that the device must be defined in SIMPL Windows and the join number to use must have a signal tied to it for this function to work.

Parameters:

CIPID is an INTEGER containing the ID of the CIP device to set the join number. JOIN_NUMBER is an INTEGER containing the Join number to set. TYPE is one of several predefined constants:

din:	Digital inputs from device (symbol output list)
ain:	Analog inputs from device (symbol output list)
dout:	Digital outputs to device (symbol input list)
aout:	Analog outputs to device (symbol input list)

Return Value:

None.

Example:

Assuming a CEN-IO has been defined at CIP ID 03 and Relay1 has a signal name tied to it, and a touchpanel has been defined at Cresnet ID 07, and press 42 has a signal name tied to it, this SIMPL+ statement will connect the two:

```
SetCIP(0x03,1,dout) = GetCresnet(0x07,42,din);
```

NOTE: This is not a permanent connection; it will only set the state when this statement is executed.

Version:

SIMPL+ Version 2.00 only. This function is not available in Versions 1.00 or 3.00.

Control System:

X-Generation only

SetCresnet

Name:

SetCresnet

Syntax:

```
SetCresnet(INTEGER CRESNET_ID, INTEGER JOIN_NUMBER,
INTEGER TYPE);
```

Description:

Sets the state of the join number on a particular Cresnet Network ID. Note that the device must be defined in SIMPL Windows and the join number to use must have a signal tied to it for this function to work.

Parameters:

CRESNET_ID is an INTEGER containing the ID of the Cresnet Network device to set the join number. JOIN_NUMBER is an INTEGER containing the Join number to set. TYPE is one of several predefined constants:

din:	Digital inputs from device (symbol output list)
ain:	Analog inputs from device (symbol output list)
dout:	Digital outputs to device (symbol input list)
aout:	Analog outputs to device (symbol input list)

Return Value:

None.

Example:

Assuming a touchpanel has been defined at Cresnet ID 07, and press 42 and feedback 69 have signal names tied to them, this SIMPL+ statement will connect the two:

```
SetCresnet(0x07,69,dout) = GetCresnet(0x07,42,din);
```

NOTE: This is not a permanent connection; it will only set the state when this statement is executed.

Version:

SIMPL+ Version 2.00 only. This function is not available in Versions 1.00 or 3.00.

Control System:

X-Generation only

SetSlot

Name:

SetSlot

Syntax:

```
SetSlot(INTEGER SLOT_NUMBER, INTEGER JOIN_NUMBER,
INTEGER TYPE);
```

Description:

Sets the state of the join number on a particular card slot. Note that the device must be defined in SIMPL Windows and the join number to use must have a signal tied to it for this function to work.

Parameters:

SLOT_NUMBER is an INTEGER containing the slot number of card to set the join number. JOIN_NUMBER is an INTEGER containing the Join number to set. TYPE is one of several predefined constants:

din:	Digital inputs from device (symbol output list)
ain:	Analog inputs from device (symbol output list)
dout:	Digital outputs to device (symbol input list)
aout:	Analog outputs to device (symbol input list)

Return Value:

None.

Example:

Assuming a relay card has been defined in Slot 1 and Relay A2 has a signal name tied to it, and a touchpanel has been defined at Cresnet ID 07, and press 42 has a signal name tied to it, this SIMPL+ statement will connect the two:

```
SetSlot(1,2,dout) = GetCresnet(0x07,42,din);
```

NOTE: This is not a permanent connection; it will only set the state when this statement is executed.

Version:

SIMPL+ Version 2.00 only. This function is not available in Versions 1.00 or 3.00.

Control System:

X-Generation only

Interfacing to the CEN-OEM

Interfacing to the CEN-OEM via a SIMPL+ Program Overview

When using a X-Generation system to communicate over Ethernet to a CEN-OEM, the CEN-OEM definition is used from the SIMPL Windows Configuration Manager. This symbol has analog inputs, analog outputs, digital inputs, digital outputs, serial inputs, and serial outputs.

When a list of variables such as DIGITAL_INPUTs is declared, they normally start at Digital Input 1 on the symbol and progress linearly up. For some applications, it may be desirable to change the join numbers (leave gaps on the symbol) for a better visual look.

#ANALOG_INPUT_JOIN

Name:

#ANALOG_INPUT_JOIN

Syntax:

```
#ANALOG_INPUT_JOIN<constant>
```

Description:

Changes the join number starting with the next ANALOG_INPUT definition to the join number specified by <constant>.

Example:

```
ANALOG_INPUT SIG1, SIG2, SIG3, SIG4;
```

In this example, SIG1 references Join #1, SIG2 references Join #2, SIG3 references Join #3, and SIG4 references Join #4.

```
ANALOG_INPUT SIG1, SIG2;
#ANALOG_INPUT_JOIN 20
ANALOG_INPUT SIG3, SIG4;
```

Here, SIG1 and SIG2 still reference Join #1 and Join #2, but SIG3 has been changed to reference Join #20, and SIG4 references Join #21.

Version:

SIMPL+ Version 2.00

#ANALOG_OUTPUT_JOIN

Name:

#ANALOG_OUTPUT_JOIN

Syntax:

```
#ANALOG_OUTPUT_JOIN<constant>
```

Description:

Changes the join number starting with the next ANALOG_OUTPUT definition to the join number specified by <constant>.

Example:

```
ANALOG_OUTPUT SIG1, SIG2, SIG3, SIG4;
```

In this example, SIG1 references Join #1, SIG2 references Join #2, SIG3 references Join #3, and SIG4 references Join #4.

```
ANALOG_OUTPUT SIG1, SIG2;
#ANALOG_OUTPUT_JOIN 20
ANALOG_OUTPUT SIG3, SIG4;
```

SIG1 and SIG2 still reference Join #1 and Join #2, but SIG3 has been changed to reference Join #20, and SIG4 references Join #21.

Version:

SIMPL+ Version 2.00

#DIGITAL_INPUT_JOIN

Name:

#DIGITAL_INPUT_JOIN

Syntax:

```
#DIGITAL_INPUT_JOIN<constant>
```

Description:

Changes the join number starting with the next DIGITAL_INPUT definition to the join number specified by <constant>.

Example:

```
DIGITAL_INPUT SIG1, SIG2, SIG3, SIG4;
```

In this example, SIG1 references Join #1, SIG2 references Join #2, SIG3 references Join #3, and SIG4 references Join #4.

```
DIGITAL_INPUT SIG1, SIG2;
```

```
#DIGITAL_INPUT_JOIN 20
DIGITAL_INPUT SIG3, SIG4;
```

SIG1 and SIG2 still reference Join #1 and Join #2, but SIG3 has been changed to reference Join #20, and SIG4 references Join #21.

Version:

SIMPL+ Version 2.00

#DIGITAL_OUTPUT_JOIN

Name:

#DIGITAL_OUTPUT_JOIN

Syntax:

```
#DIGITAL_OUTPUT_JOIN<constant>
```

Description:

Changes the join number starting with the next DIGITAL_OUTPUT definition to the join number specified by <constant>.

Example:

```
DIGITAL_OUTPUT SIG1, SIG2, SIG3, SIG4;
```

In this example, SIG1 references Join #1, SIG2 references Join #2, SIG3 references Join #3, and SIG4 references Join #4.

```
DIGITAL_OUTPUT SIG1, SIG2;
#DIGITAL_OUTPUT_JOIN 20
DIGITAL_OUTPUT SIG3, SIG4;
```

SIG1 and SIG2 still reference Join #1 and Join #2, but SIG3 has been changed to reference Join #20, and SIG4 references Join #21.

Version:

SIMPL+ Version 2.00

#STRING_INPUT_JOIN

Name:

#STRING_INPUT_JOIN

Syntax:

```
#STRING_INPUT_JOIN<constant>
```

Description:

Changes the join number starting with the next STRING_INPUT or BUFFER_INPUT definition to the join number specified by <constant>.

Example:

```
STRING_INPUT SIG1[20], SIG2[20], SIG3[20], SIG4[20];
BUFFER_INPUT B1$[20]
```

In this example, SIG1 references Join #1, SIG2 references Join #2, SIG3 references Join #3, SIG4 references Join #4, and B1$ references Join#5.

```
STRING_INPUT SIG1[20], SIG2[20];
#STRING_INPUT_JOIN 20
STRING_INPUT SIG3[20], SIG4[20];
BUFFER_INPUT B1$[20]
```

SIG1 and SIG2 still reference Join #1 and Join #2, but SIG3 has been changed to reference Join #20, SIG4 references Join #21, and B1$ references Join#22.

Version:

SIMPL+ Version 2.00

#STRING_OUTPUT_JOIN

Name:

#STRING_OUTPUT_JOIN

Syntax:

```
#STRING_OUTPUT_JOIN<constant>
```

Description:

Changes the join number starting with the next STRING_OUTPUT definition to the join number specified by <constant>.

Example:

```
STRING_OUTPUT SIG1, SIG2, SIG3, SIG4;
```

In this example, SIG1 references Join #1, SIG2 references Join #2, SIG3 references Join #3, and SIG4 references Join #4.

```
STRING_OUTPUT SIG1, SIG2;
#STRING_OUTPUT_JOIN 20
STRING_OUTPUT SIG3, SIG4;
```

SIG1 and SIG2 still reference Join #1 and Join #2, but SIG3 has been changed to reference Join #20, and SIG4 references Join #21.

Version:

SIMPL+ Version 2.00

CEN-OEM-Specific Definitions

CEN-OEM Specific Definitions Overview

The CEN-OEM has one serial port which is used to communicate with a destination device. SIMPL+ defines several special purpose variables exclusively to work with the CEN-OEM to manipulate this serial port. These variables are only valid when the file is saved with an OEM extension. Each OEM variable has a specific type (DIGITAL_INPUT, etc.) to which all the same rules as any other variable declared of that type have.

In the following examples, the “device” port is the port that talks to the equipment (device) being controlled. The “main” port is the computer port of the CEN-OEM. This port is usually used for communicating with a host computer for maintenance, but various pins may be used for other applications as shown in the following examples.

_OEM_BREAK

Name:

_OEM_BREAK

Syntax:

```
_OEM_BREAK = <expression>;  // Write to Variable
```

or any expression that can use a variable as part of its contents.

Description:

When set to a non-zero value, causes a short break to be transmitted on the port. A Short break is 17-20 bits of logic low. When the system is done generating the short break, it will set the variable to 0. The variable may also be read back from to determine its current status. It is treated the same as a DIGITAL_OUTPUT.

Example:

```
_OEM_BREAK = 1; // Generate A Short Break
```

Version:

SIMPL+ Version 2.00

_OEM_CD

Name:

_OEM_CD

Syntax:

Any expression that can use a variable as part of its contents.

Description:

This variable is treated as a DIGITAL_INPUT and may be read from only. CD is the acronym for Carrier Detect. When a modem is hooked up to an RS-232 port and a connection (carrier) is made, the modem typically drives this pin high to let the connected hardware know that a data connection is present. This line may be used for other purposes depending on the hardware connected to the CEN-OEM.

Example:

```
PUSH _OEM_CD
{
PRINT(“Carrier Detect Pin has gone high!\n”);
}
```

Version:

SIMPL+ Version 2.00

_OEM_CTS

Name:

_OEM_CTS

Syntax:

Any expression that can use a variable as part of its contents.

Description:

This variable is treated as a DIGITAL_INPUT and may be read from only. CTS is the acronym for Clear To Send. In flow control for handshaking, a device will typically control this line, and raise it high when the CEN-OEM is allowed to transmit, and drop it low when it wants the CEN-OEM to stop transmitting.

It can also be used in other situations besides flow control, and in these situations, the CEN-OEM can monitor the status of the line directly through this pin.

Example:

```
PUSH _OEM_CTS
{
PRINT("CTS Pin has gone high!\n");
}
```

Version:

SIMPL+ Version 2.00

_OEM_DTR

Name:

_OEM_DTR

Syntax:

```
_OEM_DTR = <value>;
```

or any expression that can use a variable as part of its contents.

Description:

When set to a non-zero value, raises the DTR pin high. This pin is typically used to signify "Data Terminal Ready", which means that the CEN-OEM is telling an external piece of equipment that it is online and ready to function. The pin may be used for other purposes (or not at all). This value is treated as a DIGITAL_OUTPUT and may be read.

Example:

```
PUSH _OEM_CTS
{
PULSE(500, _OEM_DTR);
}
```

The above example will pulse the DTR pin for 5-seconds when the CTS line goes high.

Version:

SIMPL+ Version 2.00

_OEM_LONG_BREAK

Name:

_OEM_LONG_BREAK

Syntax:

```
_OEM_LONG_BREAK = <expression>;
```

or any expression that can use a variable as part of its contents.

Description:

When set to a non-zero value, causes the start of a break being transmitted on the port. A break is continuous logic low being generated on the port. In order to stop break generation, the variable should be set to 0. The variable may also be read back from to determine its current status. It is treated the same as a DIGITAL_OUTPUT.

If break generation is in progress and data transmission on _OEM_STR_OUT will be ignored.

Example:

```
PUSH _OEM_CTS
{
_OEM_LONG_BREAK = 1;
WAIT(100)
_OEM_LONG_BREAK=0;
}
```

In this example, the break is generated for 1-second when the CTS pin is driven high.

Version:

SIMPL+ Version 2.00

_OEM_MAX_STRING

Name:

_OEM_MAX_STRING

Syntax:

```
_OEM_MAX_STRING = <expression>;
```

or any expression that can use a variable as part of its contents.

Description:

Controls the maximum embedded packet size that is transmitted on the Ethernet port. This variable is treated the same as ANALOG_OUTPUT. The default is 250 bytes but it is recommended that this value not be changed for most applications.

Example:

```
_OEM_MAX_STRING = 1000;
```

In this example, the maximum embedded packet size is changed to 1000 bytes.

Version:

SIMPL+ Version 2.00

_OEM_PACING

Name:

_OEM_PACING

Syntax:

```
_OEM_PACING = <expression>;
```

or any expression that can use a variable as part of its contents.

Description:

Controls the number of milliseconds the system will delay between sending bytes in a given string. This variable is treated the same as ANALOG_OUTPUT. The maximum value allowed is 255 (250ms). Values greater than 255 will use the lower byte of the number.

Example:

```
CHANGE _OEM_STR_IN
{
IF(_OEM_STR_IN = “\x01\x02”)
{
_OEM_STR_OUT = “\x02ACK\x03”;
CLEARSTRING(_OEM_STR_IN);
}
}

FUNCTION MAIN()
{
_OEM_PACING = 10;
}
```

In this example, the pacing is set to 10ms. When the string “\x01\x02” comes into the port, a 5-byte string is sent out the port. The system waits 10ms after generating each character before sending the next one.

Version:

SIMPL+ Version 2.00

_OEM_RTS

Name:

_OEM_RTS

Syntax:

```
_OEM_RTS = <expression>;
```

or any expression that can use a variable as part of its contents.

Description:

This variable is treated the same as DIGITAL_OUTPUT. In a program where hardware handshaking is not being used, the program may control the RTS pin for its own application. Writing a non-zero value to this variable sets the RTS pin high, writing 0 sets it low.

Example:

```
PUSH _OEM_CTS
{
DELAY(10);
_OEM_RTS = 1;
}
```

In this program, the RTS pin will be driven high by the CEN-OEM 0.1-seconds after the CTS pin is driven high by an external system.

Version:

SIMPL+ Version 2.00

_OEM_STR_IN

Name:

_OEM_STR_IN

Syntax:

Any expression where a BUFFER_INPUT is legal.

Description:

This variable is treated the same as BUFFER_INPUT and reflects data coming into the CEN-OEM input buffer. The buffer is 255 bytes wide.

Example:

```
INTEGER I;

CHANGE _OEM_STR_IN
{
FOR(I=1 to len(_OEM_STR_IN))
IF(byte(_OEM_STR_IN, I) = 0x7F
_OEM_STR_OUT = “\x15”;
CLEARSTRING(_OEM_STR_IN);
}
```

In this example, whenever the input buffer changes, it is scanned for the character with the hex value of 0x7F. Each time it is present, a 0x15 is transmitted. The buffer is cleared at the end of the iteration.

Version:

SIMPL+ Version 2.00

_OEM_STR_OUT

Name:

_OEM_STR_OUT

Syntax:

Any expression where a BUFFER_OUTPUT is legal.

Description:

This variable is treated the same as BUFFER_OUTPUT and reflects data coming from the CEN-OEM input buffer. The buffer is 255 bytes wide.

Example:

```
INTEGER I;

CHANGE _OEM_STR_OUT
{
FOR(I=1 to len(_OEM_STR_OUT))
IF(byte(_OEM_STR_OUT, I) = 0x7F
_OEM_STR_IN = “\x15”;
CLEARSTRING(_OEM_STR_OUT);
}
```

In this example, whenever the input buffer changes, it is scanned for the character with the hex value of 0x7F. Each time it is present, a 0x15 is transmitted. The buffer is cleared at the end of the iteration.

Version:

SIMPL+ Version 2.00

Index

Software License Agreement

If You are a business or organization, You agree that upon request from Crestron or its authorized agent, You will within thirty (30) days fully document and certify that use of any and all Software at the time of the request is in conformity with Your valid licenses from Crestron of its authorized agent.

Without prejudice to any other rights, Crestron may terminate this Agreement immediately upon notice if you fail to comply with the terms and conditions of this Agreement. In such event, you must destroy all copies of the Software and all of its component parts.

PROPRIETARY RIGHTS

Copyright. All title and copyrights in and to the Software (including, without limitation, any images, photographs, animations, video, audio, music, text, and "applets" incorporated into the Software), the accompanying media and printed materials, and any copies of the Software are owned by Crestron or its suppliers. The Software is protected by copyright laws and international treaty provisions. Therefore, you must treat the Software like any other copyrighted material, subject to the provisions of this Agreement.

Submissions. Should you decide to transmit to Crestron's website by any means or by any media any materials or other information (including, without limitation, ideas, concepts or techniques for new or improved services and products), whether as information, feedback, data, questions, comments, suggestions or the like, you agree such submissions are unrestricted and shall be deemed non-confidential and you automatically grant Crestron and its assigns a non-exclusive, royalty-tree, worldwide, perpetual, irrevocable license, with the right to sublicense, to use, copy, transmit, distribute, create derivative works of, display and perform the same.

Trademarks. CRESTRON and the Swirl Logo are registered trademarks of Crestron Electronics, Inc. You shall not remove or conceal any trademark or proprietary notice of Crestron from the Software including any back-up copy.

GOVERNING LAW

This Agreement shall be governed by the laws of the State of New Jersey, without regard to conflicts of laws principles. Any disputes between the parties to the Agreement shall be brought in the state courts in Bergen County, New Jersey or the federal courts located in the District of New Jersey. The United Nations Convention on Contracts for the International Sale of Goods, shall not apply to this Agreement.

CRESTRON LIMITED WARRANTY

CRESTRON warrants that: (a) the Software will perform substantially in accordance with the published specifications for a period of ninety (90) days from the date of receipt, and (b) that any hardware accompanying the Software will be subject to its own limited warranty as stated in its accompanying written material. Crestron shall, at its option, repair or replace or refund the license fee for any Software found defective by Crestron if notified by you within the warranty period. The foregoing remedy shall be your exclusive remedy for any claim or loss arising from the Software.

CRESTRON shall not be liable to honor warranty terms if the product has been used in any application other than that for which it was intended, or if it as been subjected to misuse, accidental damage, modification, or improper installation procedures. Furthermore, this warranty does not cover any product that has had the serial number or license code altered, defaced, improperly obtained, or removed.

Notwithstanding any agreement to maintain or correct errors or defects Crestron, shall have no obligation to service or correct any error or defect that is not reproducible by Crestron or is deemed in Crestron's reasonable discretion to have resulted from (1) accident; unusual stress; neglect; misuse; failure of electric power, operation of the Software with other media not meeting or not maintained in accordance with the manufacturer's specifications; or causes other than ordinary use; (2) improper installation by anyone other than Crestron or its authorized agents of the Software that deviates from any operating procedures established by Crestron in the material and files provided to You by Crestron or its authorized agent; (3) use of the Software on unauthorized hardware; or (4) modification of, alteration of, or additions to the Software undertaken by persons other than Crestron or Crestron's authorized agents.

ANY LIABILITY OF CRESTRON FOR A DEFECTIVE COPY OF THE SOFTWARE WILL BE LIMITED EXCLUSIVELY TO REPAIR OR REPLACEMENT OF YOUR COPY OF THE SOFTWARE WITH ANOTHER COPY OR REFUND OF THE INITIAL LICENSE FEE CRESTRON RECEIVED FROM YOU FOR THE DEFECTIVE COPY OF THE PRODUCT. THIS WARRANTY SHALL BE THE SOLE AND EXCLUSIVE REMEDY TO YOU. IN NO EVENT SHALL CRESTRON BE LIABLE FOR INCIDENTAL, CONSEQUENTIAL, SPECIAL OR PUNITIVE DAMAGES OF ANY KIND (PROPERTY OR ECONOMIC DAMAGES INCLUSIVE), EVEN IF A CRESTRON REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR OF ANY CLAIM BY ANY THIRD PARTY. CRESTRON MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO TITLE OR INFRINGEMENT OF THIRD-PARTY RIGHTS, MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY OTHER WARRANTIES, NOR AUTHORIZES ANY OTHER PARTY TO OFFER ANY WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY FOR THIS PRODUCT. THIS WARRANTY STATEMENT SUPERSEDES ALL PREVIOUS WARRANTIES.

Return and Warranty Policies

Merchandise Returns / Repair Service

1. No merchandise may be returned for credit, exchange, or service without prior authorization from CRESTRON. To obtain warranty service for CRESTRON products, contact the factory and request an RMA (Return Merchandise Authorization) number. Enclose a note specifying the nature of the problem, name and phone number of contact person, RMA number, and return address.
2. Products may be returned for credit, exchange, or service with a CRESTRON Return Merchandise Authorization (RMA) number. Authorized returns must be shipped freight prepaid to CRESTRON, Cresskill, N.J., or its authorized subsidiaries, with RMA number clearly marked on the outside of all cartons. Shipments arriving freight collect or without an RMA number shall be subject to refusal. CRESTRON reserves the right in its sole and absolute discretion to charge a 15% restocking fee, plus shipping costs, on any products returned with an RMA.
3. Return freight charges following repair of items under warranty shall be paid by CRESTRON, shipping by standard ground carrier. In the event repairs are found to be non-warranty, return freight costs shall be paid by the purchaser.

CRESTRON Limited Warranty

CRESTRON ELECTRONICS, Inc. warrants its products to be free from manufacturing defects in materials and workmanship under normal use for a period of three (3) years from the date of purchase from CRESTRON, with the following exceptions: disk drives and any other moving or rotating mechanical parts, pantilt heads and power supplies are covered for a period of one (1) year; touchscreen display and overlay components are covered for 90 days; batteries and incandescent lamps are not covered.

This warranty extends to products purchased directly from CRESTRON or an authorized CRESTRON dealer. Purchasers should inquire of the dealer regarding the nature and extent of the dealer's warranty, if any.

CRESTRON shall not be liable to honor the terms of this warranty if the product has been used in any application other than that for which it was intended, or if it has been subjected to misuse, accidental damage, modification, or improper installation procedures. Furthermore, this warranty does not cover any product that has had the serial number altered, defaced, or removed.

This warranty shall be the sole and exclusive remedy to the original purchaser. In no event shall CRESTRON be liable for incidental or consequential damages of any kind (property or economic damages inclusive) arising from the sale or use of this equipment. CRESTRON is not liable for any claim made by a third party or made by the purchaser for a third party.

CRESTRON shall, at its option, repair or replace any product found defective, without charge for parts or labor. Repaired or replaced equipment and parts supplied under this warranty shall be covered only by the unexpired portion of the warranty.

Except as expressly set forth in this warranty, CRESTRON makes no other warranties, expressed or implied, nor authorizes any other party to offer any warranty, including any implied warranties of merchantability or fitness for a particular purpose. Any implied warranties that may be imposed by law are limited to the terms of this limited warranty. This warranty statement supersedes all previous warranties.

Trademark Information
All brand names, product names, and trademarks are the sole property of their respective owners. Windows is a registered trademark of Microsoft Corporation. Windows95/98/Me/XP and WindowsNT/2000 are trademarks of Microsoft Corporation.

Crestron Electronics, Inc.
15 Volvo Drive Rockleigh, NJ 07647
Tel: 888.CRESTRON
Fax: 201.767.7576
www.crestron.com

Language Reference Guide – DOC. 5797G
04.03

Specifications subject to
change without notice