Crestron® SIMPL+® Software

Programming Guide

Crestron Electronics, Inc.

The original language version of this document is U.S. English.
All other languages are a translation of the original document.

Contents

Introduction

This document describes the core concepts of Crestron® SIMPL+® software and provides instructions and examples for programming in SIMPL+.

What is SIMPL+?

SIMPL+ is a language extension to SIMPL programming software. With SIMPL+, it is possible to use a procedural C-like language to code elements of the program that would be difficult or impossible using SIMPL alone. A SIMPL+ program is a module that interacts with a control system, and it requires two essential elements.

The first element is a starting point, which is needed for two reasons:

- It serves as a convenient place to initialize any global variables that are declared within the module.
- Any functionality that the module needs to perform on its own (instead of being triggered though an event) can be instantiated here.

The second element is event processing. Events are functions that are triggered through input signals from the control system. Input and output (I/O) signals can be either digital, analog, or serial and are declared within the SIMPL+ module. Events tell the SIMPL+ module that something has changed within the control system and allows the module to perform any action accordingly. In order for a SIMPL+ module and a control system to interact, they must be able to send and receive signals to and from one another. I/O signals are tied directly to the control system. Input signals are sent from the control system and are received within the SIMPL+ module. Output signals are sent from the SIMPL+ module to the control system.

Who is the Audience for this Document?

The Crestron SIMPL+ Software Programming Guide is appropriate for a novice SIMPL+ programmer or for an expert programmer looking for a refresher course. This document assumes the programer has a working knowledge of the SIMPL programming environment, including the ability to configure a new program (define the hardware) and to interconnect user interfaces (such as a touch screen) and system outputs (such as a relay). Knowledge of the SIMPL logic symbols is not required.

When Should I Use SIMPL+?

Most tasks can be accomplished in either SIMPL or SIMPL+. However, Crestron® control system programming is most effective when the strengths of both environments are leveraged.

Almost every Crestron program will have some elements of SIMPL. Any time a button is needed to act as a toggle, or it is necessary to interlock a group of source buttons, it is generally easier to handle these tasks with SIMPL.

SIMPL+ is best used for more complex and algorithmic tasks, such as building complex strings, calculating checksums, or parsing data coming from another device. Complex decision making, especially when dealing with time and date, is generally much easier to handle in SIMPL+. Data storage and manipulation may also be better suited to SIMPL+ than to SIMPL.

What is Required to Use SIMPL+?

SIMPL+ version 2.0 requires a CNX-series control processor and SIMPL v1.23 or later.

SIMPL+ version 3.0 accompanies SIMPL v2.00 or later, and may be used to program a CNX-series control system, a 2-Series control system, a 3-Series® control system, a 4-Series™ control system, or later.

> **NOTE:** SIMPL+ is supported on Crestron Virtual Control (VC-4) as of VC-4 software version 4.0000.00007.

Where Can I Get More Information?

This document should contain all the information required to program in SIMPL+. For specific information about the language syntax, refer to the SIMPL+ Language Reference Guide.

Write Your First SIMPL+ Module

This section describes how to create, modify, deploy, and test an example SIMPL+ module.

Although programs can be written in SIMPL+, all control system I/O must be defined within SIMPL. This SIMPL program acts as a shell that contains SIMPL+ modules. In most cases, the SIMPL program consists of hardware definitions and raw SIMPL code. SIMPL+ modules appear as logic symbols in the overall SIMPL program.

Because SIMPL+ modules can exist only inside this wrapper, a skeleton SIMPL program must be created before testing the SIMPL+ module. Refer to Deploy the SIMPL+ Module on page 5 for more information.

Create a New SIMPL+ Module

To create a new SIMPL+ module within SIMPL:

1. Open SIMPL software.
2. Navigate to **File > New > New SIMPL+ Module**. The SIMPL+ programming environment is displayed.

   > **NOTE:** The SIMPL+ programming environment shows a skeleton program with commented code instead of an empty window.

3. Locate the required lines of code, uncomment them, and add the appropriate code as described in the following sections.

   > **NOTE:** To uncomment a line of code, either remove the `//` that appears at the start of the line, or remove the multiline comment indicators `/*…*/`.

Modify the Code

SIMPL+ modules communicate with the SIMPL wrapper program via inputs and outputs. These inputs and outputs correspond to signals in SIMPL and can be digital, analog, or serial signals (these terms are covered in more detail in Deploy the SIMPL+ Module on page 5).

For this first module, only a single digital input is defined. Uncomment the line of code that says `// DIGITAL_INPUT` and then edit it as follows:

```
DIGITAL_INPUT speak;
```

The code above defines the variable **speak** as the first digital input to the SIMPL+ module. Most lines, and all statements, in SIMPL+ end in a semicolon (`;`).

> **NOTE:** The definition of a statement in SIMPL+ can be found in the SIMPL+ Language Reference Guide.

Next, uncomment the line of code that says `// #PRINT_TO_TRACE` and then edit it as follows:

```
#PRINT_TO_TRACE
```

When a digital input goes from low to high, a push event is generated. To define a push event function for that signal, program this function to yield the desired actions. From the skeleton program, find the commented line of code that says `PUSH input`. Uncomment the function and edit it as follows:

```
PUSH speak
{
    Print( "Hello world!\n" );
}
```

This function sends the string **hello world** with a carriage return and line feed out of the control system computer port when the signal speak goes high. Notice the curly braces (`{}`) surrounding the `Print` statement above. In SIMPL+, these braces are used to group multiple statements into a compound statement. In the case of a function definition, always surround the contents of the function with these braces.

The next step is to add another event function, one that responds when a signal goes from high to low. This event is called a release event. Uncomment the line of code in the skeleton program that says `RELEASE input` and edit it as follows:

```
RELEASE speak
{
    Print( "Crestron programmers make a difference\n" );
}
```

Finally, define what happens when the control system first boots up with **Function Main**. Upon system startup, the program code defined in this function executes. Unless there are looping constructs (discussed in Controlling Program Flow: Loops on page 36) defined in this function, this code executes only once when control system is first started (or after it is rebooted). Find the section of the skeleton program that says `Function Main ()`, and edit as follows:

```
Function Main ()
{
    Print( "I am born!\n" );
}
```

This function sends the text **I am born!** out of the computer port only upon startup.

Save and Compile the Module

To save the new module, navigate to **File > Save** from the program window. Assign the name **My first SIMPL+** to the file.

To compile the file, navigate to **Build > Save and Compile** from the program window. This selection saves the code module, compiles it, and tells SIMPL how to present it to the SIMPL programmer.

SIMPL+ version 2.0 requires that all SIMPL+ modules reside in the **User SIMPL+** directory. View this directory in SIMPL by navigating to **Options > Preferences**, and then select the **Directories** tab. In SIMPL+ 3.0 and later, SIMPL+ modules and SIMPL programs reside in the corresponding **SIMPL Project Directory**. Each time the program is saved, an update log appears at the bottom of the screen to show the results of the save, compile, and update process. The window should display something similar to this code:

```
Compiling c:\Crestron\simpl\usrsplus\my first simpl+.usp
Total Error(s): 0
Total Warning(s): 0
SIMPL+ file saved successfully
No errors found: SIMPL Windows Symbol Definition updated
```

The example SIMPL+ module is now complete and ready for testing.

Deploy the SIMPL+ Module

This section describes how to make the example SIMPL+ module created in Create a New SIMPL+ Module on page 3 work inside a Crestron control system and how to set up this program in SIMPL.

Create the Skeleton SIMPL Program

To create the skeleton SIMPL program that will house the SIMPL+ module:

1. Create a new SIMPL program and enter the required information in the **Program Header Information** window. Make sure to select the relevant information from the **Program ID Tag** and **Control Processor** drop-down menus.

   For this example, use SIMPL Debugger to trigger the digital input. As a result, there is no need to define a touch screen or another user interface device, although it is preferred if one is available for testing.

   > **NOTE:** Only CNX-series control systems and later are supported by SIMPL+ software. No earlier control system will appear in the **Control Processor** drop-down menu.

2. After the SIMPL program is configured, switch to the **Program Manager** by navigating to **Project > Program System**. Ensure that the **Symbol Library** pane is visible on the left side of the screen.
3. Find and open the **User Modules** folder. An icon representing the SIMPL+ module written in the previous section is displayed.
4. Drag this icon into the **Logic** folder in the **Program View** pane. The SIMPL+ module is added as another symbol in the SIMPL program.
5. Double-click on the logic symbol to bring it into the **Detail View** window. The logic symbol should have a single input, labeled **speak**. This input corresponds with the declarations section of the SIMPL+ code, where only a single input and no outputs were defined.
6. Define a signal for this input. Call the signal name **test_me**. If a user interface was defined in an earlier step, assign this same signal to a button press.
7. Compile the program by selecting the compile toolbar button or navigate to **Project > Convert/Compile**. The compile process automatically recognizes that there is a SIMPL+ module in the program and compiles it along with the SIMPL code.
8. After compilation, select **Yes** when prompted to transfer the program to a connected control system. The SIMPL code section is sent first (followed by the save permanent memory image). Once completed, the SIMPL+ module code is sent along with the SIMPL code base.

> **NOTE:** After compilation in 2-Series control systems and older, the save permanent memory image is sent after the SIMPL code section.

Test the SIMPL+ Module

The SIMPL program with the example SIMPL+ module code has now been loaded into the control system and is ready to be tested.

Start SIMPL Debugger by opening Crestron Toolbox™ software and selecting **Tools > SIMPL Debugger**. In the bottom left corner of the **Debugger** window, select the control system running the SIMPL program from the address bar. Select **Yes** in the **Synchronize Signal** window.

To test the program, drive the signal to the high state by selecting it from the left pane and selecting the **Rising Edge** button from the **Stimulus** window above the signal tree. Driving the signal to the high state triggers the push event. In the right pane, known as the **Trace Window**, the events that occur as a result of the button press display. The string **Hello world!** is displayed under the **Value** tab.

Select the **Falling Edge** button to drive the signal low and trigger the release event. In the **Trace** window, the string **Crestron programmers make a difference** is displayed.

By selecting the **Positive Pulse** button, both strings appear one after the other, since the push and release events are triggered in rapid succession.

> **NOTE:** There are multiple ways to manually trigger the events above. Using the **Momentary Press** button in the **Stimulus** window, for instance, will trigger multiple events that display in the **Trace Window**. The various strings discussed above will all appear under the **Value** tab in as separate events.

The startup text **I am born** is displayed before SIMPL Debugger was started because **Function Main** only runs on system startup. To see it now, restart the control system by selecting **Options > Reset Rack**.

The Structure of a SIMPL+ Module

This section provides an overview of the code structure that makes up a SIMPL+ module in the typical order that it is used.

Compiler Directives

Compiler directives provide explicit instructions to the compiler and should appear at the beginning of the module. These elements are not part of the SIMPL+ language itself. These directives are distinguished from actual SIMPL+ code by preceding them with a pound sign (#).

Currently, there are thirty compiler directives, nineteen of which are provided in the template file that is created when a new module is started. The compiler directives are as follows:

- **#SYMBOL_NAME:** Allows the user to specify the name that SIMPL uses for this module. If this directive is left out, the filename will be used by default.
- **#HINT:** Provides text that appears in the SIMPL status bar whenever the module icon is selected.
- **#CATEGORY:** (SIMPL+ 3.0 and later) Specifies the SIMPL symbol tree category number for this SIMPL+ module, which controls where the SIMPL+ module is listed in the symbol tree in **Program Manager**. Selecting **Edit** > **Insert Category** from the menu will display a list of available categories to choose from and will insert the selected category in to the program module automatically.
- **#DEFAULT_NONVOLATILE:** (SIMPL+ 3.0 and later) Specifies that all program variables will retain their values if hardware power is lost. If neither the **#DEFAULT_VOLATILE** nor **#DEFAULT_NONVOLATILE** are specified, the compiler will default all variables declared within the SIMPL+ module as nonvolatile in CNX-series control systems. In 2-Series, 3-Series, and 4-Series control systems (or later), the compiler will default all variables as volatile.
- **#DEFAULT_VOLATILE:** (SIMPL+ 3.0 and later) Program variables will not retain their value if hardware power is lost.
- **#HELP_BEGIN / #HELP_END:** Allows online help to be entered for this module. This text appears when the user selects the module and presses **F1** from within SIMPL.
- **#DEFINE_CONSTANT:** Allows constant numeric/string values to be assigned to alphanumeric names. This is useful for writing changeable and readable code.

Using constant definitions are a very important part of writing readable code. Take the following example:

```
PUSH vcr_select
{
    switcher_input = 3;
    switcher_output = 2; // video projector
}

PUSH dvd_select
{
    switcher_input = 4;
    switcher_output = 2; // video projector
}
```

In this example, the value of the variable `switcher_input` is set to `3` if the **VCR** button is pressed or `4` if the **DVD** button is pressed. In both cases, `switcher_output` is set to `2`, which is the output connected to the video projector. In a larger module, these variables would be used somewhere else to generate a command string to control a switcher. Using numbers in a small program like this still produces a readable program, though there are problems with it. For instance, If the switcher configuration is changed and the inputs and outputs are rearranged, the user must change all the appropriate values in the program for the switcher input and output.

Examine the following equivalent program, which uses constant definitions in place of actual numbers:

```
#DEFINE_CONSTANT VCR_INPUT 3
#DEFINE_CONSTANT DVD_INPUT 4
#DEFINE_CONSTANT VPROJ_OUTPUT 2

PUSH vcr_select
{
    switcher_input = VCR_INPUT;
    switcher_output = VPROJ_OUTPUT; // video projector
}

PUSH dvd_select
{
    switcher_input = DVD_INPUT;
    switcher_output = VPROJ_OUTPUT; // video projector
}
```

In this version of the program, the code is more readable, and it is clear that changing a numeric value in one place (the `#DEFINE_CONSTANT`) can affect the value everywhere in the program. Using capital letters for constant definitions is not required, but it clarifies the difference between variables and constants when reading through a program.

Include Libraries

Libraries are a means of grouping common functions into one source file to enable modularity and reusability of source code. Libraries are different from modules in that they do not contain a starting point (**Function Main**), and they cannot interact with the control system (through I/O signals and events). Libraries can include other libraries, but they cannot include a SIMPL+ module. Only functions and defined constants are allowed to be declared and defined within libraries. Global variable declarations are not allowed. Functions, however, can contain local variables. Other advantages are:

1. **Modularity**: Large SIMPL+ module are better organized by placing sections of code into a **User-Library**. Crestron recommends creating libraries that contain sets of related functions. For example, a library might be created that contains only functions that perform certain math related functions. Another library might be created that contains functions performing special string parsing routines.
2. **Reusability**: SIMPL+ modules may need pieces of functionality that were written in other modules. Common portions of code can be extracted and placed into one or more libraries. Once placed into a library, one or more SIMPL+ modules can include and make use of them.

SIMPL+ modules include libraries using the following syntax:

```
#USER_LIBRARY "<library_name>"
#CRESTRON_LIBRARY "<library_name>"
#INCLUDEPATH "<absolute_path or reltive_path>"
```

> **NOTE:** `library_name` is the name of the library without the file extension.

User-Libraries are libraries that the end user writes. These can exist either in the SIMPL+ module's project directory or in the **User SIMPL+** directory (set in SIMPL).

Crestron-Libraries are provided from Crestron and are contained within the Crestron Database.

`#INCLUDEPATH` directs the compiler to search for user libraries in the specified paths.

Variable Declarations

Variables act as storage areas to keep data. When writing all but the most basic of programs, users need to use variables to store values.

This section describes the different types of variables in SIMPL+ and how to define them. Any variable used in a SIMPL+ module must be declared before it is used. This declaration also tells the operating system how much space must be reserved to hold the values of these variables.

Inputs, Outputs, and Parameters

SIMPL+ modules communicate with the SIMPL program in which they are placed through input variables, output variables, and parameter values. This is similar in concept to the **Define Arguments** symbol used in SIMPL macros.

Input variables can be of three types: digital, analog, and string types. These correspond directly to the same signal types in SIMPL and the buffer input, which is a special case of the string input. Output variables can only be of the digital, analog, or string variety.

Input variables are declared using the following syntax:

```
DIGITAL_INPUT <dinput1>,<dinput2>,…<dinputn>;
ANALOG_INPUT <ainput1>,<ainput2>,…<ainputn>;
STRING_INPUT <sinput1>[size],<sinput2>[size],...<sinputn>[size];
BUFFER_INPUT <binput1>[size],<binput2>[size],...<binputn>[size];
```

NOTE: For more information on the `buffer_input`, refer to Working with Strings on page 60.

Digital and analog output variables are declared in the same way, except the word input is replaced with output, as shown below. String output variables do not include a size value. There is no output version of the buffer variable.

```
DIGITAL_OUTPUT <doutput1>,<doutput2>,…<doutputn>;
ANALOG_OUTPUT <aoutput1>,<aoutput2>,…<aoutputn>;
STRING_OUTPUT <soutput1>,<soutput2>,…<soutputn>;
```

The inputs and outputs declared in this way govern the appearance of the SIMPL+ symbols that are presented via SIMPL. The order of the signal declarations is important only within signal types. In SIMPL, digital signals always appear at the top of the list, followed by analogs, and then serials.

Variables

In addition to input and output variables, the user can define and use variables that are only seen by the SIMPL+ module. The SIMPL program that holds this module has no knowledge of these variables. In addition, any other SIMPL+ modules included in the SIMPL program would not have access to these variables.

NOTE: For more information on variables, refer to Working with Data (Variables) on page 18.

Declaring variables tells the SIMPL+ compiler how much memory to put aside to hold the workable data. These variable declarations are similar to input/output declarations. However, instead of digital, analog, and serial (string and buffer) types, integer and string variables are also available with the INTEGER and STRING datatype.

Integers are 16 bit quantities. For the 2-Series, 3-Series, and 4-Series control systems, 32-bit quantities are supported with the LONG_INTEGER datatype. Both INTEGER and LONG_INTEGER are treated as unsigned values. Signed versions for both of these datatypes are available by using the SIGNED_INTEGER and SIGNED_LONG_INTEGER datatypes.

The following example illustrates how each of these datatypes can be used within a program module:

```
INTEGER intA, intB, intC;
STRING stringA[10], stringB[20];
LONG_INTEGER longintA, longIntB;
SIGNED_INTEGER sintA, sintB;
SIGNED_LONG_INTEGER slongIntA;
```

All variables declared in this manner are nonvolatile. They remember their values when the control system power cycles or reinitializes. Since input/output variables are attached to signals defined in the SIMPL program, they are not nonvolatile unless the signals they are connected to are made nonvolatile through the use of special symbols.

Structures

> **NOTE:** Arrays are essential for understanding Structures. Refer to Arrays on page 26 for more information.

Sets of data are sometimes needed rather than individual pieces. Variables store a piece of data, but are not related to other variables in any way. Structures are used to group individual pieces of data together to form a related set.

Before structures can be used, a structure definition must be defined. Defining a structure is defining a custom datatype (such as STRING and INTEGER). Once this new type (the STRUCTURE) is defined, variables of that type can be declared.

The following example illustrates how a structure can be defined and used within a program module:

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

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

```
PUSH BUTTON
{
    OneEntry.Names="David";
    OneEntry.Address="100 Main Street";
    OneEntry.PhoneNumber="555-555-5555";
    OneEntry.Age="30";

    Entry[5].Names="Andrew";
    Entry[5].Address="15 Volvo Drive";
    Entry[5].PhoneNumber="555-555-0000";
    Entry[5].Age="23";
}
```

User-Defined Functions

Creating a user-defined function to perform common tasks solves the issues that arise when reusing large sections of code, such as when a change is required in the code and must be made in multiple places. A user-defined function is similar to a built-in function like `Date` or `MakeString`, with some important exceptions.

To invoke a user-defined function, use the following syntax:

```
CALL MyUserFunction();
```

Event Functions

Event functions make up the core of most SIMPL+ modules. Because control systems are event driven, most code is activated in response to certain events when they occur. Event functions allow the user to execute code in response to some change that has occurred to one or more of the input signals feeding the SIMPL+ module from the SIMPL program.

Event functions can be used with input variables only (not with locally defined variables), and they are only triggered by the operating system at the appropriate time. They cannot be called manually by the programmer.

Event functions are multitasking. An event can be triggered even if another event in the same SIMPL+ module is already processing. As described in Understanding Processing Order on page 65, this only happens if events are triggered on the same logic wave, or if one event function has caused a task switch.

The structure of an event function is as follows:

```
event_type <input list>
{
    <statements>
}
```

Three basic event types can occur in SIMPL+: **Push**, **Release**, and **Change**. A fourth type, called **Event**, is only used in specific circumstances. All event types are described below.

Push and Release Events

Push and **Release** events are valid only for `DIGITAL_INPUT` variables. The **Push** event is triggered when the corresponding digital input goes from a low to a high state (positive or rising edge). The **Release** event occurs when the signal goes from a high to a low state (negative or falling edge). For example, the following code sends a string instructing a camera unit to pan left when the left button is pressed and stop when the button is released.

```
DIGITAL_INPUT cam_up, cam_down, cam_left, cam_right;
STRING_OUTPUT camera_command;

PUSH cam_left
{
    camera_command = "MOVE LEFT";
}

RELEASE cam_left
{
    camera_command = "STOP";
}
```

> **NOTE:** This example assumes that the camera unit being controlled continues to move in a given direction until a stop command is issued. Some devices do not function in this manner.

Change Events

Change events are triggered by digital, analog, string, or buffer inputs. Anytime the corresponding signal changes its value, the **Change** event will be triggered. For digital signals, this means that the event will trigger on both the rising and falling edges (push and release). For buffer inputs, this event triggers when another character is added to the buffer.

The following example code sends a command to a CD player to switch to a different disc whenever the analog input `disc_number` changes value:

```
ANALOG_INPUT disc_number;
STRING_OUTPUT CD_command;

CHANGE disc_number
{
    CD_command = "GOTO DISC " + itoa(disc_number);
}
```

This program uses the `itoa` function to convert the analog value in `disc_number` into a string value which can be concatenated onto `CD_command`. The string concatenation operator (`+`) and system functions (such as `itoa`) are discussed in the SIMPL+ Language Reference Guide.

Compound Events

Compound events allow the same (or similar) action to occur when any of a number of other events occur. For example, there may be a need to generate a switcher command string each time any of a group of **output** buttons are pressed.

Compound events can be created in two ways. One way is to provide a list of input signals separated by commas in the event function declaration as shown in the following example:

```
PUSH button1, button2, button3

{
    <statements>
}
```

A second form of **Compound** event occurs when combining different types of events into a single function. For example, there may be a need to execute code when a button is pushed or the value of an analog signal changes. To accomplish this, stack the event function declarations as shown below:

```
CHANGE output_value
PUSH button1, button2

{
    <statements>
}
```

A single input can have more than one event function defined for it. This makes it possible to write one event function for a specific input only and another event function for a group of inputs as shown in the following example:

```
PUSH button1
{
    // code here only runs when
    // button1 goes high
}

PUSH button1, button2, button3
{
    // this code runs when any of
    // these inputs goes high
}
```

The Global Event

Event, a special form of event, is triggered when any of the inputs to a SIMPL+ module changes. This is simply a shortcut for having to build a compound event manually, and it includes all the inputs separated by commas in a `CHANGE` event declaration. Access this special event function by using the `EVENT` keyword as shown below:

```
EVENT
{
    // this code runs anytime anything
    // on the input list changes
}
```

If the user has a SIMPL+ module in which a change on any input causes the same code to execute, this type of event is useful. However, if additional inputs are added at a later time, it is executed when these new inputs change as well.

Function Main

`Main` is a special case of a user-defined function. The `Main` function is executed when the control system initializes and is never called again. In many cases, the `Main` function is used to initialize variables. It may not contain any statements at all.

```
Function Main()
{
    Integer x;
    String LocalString[50000];

    x = 0;

}
```

NOTE: Some `Main` functions in existing SIMPL+ modules may contain loops. However, Crestron does not recommend placing loops within `Main` functions, as this is not a programming best practice. For more information on loops, refer to Controlling Program Flow: Loops on page 36.

Working with Data (Variables)

Programming is the manipulation of data. Examples of data in a program can include a `switcher` input and output numbers, the name of a next speaker, and the amount of time left before a system shuts down automatically. This section covers the different data types available in SIMPL+.

Input/Output Types

Input/output variables are used to transfer data between SIMPL+ modules and the surrounding SIMPL program. Each input or output variable in SIMPL+ is connected to a signal in the SIMPL program. The table below shows the type of data conveyed by the three signal types.

SIMPL Signal Styles

Signal Type	Data	Example
Digital	Single bit	Button push/release
Analog	16-bit (0 to 65535)	Volume level
Serial	Up to 1024 bytes	Serial data input from a COM port

Depending on the application, it may be more convenient to generate an analog signal in SIMPL and connect it to a SIMPL+ module, rather than connecting a large number of digital signals and setting a variable based on which signal was pressed last.

Digital Inputs/Outputs

Digital signals comprise the majority of signals in a typical SIMPL program. In SIMPL+, they primarily trigger events on the rising or falling edge of the signal, though they can also be used in expressions.

The state (or value) of a digital signal is always either **1** or **0** (also referred to as **On** or **Off**). In SIMPL+, assigning a value of **0** to a digital signal turns it **Off**. Assigning it any nonzero value will turn it **On**. Crestron recommends using the value **1** for clarity.

Analog Inputs/Outputs

Analog signals in SIMPL are used to accomplish tasks for which digital signals are inadequate. Typical examples include volume control and camera pan or tilt control. In SIMPL+, analog signals provide an easy way of transferring data (16 bits at a time) into and out of SIMPL+ modules.

In SIMPL+, analog signals are treated similar to how they are in SIMPL. They are 16-bit numbers that can range between 0 and 65,535 (unsigned) or -32768 and +32,767 (signed). Signed and unsigned numbers are discussed further in Integers on page 21.

String Inputs/Outputs and Buffer Inputs

SIMPL+ stores serial data into temporary string variables. When a serial signal is connected to a SIMPL+ module and the SIMPL program causes data to be sent via this signal, the SIMPL+ module copies the data from this signal into local memory. The data is kept there until the SIMPL program changes it.

By storing the serial data into a string variable, SIMPL+ programmers can perform tasks on strings that were difficult or impossible using SIMPL alone. For example, it is easy to evaluate a string and add a `checksum` byte on the end to insert or remove characters from a string or to parse information out of a string for use elsewhere. Functions that are designed to work with string signals and string variables are discussed in the SIMPL+ Language Reference Guide.

> **NOTE:** Issues may arise when serial data streams in instead of appearing at one time. A string input is completely replaced each time new data is detected on the input. The buffer input, an alternate type of serial input type, may be used to solve this problem. Refer to Working with Strings on page 60 for more information on buffer inputs.

Signal Scope

Signals are global throughout the SIMPL program and within the SIMPL+ module they are connected to. In a SIMPL+ module, the values of digital, analog, and string inputs are read. However, their values cannot be changed from within SIMPL+, so they are considered read only. Buffer inputs can be read from and modified.

Digital and analog output signals in SIMPL+ can be read and modified. String outputs can be modified, but cannot be read back because of how the contents of an output variable are viewed. The value of any output is the value of the signal as seen by the outside SIMPL program at that instant. SIMPL+ does not propagate outputs to the SIMPL program each time they are changed in the program. As a general rule, assume that analog and serial outputs are propagated at the time they are assigned new values. Digital signals, however, are not propagated until a task switch occurs.

String outputs cannot be read due to the nature of serial signals in SIMPL. These signals do not actually store strings in them, but rather point to locations in memory where a string exists. Since the data stored at a particular location in memory can change at some later time, there is no guarantee that the string data is still there. As a result, SIMPL+ does not allow a string output to be examined.

Examine the following code example:

```
DIGITAL_OUTPUT d_out;
ANALOG_OUTPUT a_out;
STRING_OUTPUT s_out;

PUSH someEvent
{
    d_out = 1; // set this digital output to 'on'
    a_out = 2000; // set this analog output to 2000
    s_out = "hello"; // set this string output to "hello"

    if (d_out = 1) // this WILL NOT be true until the Print ("d_out is on\n"); // next
task-switch

    if (a_out = 2000)       // this WILL be true Print ("a_out = 2000");

    if (s_out = "hello")    // this WILL NOT be true due to the Print ("s_out is hello");
// nature of serial signals

    ProcessLogic(); // force a task-switch

    if (d_out = 1) // NOW this is true Print ("d_out is on\n");
}

Function Main()   // initialization
{
    d_out=0;
    a_out = 0;
}
```

In this example, the digital output, `d_out`, and the analog output, `a_out`, are set to `0` on system startup in the `Function Main`. In the `push` function, the first conditional `if` statement evaluates to `False` because the digital output signal, `d_out`, is considered `Off` until this value is propagated to the SIMPL program. With digital outputs, this does not happen until the SIMPL+ module performs a task switch. The analog and string outputs, on the other hand, are propagated as soon as they are assigned new values, so the second `if` condition evaluates to `True` and the subsequent `print` statement is executed. The third `if` statement can still evaluate to `False`, however, due to the nature of serial signals in SIMPL, as previously described.

The `ProcessLogic` function call forces a task switch from SIMPL+ to the SIMPL logic processor. This task switch causes the digital signal to be propagated out to the SIMPL program. The next time the logic processor passes control back to this SIMPL+ module, it picks up where it left off. As a result, the fourth `if` condition evaluates to `True` and executes the print statement.

> **NOTE:** The `if` language construct is described in detail in Controlling Program Flow: Branching on page 31. Evaluation of `True` and `False` expressions are covered in Operators, Expressions, and Statements on page 28.

All About Variables

In addition to input and output signals, additional variables can be declared that are only used inside the SIMPL+ module. The outside SIMPL program has no knowledge of these variables and no access to them. These variables are critical for use as temporary storage locations for calculations.

Unless otherwise specified with a compiler directive, all variables in SIMPL+ are volatile in 2-Series, 3-Series, and 4-Series (or later) control systems, which means that they do not remember their values if power is lost. The compiler directive, `#DEFAULT_NONVOLATILE`, can be used to change this behavior so that the variable's values are retained after power is lost. Variables in CNX-series control processors are nonvolatile by default. Crestron recommends initializing variables to some value before using them, except in the cases where it becomes necessary to take advantage of their nonvolatility. This can be accomplished within `Function Main`.

SIMPL+ allows for two different types of variables: integers and strings. In addition, variables of either type may be declared as one or two-dimensional arrays.

Integers

Integers contain 16-bit whole numbers. Their range is identical to the range of analog signals because analog signals are also treated as 16-bit values.

Integers are declared as follows:

```
INTEGER <int1>, <int2>,…,<intn>;
```

Depending on how they are used, integers can either be unsigned or signed. An understanding of signed and unsigned integers assists in the debugging process. Unsigned integers have values between 0 and 65,535. Signed integers have values between -32,768 and 32,767. These values cannot contain a decimal point.

The difference between signed and unsigned integers is how the control system views them. For any given value, the number can be thought of as being a signed number or an unsigned number. Depending upon which operations are performed on a number, the control system decides whether to treat that number as signed or unsigned.

When an integer has a value of between 0 and 32,767, it is identical whether it is considered signed or unsigned. However, numbers above 32,767 may be treated as negative numbers. If they are, they will have a value of (x - 65536), where x is the unsigned value of the number. This means that the value 65,535 has a signed value of -1, 65534 has a signed value of -2, and so forth. This scheme is referred to as two's complement notation.

In control system programming, negative numbers are rarely needed. As a result, the most common operations treat integers as unsigned. The following table lists operators and functions that are unsigned. Any operators or functions not shown here do not need special consideration.

NOTE: Signed operations have been depreciated from SIMPL+. Use the appropriate unsigned operations instead.

Unsigned Operators and Functions

Description	Unsigned Operators/Functions
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Integer division	/
Maximum	Max()
Minimum	Min()

Examine the following example:

```
INTEGER j, k;

Function Main()
{
    j = 2;
    k = -1; // this is the same as k = 65535

    if (j > k) // this will evaluate to FALSE
        Print( "j is bigger as unsigned numbers\n" );
}
```

In this example, the condition, `j > k`, evaluates to `False` and the `Print` statement does not execute. This is because the > operator performs an "unsigned greater than" operation. If both `j` and `k` are converted to unsigned values, `j` remains at `2`, but `k` becomes `65,535`, and thus `k` is not smaller than `j`.

Examine the following example:

```
INTEGER a, b, c, d;

Function Main()
{
    a = 100;
    b = -4;

    c = a / b; // c = 0 (100/65532)
    d = a / b; // d = -25 (100/-4)
}
```

c calculates to zero because the / operator is unsigned. It treats the variable b as +65,532. Since the / operator truncates the decimal portion of the result, c becomes zero. In regard to the variable d, b is treated as -4 and the result is -25.

> **NOTE:** If an operation results in a number that is greater than 65,535, then that number overflows. The value wraps around again starting at zero to allow certain operators (such as +, -, and *) to operate with no regard to sign.

Strings

String variables are used to hold multiple characters in a single variable. Typically, strings in SIMPL+ are used to hold items such as serial commands, database records, etc.

Strings are declared as follows:

```
STRING <string1[size]>, <string2[size]>,…,<stringn[size]>;
```

The number in square brackets following the variable name defines the size of the string variable. When declaring strings, choose a size that is large enough to hold any amount of data that might be needed. However, choosing too large a size wastes space. For example, it is unnecessary to set the variable size to 100 characters when a given variable in an application does not contain more than 50 characters.

To assign a value to a string, refer to the following example:

```
STRING myString[50];
myString = "Tiptoe, through the tulips\n";
```

In this example, a variable called myString is declared, which can contain up to 50 characters of data. The value, Tiptoe, through the tulips\n, is the value assigned to the variable. The double quotation marks surrounding the data defines a literal string expression, which is defined when the program is compiled and cannot change while the program is running. The \n at the end of this string is also literal to represent a newline or a carriage return followed by a line feed. Finally, square brackets were not included after the variable name, as was done when it was declared. When assigning a value to a string, that value is always assigned starting at the first character position.

> **NOTE:** For a complete list of shortcuts similar to \n, refer to the SIMPL+ Language Reference Guide.

The length of the value does not exceed total length allocated for myString (in this case, 50 characters). If the declared variable is not large enough to hold the data being assigned to it, the data is truncated to as many characters as the string can hold. When the program is executed, a string overflow error is reported within the control system's error log.

A common task in control system programming is to control an audio/video matrix switch. To control such a router, the input and output that make up the desired matrix crosspoint need to be specified. For example, assume the switch expects to see a command in a format as shown below:

```
IN<input#>OUT<output#><CR><LF>
```

This protocol allows the input and output to be specified as numbers. For example, to switch `input 4` to `output 10`, the command would be as follows:

```
IN4OUT10<CR><LF>
```

> **NOTE:** `<CR><LF>` represents the carriage return and line feed characters.

Instead of creating a literal string expression for each possible router crosspoint, build the control string dynamically as the program runs with the string concatenation operator (+). This is identical to the addition operator, but the SIMPL+ compiler detects whether integers are added or string expressions are concatenated as addressed in the following example:

```
DIGITAL_INPUT do_switch;
STRING_OUTPUT switcher_out[10];
INTEGER input, output;

PUSH do_switch
{
    switcher_out = "IN" + itoa(input) + "OUT" + itoa(output) + "\n";
}
```

In this example, the + operator is used to concatenate multiple string expressions. The `itoa` function has been used, which converts an integer value (in this case from analog input signals) into a string representation of that number. For instance, 23 becomes "23".

As an alternate way to build strings in SIMPL+, the `MakeString` function provides functionality similar to the concatenation operator while providing more power and flexibility. The following line is equivalent to the concatenation statement above:

```
MakeString( switcher_out, "IN%dOUT%d\n", input, output );
```

The first argument to the `MakeString` function, `switcher_out`, is the destination string, which is where the resulting string created by `MakeString` is placed. The second argument, the part embedded in double quotation marks, is the format specification. This determines the general form of the data. The constant parts of the string are entered directly.

Variable Scope

Variable declarations in SIMPL+ are either global or local. Global variables are defined in the **Define Variables** section of the code and exist throughout the entire SIMPL+ module. Any event function or user-defined function can reference and modify global variables. When the value of a global variable is being set or modified, it is reflected throughout the entire program.

Local variables are defined inside a function declaration and exist only inside that particular function. If a local variable, `byteCount`, was defined inside of a function, `CalcChecksum`, any reference to `byteCount` outside of the scope of this function will result in a compiler syntax error. Different functions can use the same variable names when defining local variables. Examine the following example:

```
// simple function to add up all the bytes in a
// string and append the sum as a single byte
// onto the original string.

String_Function CalcChecksum(STRING argData)
{
    INTEGER i, checksum;

    checksum = 0;

    for (i = 1 to len(argData))
        checksum = checksum + byte(argData,i);

    return (argData + chr(checksum));
}
```

In this example, `i` and `checksum` are local variables that only exist inside the function, `CalcChecksum`. This example also introduces an additional way to implement a local variable: by passing it as an argument to the function, as was done with the `STRING` variable, `argData`.

> **NOTE:** For more information on local variables and argument passing, refer to User Defined Functions on page 43.

Local variables can help keep your programs organized and easier to debug as opposed to using global variables. A disadvantage of global variables is that modifying a variable may have an adverse effect on another part of the program. Since local variables can only be used inside of a function, this issue will not occur.

Arrays

When INTEGER or STRING variables are declared, the user may also declare them as one or two-dimensional (integers only) arrays. An array is a group of data of the same type arranged in a table. A one-dimensional array acts as a single row with two or more columns, while a two-dimensional array acts as a table with multiple rows . In SIMPL+, arrays are declared as follows:

```
INTEGER myArray1[15]    // 1-D integer array with 16 elements
INTEGER myArray2[10][3] // 2-D integer array with 11x4 elements
STRING myArray3[8][50]  // 1-D string array with 9 elements
```

The first two lines above define 1D and 2D integer arrays, respectively. The last line appears to declare a 2D array of strings, yet the comments state that it declares a 1D array of strings. Recall that in Strings on page 23, it was necessary to define the maximum size of the string in square brackets, which is the same notation used for arrays. In the example above, a nine-element array of 50-byte strings is being declared. The user cannot declare a 2D array of strings in SIMPL+.

Declaring myArray1[15] creates an array with 16 elements instead of 15 because array elements start at 0 and go to the declared size (15 in this case). This convention makes for an easy transition to SIMPL+ for programmers familiar with other languages (some of which start at 0 and others which start at 1). If the programmer is comfortable with treating arrays as starting with element 0, then programming can be continued in this manner. However, if the programmer has used languages that treat the first element in an array as element 1, then the programmer may want to use that notation instead.

To reference a particular element of an array when programming, use the variable name followed by the desired element in square brackets. Using the arrays declared in the example above, the following statements are all valid in SIMPL+:

```
j = 5;                  // set an integer variable to 5
myArray1[3] = j;        // set the 3rd element of the array to 5
myArray1[j*2] = 100;    // set the 10th element of the array
                        // to 100

myArray2[j][1] = k;     // set the j,1 element of myArray2 to
                        // the value of k
m = myArray2[j][k-1];   // set the variable m to the value in
                        // the j,k-1 element of myArray2
myArray3[2] = "test";   // set the 3rd element of the string
                        // myArray3 to "test"
```

As shown in the previous examples, the user may use constants, variables, or expressions (discussed in Operators, Expressions, and Statements on page 28) inside of the brackets to access individual array elements. Array elements can appear on either side of the assignment (=) operator. They can be written to (left side) or read from (right side). Only one set of brackets was used to store a value into the string array `myArray3` even though two sets of brackets are needed when declaring the array. The first set of brackets in the declaration specified the size (in characters) of each string element. As discussed earlier, the size field is not included when referring to strings.

For example, refer to the following:

```
myString = "hello!";    // we do not use the size brackets here
                        // to assign a value to a string variable
```

As a result, when working with string arrays, only use one set of brackets, which refer to the array element, not the string size.

Operators, Expressions, and Statements

This sections covers the core programming elements in SIMPL+.

Operators

Operators take one or two operands and combine them to produce a result. In SIMPL+, operators are binary (takes two arguments) or unary (takes a single argument). For example, the **+** operator is binary (such as **x + y**), while the **-** operator can be binary or unary (**x - y** or **-x** are valid). Most operators in SIMPL+ are binary. Operands do not have to be simple constaints or variables. Instead they can be complex expressions that result in an integer.

SIMPL+ operators can be classified into three categories: arithmetic, bitwise, and relational. The sections below describe each category briefly. For a complete list of operators and their function, refer to the SIMPL+ Language Reference Guide.

Arithmetic Operators

Arithmetic operators are used to perform basic mathematical functions on one or two variables. In all but one case, these operations are only for integer types (includes analog inputs and outputs). For example, to add two integers, `x` and `y`, together, use the addition operator `+`, as follows:

```
x + y
```

The only exception to using operators with integers occurs when the `+` operator is used with string variables. In this case, the operator performs a concatenation instead of addition, which is useful for generating complex strings from smaller parts.

Bitwise Operators

Arithmetic operators deal with integers as a whole. Bitwise operators treat the individual binary bits of a number independently. For example, the unary operator `NOT` negates each bit in number, while the `&` operator performs a binary "and" operation to each bit in the arguments (bit0 with bit0, bit1 with bit1, and so forth).

Relational Operators

Relational operators are used in expressions when it is necessary to relate two values in some way (the exception to this is the unary operator `NOT`). When a comparison is done using a relational operator, the result is an integer, which represents `True` or `False`. In SIMPL+, `True` results equal `1` and `False` results equal `0`. In general, any nonzero value is considered by SIMPL+ to be `True`, while `False` is always `0`.

Typically, relational operators are used to help control the program flow. For more information on program flow, refer to Controlling Program Flow: Branching on page 31.

Expressions

An expression in SIMPL+ is anything that consists of operators and operands. Operands are the things on which operators act. For example, refer to the following expression.

```
x + 5
```

In this expression the operator is the addition operator (+), and the operands are `x` and `5`. Expressions can contain constants, variables, and function calls in addition to operators. One expression may be made up of many smaller expressions. The following are all valid SIMPL+ expressions:

```
max(x,15)
y * x << z
a = 3(
26 + byte(aString,i) mod z = 25
```

The equal sign in SIMPL+ expressions can have different meanings as demonstrated in the examples above. It can act as an assignment operator, where it assigns a value to a variable as shown with the value `3` and the variable `a` above. The equal sign can also act as an equivalency comparison operator, where it compares the value `a` to `3` to determine if the two are equal.

> **NOTE:** An expression cannot contain an assignment because it would become a statement, as discussed in Statements on page 30. It is recognized as a comparison operation.

Expressions always evaluate to either an integer or a string, as shown in the following example:

```
x + 5               // this evaluates to an integer
chr(i) + myString   // evaluates to a string
a = 3               // evaluates to 1 if true, 0 if false
b < c               // evaluates to 1 if true, 0 if false
```

The last two expressions are comparisons. Comparison operations always result in a `true` or `false` value. In SIMPL+, `true` expressions result in a value of `1` and `false` expressions result in a value of `0`. Understanding this concept is key to performing decision making in SIMPL+. Any expression that evaluates to a nonzero value is considered `true`. For more information, refer to Controlling Program Flow: Branching on page 31 and Controlling Program Flow: Loops on page 36.

Statements

Statements in SIMPL+ consist of function calls, expressions, assignments, or other instructions. Statements are either simple or complex. Simple statements end in a semicolon (`;`), as shown below:

```
x = MyInt / 10;                  // An assignment
print("hello, world!\n");        // A function call
checksum = atoi(MyString) + 5;   /* Assignment using function calls and operators */
```

A complex statement is a collection of simple statements surrounded with curly braces (`{}`), as shown below:

```
{ // start of a complex statement
    x = MyInt / 10;
    print("hello, world!\n");
    checksum = atoi(MyString) + 5;
} // end of a complex statement
```

Controlling Program Flow: Branching

In any substantial program, the program is controlled through decision making. SIMPL+ provides two constructs for branching the program based on the value of expressions: the `if-else` and `switch-case` statements.

if-else

`if-else` is the most commonly used branching construct. In its most basic form, it is structured as follows:

```
if (expression1)
{
    // do something here
}
```

`expression1` represents any valid SIMPL+ expression, including variables, function calls, and operators. If this expression evaluates to `True`, then the code inside the braces is executed. If this expression evaluates to `False`, then the code inside the braces is skipped.

As noted in Operators, Expressions, and Statements on page 28, expressions, which evaluate to a nonzero result, are considered `True`, and expressions that evaluate to `0` are considered `False`. For example, refer to the expressions in the following table:

Expressions

Expression	Evaluates to
a = 3	true if a=3, false otherwise
b*4 - a/3	true as long as the result is non-zero
1	always true
0	always false

One limitation with the `if` construct, as shown above, is that the code inside the `if` is run whenever `expression1` evaluates as `True`, but any code after the closing braces runs regardless. It is useful to execute one set of code when a condition is `True` and then another set of code if that same condition is `False`.

For this application, use the `if-else` construct as shown below:

```
if (expression1)
{
    // do something if expression1 is true
}

else
{
    // do something else if expression1 is false
}
```

> **NOTE:** Programming to anticipate user errors and handle them appropriately is called error-trapping. It is a recommended programming practice.

The programmer must ensure that the code following the `if` runs whenever `expression1` evaluates to `True` and the code following the `else` executes whenever `expression1` evaluates to `False`. There can never be a case where both sections of code execute together.

The following example is designed to control a CD changer. Before telling the CD player to go to a particular disc number, it checks to see that the analog value, which represents the disc number, does not exceed the maximum value.

```
#DEFINE_CONSTANT NUMDISCS 100

ANALOG_INPUT disc_number;
STRING_OUTPUT CD_command, message;

CHANGE disc_number
{
    if (disc_number <= NUMDISCS)
    {
        CD_command = "DISC " + itoa(disc_number) + "\r";
        message = "Changing to disc " + itoa(disc_number) + "\n";
    }
    else
    {
        message = "Illegal disc number\n";
    }
}
```

In the example above, the decision made is binary. In some cases, decisions are more complex. For example, to check the current day of the week, execute one set of code if it is Saturday, another set of code if it is Sunday, and yet some other code if it is any other day of the week.

One way to accomplish this is to use a series of `if-else` statements, as shown below:

```
today = GetDayOfWeekNum();          // gets the current day of the week

if (today = 0)                      // is today Sunday?
{
    // code to run on Sundays
}

else if (today = 5)                 // is today Friday?
{
    // code to run on Friday
}

else if (today = 6)                 // is today Saturday?
{
    // code to run on Saturdays
}
else                                // only gets here if the first three
{                                   // conditions are false
    // code to run on all other days
}
```

NOTES:

- There can be as many `if-else` statements in a single construct as necessary. The `switch-case` construct can better manage these, as discussed in the next section.
- `if` statements can be nested inside other `if` statements.

switch-case

When the `if-else` construct is used for making a choice between mutually exclusive conditions, the syntax can be cumbersome. For this particular case, SIMPL+ offers the `switch-case` construct.

The `switch-case` construct is a compact way of writing an `if-else` construct. The basic form of the `switch-case` is as follows:

```
switch (expression)
{
    case (expression1):
    {
        // code here executes if
        // expression = expression1
     }
     case (expression2):
     {
         // code here executes if
         // expression = expression2
     }
     default:
     {
         // code here executes if none
         // of the above cases are true
     }
}
```

NOTE: The use of the default keyword allows specific code to execute if none of the other cases are true. This is identical to the final else statement in the `if-else` construct described in if-else on page 31.

The following example uses `switch-case` to set the value of a variable that holds the number of days in the current month. `getMonthNum`, as shown in the example below, returns an integer from 1–12 corresponding to the current month of year.

```
switch (getMonthNum())
{
    case (2): //February
    {
        if (leapYear) // this variable was set elsewhere
            numdays = 29;
        else
            numdays = 28;
    }
    case (4): // April
        numdays = 30;
    case (6): // June
        numdays = 30;
    case (9): // September
        numdays = 30;
    case (11): // November
        numdays = 30;
    default:   // Any other month
        numdays = 31;
}
```

NOTE: For most SIMPL+ constructs, curly braces are only needed when more than one statement is grouped together. If the program has only a single statement following the case keyword, then the braces are optional.

Controlling Program Flow: Loops

Controlling Program Flow: Branching on page 31 discussed constructs for controlling the flow of a program by making decisions and branching. Sometimes a program should execute the same code a number of times. This is called looping. SIMPL+ provides three looping constructs: the `for` loop, the `while` loop, and the `do-until` loop.

for Loops

The `for` loop is useful to cause a section of code to execute a specific number of times. For example, consider clearing each element of a 15-element string array (set it to an empty string). Use a `for` loop set to run 15 times and clear one element each time through the loop.

Control the number of loops a `for` loop executes through the use of an index variable, which must be an integer variable previously declared in the variable declaration section of the program. Specify the starting and ending values for the index variable, and an optional step value (how much the variable increments by each time through the loop). Inside the loop, the executing code can reference this index.

The syntax of the `for` loop is provided below.

```
for (<variable> = <start> to <end> step <stepValue>)
{
    // code in here executes each time through the loop
}
```

To see an example of the `for` loop, use the situation described above (a need to clear each string element in a string array). A program to accomplish this might look like the following.

```
DIGITAL_INPUT clearArray;                // a trigger signal
INTEGER i;                               // the index variable
STRING stringArray[50][14];              // a 15-element array

PUSH clearArray                          // event function
{
    for (i = 0 to 14)
    {
        stringArray[i] = "";             // set the ith element
                                         // to an empty string
        print("cleared element %d\n",i); // debug message
    }                                    // this ends the for
                                         // loop
}                                        // this ends the push
                                         // function
```

In this example, the loop index `i` is set to run from `0 to 14`, which represents the first and last elements in `stringArray` respectively. Also notice that the step keyword is omitted. This keyword is optional and if it is not used, the loop index increments by 1 each time through the loop. To clear only the even-numbered array elements, the following could have been used.

```
for (i = 0 to 14 step 2)
{ . . . }
```

The step value can also be negative, allowing the loop index to be reduced by some value each time though the loop.

The `for` loop flexibility can be enhanced further by using expressions instead of constant values for the start, end, and step values. For example, there might be a need to add up the value of each byte in a string in order to calculate the value of a `checksum` character. Since the length of the string can change as the program runs, the number of iterations through the loop is unknown.

The following code uses the built-in function, `len`, to determine the length of the string and only run through the for loop the necessary number of times. System functions are described in detail in Using System Functions on page 41.

```
checksum = 0; // initialize the chucksum variable

/* iterate through the string and add up the bytes.
    Note that the { } braces are not needed here
    because the contents of the for-loop is only
    a single line of code */
for (i = 1 to len(someString))
    checksum = checksum + byte(someString,i);

/* now add the checksum byte on to the string
    using the chr function. Note that in this
    example we only use the low-order byte from
    the checksum variable */
someString = someString + chr(checksum);
```

while and do-until Loops

The `for` loop discussed in an earlier section is useful for iterating through code a specific number of times. However, sometimes the exact number of times a loop should repeat is unknown. Instead, it may be necessary to check to see if a condition is true after each loop to decide whether or not the loop should execute again.

There are two looping constructs in SIMPL+ which allows execution of a loop only for as long as a certain condition is true. These are the `while` and `do-until` loops. The `while` loop has the following syntax.

```
while (expression)
{
    <statements>
}
```

When the `while` statement is executed, the expression contained in parentheses is evaluated. If this expression evaluates to `True`, then the statements inside the braces are executed. When the closed brace is reached, the program returns to the `while` statement and reevaluates the expression. At this point the process is repeated. It should become clear that the code inside the braces is executed over and over again as long as the `while` expression remains `True`.

NOTE: For 2-Series control systems, Crestron recommends implementing a `THREADSAFE` keyword to parse Buffer Input data for a `while(1)` loop and to prevent reentry in the event handler. For 3-Series and 4-Series control systems, Crestron recommends implementing a `try/catch` statement inside of a `while(1)` loop to trap any faults that occur in the `try` clause and to extract data. Faults that are trapped in the `try` clause are reported in the `catch` clause. For more information, refer to Crestron online help answer ID 5913.

The nature of the `while` loop means that it is the responsibility of the programmer to ensure that the loop is exited at some point. Unlike the `for` loop discussed previously, this loop does not run for a set number of times and then finishes. Consider the following example.

```
x = 5;
while (x < 10)
{
    y = y + x;
    print("Help me out of this loop!\n");
}
```

NOTE: Endless loops cause the SIMPL+ module (in which they occur) to rerun the same code forever. However, due to the multitasking nature of the operating system, an endless loop in one module does not cause the rest of the SIMPL program (including other SIMPL+ modules) to stop running. This is discussed in more detail in Understanding Processing Order on page 65.

This example shows an endless loop, which is a loop that runs forever and never exits. The problem is that the value of the `while` expression never changes once the loop is entered. Thus the expression can never evaluate to `False`. Include code inside the loop that affects the `while` expression (in this case the variable `x` must be modified in some way) and allows the loop to exit at some point. In this case, the variable `x` must be modified in some way.

The `do-until` looping construct is similar to the `while` loop. The difference lies in where the looping expression is evaluated and how this evaluation affects the loop. To see the difference, examine the form of a `do-until` loop.

```
do
{
    <statements>
} until (expression)
```

From the syntax, it is clear that the looping expression for a `do-until` appears after the looping code. This expression appears before the code in a `while` loop. This discrepancy affects the way the loop is initially entered. As was shown above, a `while` loop first evaluates the expression to see if it is `True`. If it is, then the loop runs through one time and then the expression is evaluated again.

The `do-until` loop differs from this in that it always executes at least one time. When the `do` keyword is reached, the code that follows (enclosed in braces) is executed before the value of the `until` expression is evaluated. After the initial pass through this code, the value of this expression determines whether or not the code should be executed again. Here lies the other difference between the `while` and `do-until`. The `while` loop executes as long as the expression remains `True`. A `do-until` loop executes until an expression becomes `True`.

Use a `for` loop when the number of iterations the code should execute is known. Use a `while` or a `do-until` loop to execute a section of code continuously based on the value of some expression. Use a `do-until` loop if the loop needs to run at least one time. Use a `while` loop if the value of an expression should be checked.

Exit from Loops Early

All three loops discussed above have built-in ways to exit. The `for` loop exits when the index variable reaches the stated maximum. The `while` loop exits when the expression becomes `False`. The `do-until` loop exits when the expression becomes `True`.

It may be desirable (or necessary) to exit a loop prematurely. Consider the following example.

```
INTEGER x,y;

for (x = 3 to z)
{
    y = y + x*3 - z*z;
    if (y = 0)
        break;
}
```

Notice that in most (if not all) cases, the need for the break statement could be avoided by using a different type of loop. In the example above, this could be accomplished by using a `do-until` loop. Consider the following.

```
x = 3;

do
{
    y = y + x*3 - z*z;
    x = x + 1;
} until ((y = 0) || (x = z))
```

Using System Functions

In order to make programming in SIMPL+ simpler and more powerful, the concept of functions is introduced. A function is essentially a more complicated programming task that has been predefined and given a name. Many of the examples in previous sections of this document have used special types of functions called system functions (or built-in functions). To employ system functions, use the following format.

```
returnValue = FunctionName(Parameter1, Parameter2,...);
```

The above syntax is called a function call, because it tells the function to cause/perform an action. Notice that there are three elements to a function call. First, it is identified by `FunctionName`, which must be unique for every system function. The function name itself is followed by a comma-separated parameter list enclosed in parentheses. Each function may have a fixed number of parameters, a variable number of parameters, or no parameters at all. These parameters provide the means to supply information to a function. For example, the itoa function converts an integer value into its ASCII equivalent, which is very useful for creating strings to control switchers. The single parameter to this function would be the integer value that needs to be converted.

Parameters can be any valid SIMPL+ expression. Consider the statements:

```
returnValue = itoa(154);          // constant param
returnValue = itoa(input_num);    // variable param
returnValue = itoa(x*5-4);        // general expression param
```

The variable to the left of the equal sign in the above function calls is used to hold the return value of the function. This value can be either an integer or a string, depending on the nature of the function. Clearly, if a given function returns an integer value, an integer variable must be used to accept this value and a string variable for those functions that return strings. In the itoa examples shown above, the return value is the ASCII string which represents the number being converted. Thus, `returnValue` in this case must be a string variable.

Some functions do not return any values at all. Thus it is not reasonable to assign a variable to a function as shown above. In addition, when using functions that do return values, sometimes the return value may not be needed. In both cases, use the following syntax:

```
FunctionName(Parameter1, Parameter2, ...);
```

In this case, if the function being called does return a value, it is ignored.

Be aware as well that some functions may need to return more than one value. Since functions can only have at most a single return value, there are some functions that modify the values of the parameters that are passed to it.

SIMPL+ provides a large number of system functions, which are listed under a number of categories in the latest revision of the SIMPL+ Language Reference Guide.

User Defined Functions

To simplify programs, it may be necessary to create your own functions. These functions are considered user-defined, and they perform similarly to system functions. However, user-defined functions must be defined before they are used.

User-defined functions have several benefits. To help better organize program modules, creating several small user-defined functions makes programming easier to comprehend and debug. User-defined functions can also be called by any other function. Rather than have the same programming logic written out in several functions, one function can be defined with this logic and then called by any other function within the module, which will also greatly reduce the module's size.

Function Definitions

The programmer must ensure that the SIMPL+ compiler knows about these functions before the program tries to call them. A function definition, not a function call, must be created to tell the SIMPL+ compiler what the function does.

To help functions to be resued, any number of variables can be passed to functions through the function's argument list. This action is called parameter passing, where a variable passes into a function. Other variables or literal values can be passed to function arguments. Function arguments are another way of defining local variables. The difference between declaring a local variable within the function and declaring one as part of the parameter list is that the function argument will have the value of the calling function's variable copied into it.

Functions can also return and compute values. A function might be written to compute a value. Another function might want to perform a task and return an error code that can be evaluated by the calling function. Functions can only return at most one value, namely integers or strings. When defining a function, the returning value will determine what type of function to declare. The different types of functions are: `FUNCTION`, `INTEGER_FUNCTION` and `STRING_FUNCTION`. For the 2-Series and newer compilers, `LONG_FUNCTION`, `SIGNED_INTEGER`, and `SIGNED_LONG_FUNCTION` are also available.

The syntax of a SIMPL+ function call is as follows:

```
FUNCTION MyUserFunction( [parameter1][, parameter2][, parametern] )
{
    <statements>
}

INTEGER_FUNCTION MyUserIntFunction( [parameter1][, parameter2][, parametern] )
{
    <statements>
}

STRING_FUNCTION MyUserStrFunction( [parameter1][, parameter2][, parametern] )
{
    <statements>
}
```

The `FUNCTION` keyword tells the SIMPL+ compiler that what follows is the definition of a function and not a function call. The `FUNCTION` keyword also specifies that there will be no return value for this function. `INTEGER_FUNCTION` and `STRING_FUNCTION` specify that an integer or string value will be returned as the result of the function. These keywords are also referred to as the function type.

The next keyword is a name provided by the SIMPL+ programmer that becomes the function name. Use a unique name and not an existing SIMPL+ keyword, system function, or a previously declared variable name to avoid a compile syntax error.

The function argument list follows the function name. If no arguments are needed within this function, then the list can remain empty. Otherwise, a parameter is defined by giving a variable type and name (for example, `INTEGER myIntArgument`). One or more functions are possible by separating each with a comma.

Function definitions are global to the SIMPL+ module in which they are defined. Therefore, any event function, the `Function Main`, or even another user-defined function can call a user-defined function that has been defined in the same SIMPL+ module. Functions defined in other modules are not accessible.

When calling a function, it is critical that the function is defined in the SIMPL+ module and that it occurs before the line of code that calls the function, as shown below:

```
INTEGER x;

PUSH someSignal
{
    call MyUserFunction1();
    x = MyUserFunction2( x, 10 );
}

FUNCTION MyUserFunction1()
{
    print("This is MyFunction1 runnning!\n");
}

INTEGER_FUNCTION MyUserFunction2( INTEGER arg1, STRING arg2 )
{
    print("This is MyFunction2 runnning!\n");
}
```

This code causes a compile error, because the function `MyUserFunction1` has been called before it has been defined. This issue can be remedied by reversing the order:

```
INTEGER x;

FUNCTION MyUserFunction1()
{
    print("This is MyFunction1 runnning!\n");
}

INTEGER_FUNCTION MyUserFunction2( INTEGER arg1, STRING arg2 )
{
    print("This is MyFunction2 runnning!\n");
}

PUSH someSignal
{
    call MyUserFunction1();
    x = MyUserFunction2( x, 10 );
}
```

This program compiles without any problems. Due to this dependence on order, the SIMPL+ module template that appears each time a new program is created provides a place to put function definitions. Notice that this section comes after the input/output and variable definitions, but before the event and main functions. Following the layout suggested by the template should prevent most of these errors.

Defining Local Variables In Functions

The concept of local variables was introduced in the section All About Variables on page 21. In this section, local variables are discussed in greater detail.

A local variable is a variable (such as an integer or string) with a limited lifespan and limited scope. Global variables, on the other hand, exist for as long as the control system is powered on and the program is running. Global variables retain their values unless a programming statement modifies them. In addition, global variables can be accessed (either to use their value or to modify them) anywhere in a SIMPL+ module.

Refer to the following example:

```
DIGITAL_INPUT go;
INTEGER i,j,k; // define 3 global integers

FUNCTION sillyFunction()
{
    i = j * 2;
    k = i - 1;
}

FUNCTION anotherSillyFunction()
{
    j = i + k;
}

PUSH go
{
    i = 1;
    j = 2;
    k = 3;
    Print("i = %d, j = %d, k = %d\n", i, j, k);
    Call sillyFunction();
    Print("i = %d, j = %d, k = %d\n", i, j, k);
    Call anotherSillyFunction();
    Print("i = %d, j = %d, k = %d\n", i, j, k);
}
```

In this program, both of the functions defined, as well as the push event, have access to the three global integers `i`, `j`, and `k`.

Local variables, on the other hand, can only be accessed within the function in which they are defined. If another user-defined function or event function tries to access a local variable defined elsewhere, a compiler error will be generated. In addition, as soon as the function completes execution, all of the local variables are destroyed, meaning that any value they contained is lost. The next time this function is called, the local variables are recreated.

Creating local variables is identical to creating global variables, except that the declaration statement is placed inside of the function definition as shown below:

```
FUNCTION localExample()
{
    INTEGER i, count, test;
    STRING s[100], buf[50];
}
```

Pass Variables to Functions as Arguments

Global variables are available everywhere in a program, so these variables can be used to share data between functions and the rest of the program. However, this is considered bad programming practice since it often leads to programs that are hard to read and even harder to debug. Functions can also access any input/output signals defined in the program, but as these signals can be thought of as global variables, the same programming issues can arise.

Passing arguments (also known as parameters) into functions should be used instead of global variables to share data. Arguments can be thought of as an ordered list of variables that are passed to a function by the calling function (the term calling function simply refers to the scope of the code statement which calls the function in question). To define a function's parameters, list them inside the parentheses following the function name as shown in the following example:

```
FUNCTION some_function (INTEGER var1, INTEGER var2, STRING var3)

{
    INTEGER localInt;
    STRING localStr[100];

    var1 = var1 + 1;
    localInt = var1 + var2;
    localStr = left(var3, 10);
}
```

The function shown above has three arguments, named `var1`, `var2`, and `var3`. `var1` and `var2` are integers, while `var3` is a string. Refer to the following example of how to call this function from elsewhere in a program:

```
some_function( intVal1, 5+1, stringVal1);
```

This example assumes that the variable `intVal1` has been defined as an integer earlier in the program, as has the string variable, `stringVal1`. The second argument is a constant, and not a variable at all. This means that inside `some_function`, the value of `var2` will be set to `6`.

ByRef, ByVal, and ReadOnlyByRef

When defining a function's argument list, optional keywords can be used to gain greater control over the behavior of the arguments. These keywords are `ByRef`, `ByVal`, and `ReadOnlyByRef`.

These keywords describe the way that SIMPL+ passes variables to the function. When a function argument is defined as `ByRef`, any variable that is passed to this argument will pass enough information about itself to allow the function to modify the value of the original variable. When a function argument is defined as `ByVal`, only the value of the variable and not the variable itself is passed to the function, so the original variable cannot be modified within the function.

The following example presents a function that takes two strings as arguments and inserts one string into the other at a specified character location. Only the first string argument (`string1`) is defined as `ByRef`.

```
FUNCTION insertString(ByRef STRING string1, ByVal STRING string2, ByVal INTEGER position)

{
    STRING leftpart[20], rightpart[20];

    leftpart = left(string1,position);
    rightpart = right(string1,position);

    string1 = leftpart + string2 + rightpart;
}
```

Functions That Return Values

All user-defined functions discussed previously have had one thing in common: when the functions are finished executing, they do not return a value to the calling code. However, some of the functions do modify the values of their arguments, and thus these modified variables can be used by the calling procedure. The term return value is used to describe the core value, which is returned from the function to the calling procedure.

Many system functions discussed earlier in this document have return values. For example, the following statements use the return values of functions:

```
String1 = itoa(int1);
position = find("Artist",CD_data);
int3 = max(int1, int2);
```

Conversely, the following statements use system functions that do not have return values:

```
Print("Variable int1 = %d\n",int1);
ProcessLogic();
CancelWait(VCRWait);
```

For system functions, whether or not it returns a value depends largely upon what that function is designed to do. For example, the `itoa` function would not be valuable if it did not return a string, which could then be assigned to a variable, or used inside of an expression. On the other hand, the `Print` function simply outputs text to the console for viewing, so return value is needed.

Allowing a user-defined function to return a value is useful, as it allows such functions to be used in flexible ways, such as inside of an expression. To allow a function to return a value, use a modified version of the function keyword, as shown below:

```
STRING_FUNCTION strfunc1();  //function returns a string
INTEGER_FUNCTION intfunc1(); //function returns an integer
FUNCTION func1();            //function has no return value
```

Functions defined using the `STRING_FUNCTION` keyword will return a string value, and those defined using the `INTEGER_FUNCTION` keyword will return an integer value. Function declared using the `FUNCTION` keyword have no return value.

Once a function has been declared using the appropriate function type, it is the responsibility of the programmer to ensure that the proper value is returned. This is accomplished using the return function.

Refer to the following function example, which raises one number to the power determined by the second argument, and returns the result.

```
INTEGER_FUNCTION power(INTEGER base, INTEGER exponent)

{
    INTEGER i, result;

    if (base = 0)
        return (0);

    else if (exponent = 0)
        return (1);

    else {
        result = 0; // initialize result
        for (i = 1 to exponent)
            result = result + result * base;

        return (result);
    }
}
```

To use this function in a program, simply call the function like any built-in system function. Refer to the following example:

```
Print("5 raised to the power of 3 = %d\n",power(5,3)); x = power(y,z);
```

The next example builds a function that appends a simple checksum byte onto the end of a string. A checksum is used by many devices to provide a basic form of error checking. In this example, the checksum is formed by adding up the values of each byte in the command string and then appending the equivalent ASCII character of the result onto the string itself. If the checksum value is larger than a single byte (255 decimal), the overflow is ignored and the lower 8-bits of the result are used.

```
DIGITAL_INPUT control_device1, control_device2;
STRING_OUTPUT device1_out, device2_out;
STRING device1_cmd[20], device2_cmd[20], tempCmd[20];

STRING_FUNCTION appendChecksum(STRING command)
{
    INTGEGER checksum, i;   // define local variables

    checksum = 0;    // initialize variable

    for (i = 1 to len(command)) // calculate the sum
        checksum = checksum + byte(command,i);

    return(command + chr(checksum)); //append the byte
}

PUSH vcr_play
{
    vcr_out = appendChecksum("PLAY");
}

PUSH vcr_stop
{
    vcr_out = appendChecksum("STOP");
}
```

In this example, the system function, byte, is used inside the function to get the numeric value of each byte in the string. After the checksum has been calculated, the chr function is used to append the corresponding ASCII character to the end of the command string. This example is useful for just one simple type of checksum.

Function Libraries

Code that has many applications is best placed inside of a function. Unlike system functions, which are globally available, user-defined functions are only available inside of the SIMPL+ module in which they exist. To use a user-defined function in more than one SIMPL+ module, it can be copied and pasted from one program to another. However, this method can lead to problems when a bug is found in one program and then must be fixed in each other program that contains the copid function.

To solve this problem, SIMPL+ uses function libraries. A function library is a collection of user-defined functions that are placed in a separate file. A library can consist of only a single function, or can consist of every function written by a programmer. Crestron recommends organizing libraries so that each one contains related functions. For example, a string handling library may be crested that consists of a number of functions that perform useful operations on stings.

Once a function has been included inside of a function library, it becomes accessible to all SIMPL+ modules that are made aware of it. To make a SIMPL+ module aware of a particular library, use the `#USER_LIBRARY` compiler directive with the following syntax:

```
#USER_LIBRARY "MyStringFunctionLib"
```

The file extension is left out of this syntax. The example above refers to the function library called `MyStringFunctionLib.usl`. Any number of user libraries can be included within a SIMPL+ module.

Special function libraries that are created by Crestron and made available to all customers can be used in a similar manner. The only difference is that the `#CRESTRON_LIBRARY` compiler directive is used in place of `#USER_LIBRARY`. Crestron function library files end with the extension .csl.

Removable Media Functions

2-Series and newer control systems support reading and writing to and from removable media and other storage mediums.

The ability to store and retrieve data from a removable data source can provide many useful and powerful solutions. These solutions include backing up data, transferring data from one control system to another, reading and writing data to and from formats that other database programs can recognize, and implementing database-driven programs (the ability for a program to act dynamically based on actions defined in the database).

The SIMPL+ file functions perform file access using the control system's removable media. 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. For example, a file cannot be opened in one thread, such as `Function Main`, and then accessed with the returned file handle in another thread, such as an event handler. Files should be opened, accessed, and closed within the same thread.

CheckForDisk and WaitForNewDisk

Before accessing removable media, the program must either first check to see if removable media exists within the control system, or wait for media to be inserted.

Certain programs rely on removable media being inserted within the control system. The function in the following example, `CheckForDisk`, tests for removable media within the control system. If media is not present, the function will return an error code and the program can act accordingly.

Other programs might prompt the end-user to insert removable media. The function in the example below, `WaitForNewDisk`, stops the program and allows it to resume only when removable media is detected within the control system.

> **NOTE:** Removable syntax for 2-Series control systems or older is `CFX`, where `X` is any number starting from zero. Removable media syntax for 3-Series control systems or later is `RMX`, where `X` is any number starting from zero.

The following is an example of a program that needs to read data from removable media upon startup:

```
FUNCTION ReadMyRemovableMedia()
{
    // call functions to read the removable media
    //
    // Note that this function will exist within the same
    // thread as the calling function (Function Main)
    // Because of this, the functions, StartFileOperations
    // and EndFileOperations should not be used here.
}

Function Main()
{
    StartFileOperations();
    if (CheckForDisk() = 1)
        Call ReadMyRemovableMedia();
    else if ( WaitForNewDisk() = 0 )
        Call ReadMyRemovableMedia();

    EndFileOperations();
}
```

If the program is dependent upon data from the removable media, it is important that the program validates the existence of the media. Otherwise, the program will not have the necessary data to execute properly. The function above will first check if the removable media is already inserted into the control system upon system startup. If so, it will call the user-defined function, `ReadMyRemovableMedia`, to perform any file read operations on the removable media. If the removable media is not found in the control system, the program will wait for the media to be inserted before continuing. Once inserted, the same function, `ReadMyRemovableMedia`, is called.

Reading and Writing Data

Once the existence of the removable media is verified, the program can read and write data. Data can be read or written either with individual elements (such as a single integer or string) or with entire structures of data.

Because each datatype (such as `INTEGER`, `STRING`, and `LONG_INTEGER`) uses a different amount of storage in memory, there are different functions to read and write each of these types. The return value of each of these functions is the actual number of bytes read or written to the file. This is because data elements are written to a file by inserting one element after another. The file does not contain any information as to what that data is or how it is to be extracted out. It is up to the program that will ultimately read that file to know exactly what is contained within the file and how to extract the data back out of it.

This concept is demonstrated in the following example:

```
DIGITAL_INPUT readRemovableMedia;
DIGITAL_INPUT writeRemovableMedia;

INTEGER myInt;
LONG_INTEGER myLongInt;
STRING myStr[50];

PUSH writeRemovableMedia
{
    SIGNED_INTEGER nFileHandle;
    INTEGER nNumBytes;

    StartFileOperations();

    nFileHandle = FileOpen( "\\CF0\\MyFile", _O_WRONLY | _O_CREAT | _O_BINARY );
    if( nFileHandle >= 0 )
    {
        nNumBytes = WriteInteger( nFileHandle, myInt );
        nNumBytes = WriteLongInteger( nFileHandle, myLongInt );
        nNumBytes = WriteString( nFileHandle, myStr );

        FileClose( nFileHandle );
    }

    EndFileOperations();
}

PUSH readRemovableMedia
{
    SIGNED_INTEGER nFileHandle;
    INTEGER nNumBytes;

    StartFileOperations();

    nFileHandle = FileOpen( "\\CF0\\MyFile", _O_RDONLY | _O_BINARY );
    if( nFileHandle >= 0 )
    {
        nNumBytes = ReadInteger( nFileHandle, myInt );
        nNumBytes = ReadLongInteger( nFileHandle, myLongInt );
        nNumBytes = ReadString( nFileHandle, myStr );

        FileClose( nFileHandle );
    }

    EndFileOperations();
}
```

The functions `ReadStructure` and `WriteStructure` automate reading and writing the individual fields within the structure. These functions do not return the number of bytes read or written. Instead, both functions have an additional argument that will contain the number of bytes read or written after the function call executes.

This concept is demonstrated in the following example:

```
DIGITAL_INPUT readRemovableMedia;
DIGITAL_INPUT writeRemovableMedia;

STRUCTURE myStruct
{
    INTEGER myInt;
    LONG_INTEGER myLongInt;
    STRING myStr[50];
}
myStruct struct;

PUSH writeRemovableMedia
{
    SIGNED_INTEGER nFileHandle;
    INTEGER nNumBytes;

    StartFileOperations();

    nFileHandle = FileOpen( "\\CF0\\MyFile", _O_WRONLY | _O_CREAT | _O_BINARY );
    if( nFileHandle >= 0 )
    {
        WriteStructure( nFileHandle, struct, nNumBytes );

        Print( "The number of bytes written = %d", nNumBytes );

        FileClose( nFileHandle );
    }

    EndFileOperations();
}

PUSH readRemovableMedia
{
    SIGNED_INTEGER nFileHandle;
    INTEGER nNumBytes;

    StartFileOperations();

    nFileHandle = FileOpen( "\\CF0\\MyFile", _O_RDONLY | _O_BINARY );
    if( nFileHandle >= 0 )
    {
        ReadStructure( nFileHandle, myInt, nNumBytes );

        Print( "The number of bytes read = %d", nNumBytes );

        FileClose( nFileHandle );
    }

    EndFileOperations();
}
```

Working with Time

Sometimes, programs must have control over exactly when their statements execute. This section describes the language constructs that concern time within SIMPL+ programming.

Delay

The `Delay` function pauses the execution of the current SIMPL+ module for the time specified in the parameter field. As with most time values in SIMPL+, this time value is specified in hundredths of seconds. The following code causes the program to stop for five seconds before resuming.

```
PUSH startMe
{
    Print("I'm starting now...");
    Delay(500);     //this equals 5 seconds
    Print("and I'm ending 5 seconds later.\n");
}
```

The control system never allows a SIMPL+ module to lock up the rest of the system for any significant amount of time. Therefore, the control system performs a task switch whenever a delay function is reached, meaning that the execution of the current SIMPL+ module stops momentarily as the natural flow through the SIMPL program continues. In this case, even though the SIMPL+ module has stopped for five seconds, other code in the overall SIMPL program (including standard logic symbols and other SIMPL+ modules) continues to operate normally. The concept of task switching is covered in more detail within Understanding Processing Order on page 65.

Pulse

The `Pulse` function is used to drive a digital output signal high for a specified amount of time (also in hundredths of seconds). When a `Pulse` statement is executed, the digital output signal specified is driven high and a task switch occurs. This task switch is necessary in order for the rest of the SIMPL program to recognize that the digital signal has gone high. After the time specified has expired, the digital output is driven low and another task switch occurs.

The following program causes the digital output signal, `preset_1`, to be pulsed for a half a second.

```
#DEFINE_CONSTANT PULSE_TIME 50

DIGITAL_OUTPUT preset_1, preset_2, preset_3;

PUSH some_input
{
    Pulse(PULSE_TIME, preset_1);
}
```

> **NOTE:** The `Pulse` function is very similar in operation to the SIMPL **One Shot** symbol. In many cases, it may be more convenient to simply connect one or more **One Shot** symbols to the output signals of a SIMPL+ module.

Unlike the `Delay` function, `Pulse` does not cause a pause in the execution of the SIMPL+ code. Therefore, the statements that follow the `Pulse` execute immediately and do not wait for the expiration of the pulse time.

Wait Events

`Wait` events in SIMPL+ allow operations that are similar to the **Delay** SIMPL logic symbol. The syntax for a `Wait` event is as follows:

```
Wait (wait_time [, wait_name])
{
   <statements>
}
```

This syntax defines a `Wait` event to occur at some time in the future, defined by the value of `wait_time`. While the `Wait` event is pending execution, it is considered scheduled. The `Wait` event may have an optional name, which can be used to refer back to the event elsewhere in the code.

When a `Wait` event definition is reached during execution, the execution of the statements inside the braces is deferred until the time defined by `wait_time` has expired. These braces are not needed if the event is only one statement long. The remainder of the SIMPL+ module executes until the `Wait` event finishes. If a `Wait` event definition is nested inside of a loop, it may be reached multiple times before it executes once. If a `Wait` event is pending (it has been scheduled but not executed), it is not scheduled again until it has been completed.

Once a `Wait` event has been scheduled to execute at a later time, various operations can be performed on the event before it actually executes. However, only named `Wait` events can be modified in this manner, since it is necessary to use the name to refer to the event. The following table lists the available functions, which can operate on `Wait` events.

Functions Available During Wait Events

Function	Description
`CancelWait(name)`	Removes the named wait from the schedule. The code never executes.
`CancelAllWait()`	Removes all pending waits from the schedule.
`PauseWait(name)`	Stops the timer for the named wait. The code does not execute until the timer is started again using `ResumeWait()`.
`ResumeWait(name)`	Resumes the timer for the named wait, which had been paused earlier.
`PauseAllWait()`	Similar to `PauseWait()`, but acts on all pending wait events.
`ResumeAllWait()`	Similar to `ResumeWait()`, but acts on all paused wait events.
`RetimeWait(time, name)`	Sets the time for a pending wait event to the value specified.

The following example shows a typical use of `Wait` events, where the `SYSTEM ON` button starts a power up sequence and the `SYSTEM OFF` button starts a power down sequence.

```
#DEFINE_CONSTANT PULSETIME 50 // half second

DIGITAL_INPUT system_on, system_off;
DIGITAL_OUTPUT screen_up, screen_down, lift_up, lift_down;
DIGITAL_OUTPUT vcr_on, vcr_off, dvd_on, dvd_off;
DIGITAL_OUTPUT vproj_on, vproj_off;
DIGITAL_OUTPUT vproj_video1, vproj_video2, vproj_rgb;
DIGITAL_OUTPUT lights_pre_1, lights_pre_2, lights_pre_3;

PUSH system_on
{
    CancelWait(sysOffWait); // cancel the system off wait event

    Pulse(2000, screen_down); // lower screen for 20 sec.
    Pulse(9500, lift_down); // lower lift for 9.5 sec.

    Wait (1000, sysOnWait1) // 10 second delay
    {
        Pulse(PULSETIME, vcr_on);
        Pulse(PULSETIME, dvd_on);
        Pulse(PULSETIME, lights_pre_1);
        Pulse(PULSETIME, vproj_on);
    }

    Wait (1500, sysOnWait2) // 15 second delay
    {
        pulse(PULSETIME, vproj_video);
    }
} // end of push event

PUSH system_off
{
    CancelWait(sysOnWait1);
    CancelWait(sysOnWait2);

    Pulse(2000, screen_up);
    Pulse(PULSETIME, vproj_off);
    Pulse(PULSETIME, vcr_off);
    Pulse(PULSETIME, dvd_off);

    Wait(500, sysOffWait)
    {
        Pulse(9500, lift_up);
        Pulse(PULSETIME, lights_pre_3);
    }
} // end of push event
```

In this example, the `CancelWait` function is used to cancel any pending waits when the `SYSTEM ON` or `SYSTEM OFF` buttons were pressed. This operation is analogous to using the reset input on the **Delay** symbol in SIMPL.

Working with Strings

This section provides information for working with incoming serial data.

BUFFER_INPUT

> **NOTE:** For more information on BUFFER_INPUT, refer to Working with Data (Variables) on page 18.

Sserial data entering a SIMPL+ module may be treated as either a STRING_INPUT or as a BUFFER_INPUT.

The value of a STRING_INPUT is always the last value of the serial signal that feeds it from the SIMPL program. Every time new data is generated on the serial signal in the SIMPL program, the STRING_INPUT variable in the SIMPL+ module changes to contain that data. Any data that was previously contained in that variable is lost.

A BUFFER_INPUT, conversely, does not lose any data that was stored there previously. Instead, any new data that is generated onto the serial signal in the SIMPL program is appended to the data currently in the BUFFER_INPUT variable.

The SIMPL program shown below contains two **Serial Send** symbols, with each triggered by a button press. The outputs of these symbols are tied together so that both symbols can generate a string onto the same serial signal. This signal is then connected in two places to the SIMPL+ module. The first input is mapped to a STRING_INPUT, and the second is mapped to a BUFFER_INPUT. The declaration section for this module should appear as follows.

```
STRING_INPUT theString[100];
BUFFER_INPUT theBuffer[100];
```

The **Serial Send** symbol generates the static text defined in its parameter field onto the output serial signal whenever the trigger input sees a rising signal.

The following table shows the state of these two input variables in response to button presses.

States of Two Input Variables

Action	theString	theBuffer
system initializes	empty	empty
button 1 pressed	"Now is"	"Now is"
button 2 pressed	"the time"	"Now is the time"
button 1 pressed	"Now is"	"Now is the time Now is"

Each time the serial signal changes, `theString` assumes this value and the old data stored there is lost. On the other hand, `theBuffer` retains any old data and simply appends the new data onto the end.

Each application should dictate whether it is appropriate to use a `STRING_INPUT` or a `BUFFER_INPUT`. In general, use a `STRING_INPUT` when the serial signal that is feeding it is being driven from a logic symbol like a **Serial Send**, **Analog to Serial**, or **Serial Gather**. In these cases, the serial data is issued on a single logic wave, so the entire string is copied into the `STRING_INPUT`.

> **NOTE:** A logic wave is the time needed for a signal to propagate from the input to the output of a single logic symbol. This concept is discussed fully in Understanding Processing Order on page 65.

If, however, the signal feeding into the SIMPL+ module comes from a streaming source such as a serial port, use a `BUFFER_INPUT`, which can gather up the data as it dribbles in.

The following example program is written for a CD jukebox, which is capable of sending text strings containing the current song information. Typical data received from this device might appear as follows, where the `<CR>` at the end of the string represents a carriage return character:

```
Artist = Frank Sinatra, Track = My Way, Album = Very Good Years<CR>
```

It is likely that the operating system would not remove the entire string from the serial port in one piece due to its length. The control system checks the state of the serial ports very often and removes any data that is found there. Since this data takes some time to reach the port (depending on the baud rate), it is likely that the port's input buffer is collected before the whole string arrives. If there was a serial signal called `jukebox_in` connected to the RX terminal on the COM port definition, the program could be written as follows.

```
first pass:
    jukebox_in = "Artist=Frank Sinatra, Trac"
second pass:
    jukebox_in = "k=My Way, Album=Very Good Yea"
third pass:
    jukebox_in = "rs<CR>"
```

If the `jukebox_in` signal was then connected to a `STRING_INPUT` of a SIMPL+ module, it is likely that the string might not be seen as one complete piece. Therefore, the artist's name, the track name, and the album name might not be parsed out for display on a touch screen. On the other hand, if a `BUFFER_INPUT` signal was used instead, this buffer would collect the data as it arrived. After the processor read the port the third time, this `BUFFER_INPUT` would contain the complete string.

Remove Data From Buffers

Once data has been routed into a BUFFER_INPUT, techniques are required to extract data from it. The first thing to do with data on a BUFFER_INPUT is to pull off a completed command and store it into a separate variable. For example, most data that comes from other devices are delimited with a certain character (or characters) to denote the end of the command. In many instances, a carriage return (or carriage return followed by a line feed) is used.

The getc function is the most basic way to remove data from a buffer. Each getc call pulls one character out of the buffer and returns that character's ASCII value as the function's return value. Characters are removed from the buffer in the order they arrived, so the first character in becomes the first character out. This function now provides the ability to extract data until the desired delimiter is seen. For example, the following code reads data from the buffer until encountering a carriage return.

```
BUFFER_INPUT data_in[100];
INTEGER nextChar;
STRING temp[50], line[50];

CHANGE data_in // trigger whenever a character comes in
{
    do
    {
        nextChar = getc(data_in); // get the next character
        temp = temp + chr(nextChar);
        if (nextChar = 0x0D) // is it a carriage return?
        {
            line = temp;
            temp = "";
        }
    } until (len(data_in) = 0) // empty the buffer
}

Function Main()
{
    temp = "";
}
```

A do-until loop was used in the example above. Every time a change event is triggered for the data_in buffer, it is uncertain that only one character has been inserted. Many characters may have been added since the last change event. Because of this possibility, continue to pull characters out of the buffer with getc until the buffer is empty, which is what the expression (len(data_in) = 0) reveals.

The extracted character is also stored into an integer because getc returns the ASCII value of the character, which is an integer. On the next line, the chr function is used to convert that value into a one-byte string, which can be added to temp.

Although this example should work for real-world applications, a potential issues can occur if multiple lines of data come in on the same logic wave. If this issue occurs, only the last complete line is stored into `line` and the rest is lost. There are a few potential solutions to this problem:

- Make `line` into a string array and store each subsequent line into a different array element.
- Any code that is needed to further act upon the data could be built directly into this loop. Thus removing the need to store more than one line of data.
- Use `THREADSAFE` as described in Crestron online help answer ID 5109.

Once the data has been removed from the buffer and stored in a known format (in this case, one complete command from the device), the desired data can be extracted. Using the previous example where the data was coming from a CD jukebox, the following example could be used to extract the artist, track, and album title.

```
BUFFER_INPUT jukebox[100];
STRING_OUTPUT artist, track, album;
INTEGER startPos;
STRING searchStr[20], tempStr[100];

CHANGE jukebox
{
    do
    {
        tempStr = tempStr + chr(getc(jukebox));
        if ( right(tempStr,1) = "\r" )
        {
            searchStr = "Artist=";
            startPos = Find(searchStr,tempStr);
            if (startPos)
            { // was the string found?
                startPos = startPos + len(searchStr);
                artist = mid(tempStr, startpos,
                    Find(",",tempStr,startpos) - startpos);
                searchStr = "Track=";
                startpos = Find(searchStr,tempStr) + len(searchStr);
                track = mid(tempStr, startpos,
                    Find("\r",tempStr,startpos) - startpos);
                searchStr = "Album=";
                startpos = Find(searchStr,tempStr) + len(searchStr);
                album = mid(tempStr, startpos,
                    Find("\r",tempStr,startpos) - startpos);
                tempStr = "";
            }
        }
    } until (len(jukebox) = 0);
}

Function Main()
{
    tempStr = "";
}
```

The previous example introduces two new system functions, `Find` and `Mid`, which are useful for string manipulation. To search for the existence of a substring inside of another string, use `Find`. If it is located, the return value of the function is the character position where this string was found. If the substring was not found, then `Find` returns zero. Notice that near the top of the example, the program checked to see if the substring `Artist =` is found in the string. If it is not, then it is assumed that the incoming data was of another format and there is no reason to look for other search strings (`Track =` and `Album =`).

Understanding Processing Order

This section describes processing order within SIMPL+.

How SIMPL+ and SIMPL Interact

Advanced SIMPL programmers should be familiar with how logic is processed in SIMPL programs. This document does not explain this concept here, but it does detail how SIMPL+ modules fit into the picture. However, it is important to understand the following definitions.

- **Logic Wave**: The act of propagating signals from the input to the output of a logic symbol. In a typical program, a single logic wave may include the processing of many symbols.
- **Logic Solution**: An arbitrary number of logic waves, processed until the state of all signals in the program have settled to a stable (unchanging) state.

When a SIMPL+ event function is triggered, it is processed to conclusion in a single logic wave. If this event caused a change to one of the output signals, that signal would be valid one logic wave after the event was triggered. In this simple case, a SIMPL+ module acts identically to a SIMPL logic symbol from a timing standpoint. In addition, for multiple SIMPL+ events triggered on the same logic wave (whether or not they are in the same module), these events multitask (run at the same time) and complete before the next wave.

As SIMPL+ modules become more complex and processor intensive, however, this general rule may no longer apply. Instead, the operating system may determine that too much time has elapsed and temporarily suspend the SIMPL+ module while it continues to process the SIMPL logic (which may also include other SIMPL+ modules). For example, if an event function must run through a loop 2,000 times before completing, the processor may decide to perform a task switch and process other logic before completing the loop.

This task-switching ability has the benefit of not freezing up the rest of the program while a particularly intensive calculation is proceeding. After the completion of the current logic solution, the SIMPL+ module that was exited continues from where it left off.

Force a Task Switch

There may be times in programming when it is necessary to force a task switch to occur. For example, when a `digital_output` is set high, it normally is not propagated to the SIMPL program until the SIMPL+ completes. To guarantee that the digital signal is asserted, force the system to switch from the SIMPL+ module back into the SIMPL program.

There are two ways to force a task switch: with the `ProcessLogic` function or the `Delay` function. To provide an immediate task switch out of the current SIMPL+ module, use `ProcessLogic`. When the logic processor enters this module on the next logic solution, execution begins with the line immediately following. An immediate task switch also results from `Delay`, but the SIMPL+ module does not continue executing until the time specified has elapsed. The `Delay` function is discussed in greater detail in Working with Time on page 56.

Debugging

Programming bugs can be mistakes in syntax, typos, design errors, or a misunderstanding of certain language elements. This section is not intended to prevent mistakes, but rather to find and fix them through debugging.

Compiler Errors

Some of the easiest errors to remedy involve the compiler, because the compiler reveals what the problem is and where in the program it is located. A programmer must recognize what the error is based on the instructions from the compiler.

The following list provides the most common causes of compiler errors.

- Missing a semicolon at the end of a statement
- Having a semicolon where it does not belong (such as before an opening brace of a compound statement)
- Trying to use a variable that has not been declared
- Misspelling a variable
- Attempting to assign a value to an input variable (digital, analog, string, or buffer)
- Syntax errors

If multiple error messages are received when compiling the program, always work with the first one before attempting to fix the rest. Many times, a missing semicolon at the beginning of a program can confuse the compiler enough that it thinks there are more errors. Fixing the first error may clear up the rest.

Run-time Errors

A run-time error refers to an error that is not caught by the compiler but causes the program to crash while it is running. For example, consider the following statement.

```
x = y / z;
```

The compiler passes by this statement in the program with no problem as expected. However, if during run-time the variable `z` contains `0` when this statement executes, this becomes an illegal operation. Although the control system is robust enough to catch most errors like this and the system should not crash, unexpected results may occur.

To determine if a run-time error is occurring in your program, watch the status of the control system's computer port with a program such as with Crestron Toolbox™ software. An `Interrupt` message or some other error message can clue the user in to a problem. Locate the problem by strategically placing `Print` statements in the code. The `Print` statement is covered in the next section.

Debugging with Print()

> **NOTE:** `Print()` and `Trace()` functions are interchangeable when debugging.

The most powerful debugging tool for use with SIMPL+ is the `Print` function, which allows the user to print out messages and variable contents during program execution. The data that is generated by the `Print` function is sent to the computer port of the control system, making it viewable using a terminal emulation program such as the SIMPL Debugger tool in Crestron Toolbox software.

The `Print` function is nearly identical to the `MakeString` function discussed in Working with Data (Variables) on page 18. The only difference between the two functions is that `MakeString` generates a formatted string into a string variable, while `Print` generates a formatted string and spits it out of the control system's computer port.

The syntax of `Print` is provided in the following example.

```
Print("<specification string>",<variable list>);
```

The `<specification string>` is a string, which determines what the output looks like. It can contain static text intermixed with format specifiers. A format specifier is a percentage sign (`%`) followed by a type character. For example, `%d` is a format specifier for a decimal value, and `%s` is the format specifier for a string. Refer to the following example:

```
INTEGER extension;
STRING name[30];

PUSH print_me
{
    extension = 275;
    name = "Joe Cool";
    Print("%s is at extension %d", name, extension);
}
```

When this program is run and the digital input `print_me` goes high, the following text is output from the computer port:

```
Joe Cool is at extension 275
```

The `Print` statement works by replacing the format specifiers in the specification string with the value of the variables in the variable list. The order of the format specifiers must match the order of the variables in the list. In this example, the first format specifier encountered is `%s`, which corresponds to the string name. The next specifier is `%d`, which corresponds to the integer extension. If the variables in the list were reversed and the specification string kept the same, the output would be unpredictable because the system would try to print extension as a string and name as an integer.

Refer to the SIMPL+ Language Reference Guide for a listing of all available format specifiers for use with the `Print` and `MakeString`.

This page is intentionally left blank.

Crestron Electronics, Inc.
15 Volvo Drive, Rockleigh, NJ 07647
Tel: 888.CRESTRON
Fax: 201.767.7656
www.crestron.com

Programming Guide — Doc. 5789D
03/07/23
Specifications subject to
change without notice.