Edge Integration

Station Controller (sc)

User's Guide

May 1, 2019
Version 2.6

This page left blank intentionally

Table of Contents

Forward

Revision Notes

Versions

Version 2.6 (reflected in version 2.5.0)

- Worked through code and cleaned it up so that it would compile with the standard set to gnu18.
- Fixed various minor bugs.

Version 2.5 (reflected in version 2.4.0)

- Added a new function, file_orc(), for reading and returning a specified line within a file.
- For the "open port", more options have been added parity, data bits and stop bits
- Allow variables on the left of the ='s to contain a "-". Ex x="abc-123'; let y[$x] = ….

Version 2.4.7 (reflected in sc version 2.3.18)

- Added MORE_SECS_TMP_VARIABLES pragma, and new syntax for sending secs messages
- (see SECS Pass-Through)
- Added documentation for checksum()
- Added ALGEBRAIC_PRECEDENCE pragma
- Added new function, set_system_options(). Used by the system() and system_shell().
- Added documentation for ftoh() and htof()

Version 2.4.6 (reflected in sc version 2.3.17)

- Added new function is_number()
- Open statement allows for options=<> to be an expression.

V**ersion 2.4.5 (reflected in sc version 2.3.16)**

- Added clarification for evaluating expressions.
- Modified the open command to allow a client connection to be closed by a server
- New option for the hsms protocol to handle client disconnect.

Version 2.4.4 (reflected in sc version 2.3.14)

- New random number function rand() / rand(seed)
- Mod so that you can use strings and constants for labels.

Version 2.4.3 (reflected in sc version 2.3.13)

- New function, is_timer() returns seconds remaining for a timer
- New function, save_tmp_var(), makes tmp variables permanent
- New feature to allow linking 'C' functions with the sc executable

Version 2.4.2

- Added the Boolean Not operator. Uses the "!" character.
- New function, htou(), to convert hex values to unsigned ints.
- Fixed the htoi() function to properly return negative values.

Version 2.4.1

- Added some clarification with "Dynamic SECS Body" section. The "." should not be included in the variable data.

Version 2.4.0

- Added clarifications and various edits submitted by Douglas Kaip... thanks Douglas

Version 2.3.9

- New options to standard protocol, tmp_var=toupper or tmp_var=tolower. Converts tmp variable names to upper or lower case
- New function is_print(), returns true or false if a char is printable
- New function atoi(), returns the numeric value of a char

Version 2.3.8

- New function, mkprint()

Version 2.3.7

- No changes to this document

Version 2.3.6

- Added the crlf option to the commands protocol.

Introduction

The purpose of this document is to show how to create program files using the Edge Integration sc Station Controller. This document explains how the sc functions and the program language syntax along with programming examples.

Getting Started

In this section, we discuss how to install the sc, start the sc, stop the sc, and the sc command line options.

To install the sc, copy the sc executable (sc) to the appropriate directory on your system. Make sure that your $PATH environment variable is properly set.

To start the sc, at the system prompt type sc. You should see a greeting. Hist the *enter* key to get an sc prompt and a list of commands.

```
$ sc –i
```

To terminate the sc, at the sc prompt, type *kill*.

```
$ sc> kill
```

To get a list of the sc options, type *sc -i* at the system prompt.

```
$ sc –h
```

Always put a blank space between the *-option* and the argument.

```
$ sc –d 3
```
not `$ sc –d3`

A typical way of starting sc when being used as an Equipment Interface simulator during Equipment Interface development is:

```
$ sc –i –c "read my_program"
```

Or if the preprocessor directives are used in "my_program" FIXME at the current time my_program must be in the default directory.

```
$ sc –i –c "read –P my_program"
```

The following is a list of the sc command line options:

sc option	Description
`–c` command	Used to specify or execute an sc command when starting the sc. The specified command is any valid sc command.

sc option	Description
	`$ sc -c "read my_program_file"` `$ sc -c "read \"./src/my_program_file\""`
`-d` level	Used to specify the debug or trace level. This is used to debug the sc application and your program file. The level ranges from 0 to 5. The default is 0. which turns off debugging. `$ sc -d 3 -c "read my_program_file"`
`-g`	Used to show the GNU GENERAL PUBLIC LICENSE that this program is licensed under.
`-h`	Used to show the options. `$ sc -h`
`-i`	Runs the sc in interactive mode. The sc prompts for commands. `$ sc -i` Note: While in the interactive mode, if you press a key sc will stop processing input on any open connections until the enter key is pressed. This is usually not an issue unless you are entering a command with a lot of characters. If this is the case consider using cut and paste from a different window.
`-n` name	Assigns the name to the sc. The name is used in log messages. This is useful when running multiple sc's and error messages are sent to a common error file.
`-r`	Displays sc command reference information.
`-s` service	Used to specify an interface file socket, service when starting the sc. This socket is used to communicate with the running sc. This utility sc_talk is used to communicate with the running sc through this socket. (A good convention is to put the socket files in a sockets directory and prefix them with an "`s.`") `$ sc -c "read my_program_file" -s sockets/s.sc_tool`

sc option	Description
`-v`	Used to get the current version of the sc. `$ sc -v`
`-w`	Displays program warranty information. In short, there is **no** warranty of any kind.
`-x my_program_file`	Used to run the program syntax checker. `$ sc -x my_program_file`
`-I` path	Define an include path to search for include files. See (`#include`).
`-D` name	Define a name in a program. See (`#ifdef`). `$ sc -c "read -D DEBUG my_program_file"`
`-D` name = value	Substitute all occurrences of name with value in a program. `$ sc -c "read -D DEBUG -D TOOL=Nova my_program_file"`
`-P`	Enable the preprocessor. By default it is disabled. It is automatically enabled if the -I or -D options are used. `$ sc -c "read -P my_program_file"`

Table 1: Startup Command Line Options

sc Command Line Commands

In this section we discuss the sc commands.

The following sc commands can be entered at the sc prompt:

sc Command	Description
`help`	Used to get a list of all of the commands. `sc> help`
`debug` *`level`*	Used to set the debugging level. The level ranges from 0 (turns debugging off) to 5.

sc Command	Description
	`sc> debug 3`
`dump`	Used to dump variables and other internal information. All variables are dumped. The results are written to a file named "`dump.out`". `sc> dump`
`dump` *`variable`*	Used to dump a variable and other internal information. The specified variable is dumped. The results are written to a file named "`dump.out`". `sc> dump my_program_variable`

`event` *`label`*	Used to cause program execution to "jump" to the specified label in the running program.
	
	`sc> event start_1`
	
	Or in a shorter form:
	
	`sc> e start_1`
	
	In this example the running program will start executing at the label `start_1` defined in the running program.

`event` *`label arguments`*	Used to cause program execution to "jump" to the specified label in the running program and allow the specified arguments to be accessed by the running program.
	
	`sc> event start_1 recipe=bake_me`
	
	Or in a shorter form:
	
	`sc> e start_1 recipe=bake_me`
	
	In this example the running program will start executing at the label `start_1` defined in the running program. The value "`bake_me`" will be assigned to a temporary variable (see the section on temporary variables) with the name of `recipe`.

sc Command	Description
`kill`	Use to terminate the program. `sc> kill`
`list`	Used to dump a listing of the loaded program. The results are written to a file named "`program.list`". `sc> list`
`logging`	Used to view the current level of logging for the running program. A level of 0 (zero) indicates that logging is disabled. `sc> logging`
`logging name value`	Used to change the current level of logging for the "channel" identified with the value of the `name` parameter. `sc> logging host_connection 3` In this example the value of the `name` parameter is "host_connection". If you look in the Examples section you will see an open (see the section on the open statement) statement that has as part of it "name=host_connection". The communication logging for this connection/channel will be modified.
`ping`	Used to determine whether the running program is responsive. `sc> ping`
`read` *`my_program_file`* `load` *`my_program_file`*	Used to read, load, and start executing the specified program file. The `read` and `load` commands are synonymous. `sc> read my_program_file`
`set` *`name value`*	Used to create or assign the global program variable `name` and give it the value specified by `value`. sc> set reset_flag 1 This is equivalent to the statement "`glet reset_flag=1`" in the program itself.

sc Command	Description
`tokens`	Used to dump the program file in internal token form. The results are written to a file named "`program.tokens`". `sc> tokens`
`version`	Used to get the version of the sc that is currently running. `sc> version`

Table 2: sc Command Line Prompt Commands

sc Program

Hello World

In this section we will show how to create and run a simple program. This program will print the message "Hello World".

Use any text editor and create a file called "`hello`", with the following program statement.

```
print "Hello World"
```

To run the new program, enter the following command:

```
$ sc -c "read hello"
```

Prints the message: `yyyy/mm/dd hh:mm:ss PRINT Hello World`

Program Structure

This section discusses how to layout a program.

A program contains statements.

For example:

```
let a = 1
let a = $a + 1
```

etc.

A program may also contain subprograms. Subprograms are defined with a *begin* and *end* statement. The *begin* may contain a name. A subprogram without a name is called the default subprogram. There can only be one default program.

statements

```
begin # Default subprogram
```

  statements

```
end
begin remote # Remote subprogram
```

  statements

```
end
begin local # Local subprogram
```

  statements

```
end
```

statements

Only one subprogram is active at any time. The set_program statement is used to define the active subprogram. Only those statements within the active program can be executed. All other subprograms are temporarily ignored. However, the statements not part of any subprogram are also active. In the example below, the **bolded** statements are active when the program is first loaded.

statements...

```
set_program local


begin
```

  statements

```
end
```

`begin local`

  statements

`end`

statements

Program Execution

This section discusses how a program's statements are executed.

When a program is first read, execution starts at the beginning. i.e. The first line of the file. Execution continues until any of the following statements are encountered: `begin`, `end`, `label`, `after`, or `break`. This rule holds true whenever the program is executing statements. i.e. If a message is received and being processed the message handler will run until one of the afore mentioned statements has been reached.

When the following program is first read, the only the **bolded** lines are executed. No subprogram is active.

statements...

`# set_program local` Note: This line is commented out.

`begin`

statements

`end`

However, if we remove the comment to end of line character(`#`) the `set_program` statement, execution looks as follows and the active subprogram is local.

statements…

`set_program local`

`begin`

statements

`end`

`begin local`

statements...

`label start`

statements

`end`

The `set_program` statement instructs the sc to:

1. Make the subprogram "`local`" the active program.

2. Start executing the statements after the `begin local` statement. Execution stops when the `label start` statement is encountered.

In order to appreciate this, consider the following example:

You have created an sc program that runs through a state machine. You have the processing for each state located just after a label for that state. Now consider that the state machine still transitions through the same states when the mode is in "`remote`" vs. "`local`", however, the processing performed is different depending on the current value of `mode`. The following code snippet shows an example of what some of the code might look like without the set_program statement.

```
goto newState_$mode
label state_1_local
statements
label state_2_local
statements
label state_n_local
statements
label state_1_remote
statements
label state_2_remote
statements
    .
    .
    .
label state_n_remote
statements
```

Now if the `set_program` statement is used the code might look more like.

```
set_program $mode
begin local
label state_1
```

```
    statements
    label state_2
    statements
        .
        .
        .
    label state_n
    statements
    end
    begin remote
    label state_1
    statements
    label state_2
    statements
        .
        .
        .
    label state_n
    statements
    end
```

This method has the potential for making the code easier to understand.

Handling Event and Reply Messages

This section discusses how events are handled.

Unsolicited events and reply messages are handled by the sc in the same way. Here is how it works:

The sc listens on each opened connection. When a message arrives, the sc knows the protocol to use via the proto syntax of the open statement. The sc creates a message from the event and jumps to a `label` statement with the same name as define by the name parameter in the open statement.

For example:

```
open socket_server name=host_1 proto=hsms ...
open socket_server name=host_2 proto=hsms …

# All communications from the host_1 connection start here
label host_1
let conn = "Host 1"
let sxfy= get_tmp_var(SXFY)
goto $sxfy

# All communications from the host_2 connection start here
label host_2
let conn = "Host 2"
let sxfy= get_tmp_var(SXFY)
goto $sxfy
```

The sc creates temporary variables from the event message that are available to the program (using the `get_tmp_var ()` function). Temporary variable names are created by two rules: name/value pairs and ARGx.

Note: The following only applies when the value of the `proto` option is something other that `hsms` or `secs`.

Name/value pairs are created when the sc sees two items delimited by the "=" sign.

Given an event message with: **color=red**

sc creates the temporary variable `color`.

```
get_tmp_var (color)       # Returns red
```

When the sc sees delimited values, it creates temporary variables starting with `ARG`x, where x starts at 0 and is incremented for each delimited value. A final variable, `NUM_ARG`, is created defining the total number of `ARG` variables.

Given an event message with: **red white blue**

sc creates temporary variables: `ARG0`, `ARG1`, `ARG2`, `NUM_ARG`, and `ARGS`

```
get_tmp_var (ARG0)            # Returns red
get_tmp_var (ARG1)            # Returns white
get_tmp_var (ARG2)            # Returns blue
get_tmp_var (NUM_ARG)         # Returns 3
```

Also, a variable `ARGS` is assigned to the entire message.

```
get_tmp_var (ARGS)            # Returns red white blue
```

and remember

```
get_tmp_var ("")              # Returns all temporary variables
```

Note: This is usually used in a print statement for debugging purposes.

i.e. `print get_tmp_var("")`

Both types of variables may be created from the same message.

Parsing Rules

Messages are parsed into token values using the sc rules for parsing. When a message is tokenized, it is broken down into individual tokens. Tokens are defined as follows:

A *number token* starts with a digit, 0-9, and consists of the digits 0-9, and optionally may contain: "e", "e+", "E-", and "E+", and trailing digits 0-9. The token may not contain any blank spaces.

```
100
1.0e+2
```

A name token consists of alphanumeric characters (A-Z, a-z, 0-9), and the "$", "_", ".", "{", "}", "[", and "]" characters. The token may start with any of the above except the 0-9 digits.

```
num_of_points
point.1.value
value[1]
```

A name token may contain a "-" when referenced within a variable.

```
let x = "abc-123"

let y[$x] = ....
```

Other tokens include:

Token	Description
=	Equal sign
<= >= == !=	Equality signs
()	Open and close parenthesis
+ - * /	Plus, minus, multiplication, and division signs
& |	Bitwise ANDing and ORing signs

All other characters are considered to be delimiters. They are used to delimit tokens, but are discarded and not returned as a token or as part of a token.

Any double or single quoted string becomes a single token.

Note: Within double quotes, you can have `\n`, `\r` and `\xhh` and they will be translated to the appropriate characters. Within single quotes, data is what it is, no conversion.

Program Variables

This section discusses how to define, reference and delete program variables.

The program uses variables to store values.

```
let x = 100    # Assigns the variable x to 100
```

Program variables begin with the '`$`' character.

```
let y = $x  #  Assigns the variable y to the value of x, y will
          have a value of 100
```

Program variables can be concatenated. Variables are evaluated from left to right.

```
let first = "John_"
let last = "Smith"
let name = $first$last # The variable name is assigned to
                    "John_Smith"
```

All variables in the sc are internally stored as character strings. The '`[]`' are used to control the order in which variables are evaluated. There are **NO** arrays in the sc programming language, but arrays can

be emulated using the "[]" characters. Program variables within the '[]' are evaluated first, then the entire variable is re-evaluated until all variables have been evaluated.

```
let color[1] = "Red"
let color[2] = "White"
let color[3] = "Blue"
for i=1 to 3
    print $color[$i] # Prints "Red" "White" and "Blue"
next i
```

In the above example three variables are created. The variable names are "color[1]", "color[2]", and "color[3]". The square brackets are part of the name of the variable. The only thing the brackets do is control the order of evaluation. In the case of let color[1] = "Red" there is no evaluation necessary since "color[1]" is the variable name itself. In the case of say let color[$i] = "Red" $i would be evaluated in determining what variable the text string "Red" would be assigned to. If $i were to evaluate to the value "eye" the variable "color[eye]" would be assigned to have a value of "Red". If it did not already exist it would be created. One feature of having arrays simulated as they are is that you can create a variable with a name like color[12345678901234567890] and not have to worry about some type of array out of bounds error. It is just a text string.

The '{}' work just like the '[]', except they are invisible. They just control the order of evaluation.

```
let color1 = "Red"
let color2 = "White"
let color3 = "Blue
for i=1 to 3
    print $color{$i} # Prints "Red" "White" and "Blue"
next i
```

Note In the above example, you can't have print $color$i. This is because $color$i is expecting two variables to be defined, color, and i. And only i exists. However, even if color existed you still can't have " $color$i " unless you use the {}'s.

Variables referenced but not defined have no value. A blank or empty value is returned and no error is generated.

Nesting and mixing of the '[]'s and '{}'s is allowed.

We can use multiple $'s for indirect variable addressing. When encountered, the right most $variable is evaluated first. See the following example:

```
let x = "Hello"
let y = "x"
```

```
print $$y # prints Hello
```

First the variable `$y` is evaluated to "`x`", and then the variable `$x` is evaluated to "`Hello`".

It is possible to delete variables. In the following example three variables will be created, `color[1]`, `color[2]`, and `color[3]`.

```
let color1 = "Red"
let color2 = "White"
let color3 = "Blue
```

In the case where you desire to delete sc's knowledge of and allocated storage space for a variable you may use the delete command as follows.

```
delete color3 # delete the variable and frees any allocated
          storage space
delete "color*" # this would cause the deletion of all three
            variables
```

Dumping sc Variables

This section discusses how to view variables stored in the sc.

The sc has three kinds of variables: *User*, *Temporary*, and *Internal*.

User variables are those variables created and used by the program. Once created a user variable is available (dependent on scope rules) for the duration program run.

Temporary variables are created by the sc and are available to the program via the `get_tmp_var ()` function. A temporary variable's lifetime is limited. Typically temporary variables are created when the sc receives an event and each subsequent event erases all previously created temporary variables.

Internal variables are created and used by the sc and are not available to the program.

The sc's `dump` command is used to view all sc variables.

Variable Context

This section discusses the context of program variables.

Program variables are initially created using the `let` or `define` assignment statements. If a variable is created with the `define` statement, it cannot be modified later.

Variables have context(scope). Variables created within a `begin / end` pair are only known to the statements within the same begin/end pair. Variables created external to any `begin / end` pair, are global to all statements.

Using Timers

This section discusses how to used timers in the program file.

Timers are used to generate events to the program. There are six (6) timers available for program use.

To start a timer, use the `start_timer` statement.

```
start_timer timer=timer_id seconds=seconds handler=label
```

To stop a timer, use the stop_timer statement.

```
stop_timer timer=timer_id
```

The timer_id identifies the timer. Valid entries are 0 to 5. The seconds are the number of seconds to delay before timing-out and jumping to the label. The label is where execution begins when the timer times-out.

Once a timer expires, it is no longer active, and must be restarted if so desired.

Here is a simple program that prints "`Hello`" every five seconds.

```
start_timer timer=1 seconds=5 handler=timeout1
label timeout1
print "Hello"
start_timer timer=1 seconds=5 handler=timeout1
```

All timers are suspended while the sc is executing statements, and are only evaluated when the sc is idle. Be careful when using `sleep` and looping `goto` statements, since they may keep the sc busy for a long time. However, sc, once it is idle, will eventually process timers even if they are overdue.

The sc uses real time (not idle time) to determine when a timer is due. In the example below, the addition of a 3 second sleep will not effect the "Hello" being printed every 5 seconds.

```
start_timer timer=1 seconds=5 handler=timeout1
sleep 3
label timeout1
print "Hello"
start_timer timer=1 seconds=5 handler=timeout1
sleep 3
```

It is OK to stop a timer that is not currently running.

An optional `msg` parameter is used to define a string. When the timer times-out, the `msg` is evaluated and converted into temporary program variables prior to jumping to the `handler` *label*. These variables can be used in the program following the `handler` label.

```
start_timer timer=timer_id seconds=seconds handler=label msg=msg
```

In the example below: The temporary variable, `status`, is set to "`Error`" if the timer times-out.

```
start_timer timer=1 seconds=5 handler=timeout1 msg="Timeout status=Error"
.
.
.
label timeout1
    print "Status is <" . get_tmp_var(status) . ">" # Prints the value of
                                                    status
```

The advantage to using `start_timer` instead of a sleep statement is that a `sleep` statement causes sc to basically stop functioning for the specified amount of time. In a simple application this may not be a problem, however, in a typical complex application is it not acceptable for sc to stop processing for a while. In this situation timers would be used. Basically a timer with a timeout handler is set up. Like any other statement, after the `start_timer` command is used sc will continue processing until it runs into the next `label` statement. When it hits the next label it will stop processing and wait for the next event. The event may be from an incoming message or it may be do to a timer expiring.

Error Handling

This section discusses how to handle program statement errors.

Most program statements support the error option.

```
error=label
```

The error option must appear at the end of the statement. The label is the program label statement where program execution jumps should the given statement fail.

If a statement fails, any statements following the failed statement are not executed. The following **bolded** statements are executed should `statement_2` fails.

statements...
statement_2... **`error=err`**
statements...
`label err`
statements...

In the case of sending messages, failures will be generated due to protocol failures. The program will need to handle any return error codes.

The `no_error` option may be appended to any statement. In the event of an error, the error is logged, but execution continues with the subsequent statements. The features allows for statements to fail but to have program flow continue as normal.

```
no_error
```

If a statement fails, statements following the failed statement are executed. The following **bolded** statements are executed should statement_2 fails.

statements...
statement_2... **`no_error`**
statements…

```
label err
  statements…
```

Reference

Program Statements

Program Statement (keyword)	Description

`after` *`label`*	This defines a program label and is synonymous with the `label` keyword. Labels behave in a unique manner in sc programs. Be sure and refer to FIXME for more information. The following two statements product an identical result:
	
	`after loop`
	`label loop`
	
	Note: You may include special characters line the "-" within the label by enclosing the value in single or double quotes.
	
	`label "remote_mode"`

`begin` `begin` *`name`*	Used to define the beginning of a subprogram. The `begin` statement without a specified *name* identifies the beginning of the default

Program Statement (keyword)	Description

	program(FIXME). In the normal case this is not necessary.
	
	`begin sub_program_1`
	`end`

`break`	Use to halt program execution.
	
	`if ((return eq "error)`
	`  print "Error detected"`
	`  break  # Stop, don't continue`
	`else`
	`  .`
	`  .`
	`  .`
	`end_if`
	
	The `break` statement does not cause the sc program to "die". It causes it to stop its current execution an go back to waiting for incoming messages or timer expirations.

`break_loop`	Used to exit a `for` or `while` loop.
	
	`for i = 1 to 10`
	`  if ($i == 5)`
	`  break_loop  # Exit loop and continues`
	`  else`
	`  .`
	`  .`
	`  .`
	`  end_if`
	`next i`
	
	Program execution resumes at the statement immediately following (in this case) the `next i` statement.

`close name=`*`name`*	Used to close a connection. The *`name`* is the name that was used in the open statement.
	
	`open socket_client name=equipment_sock …`
	`close name=equipment_socket`

Program Statement (keyword)	Description
`close client=`*`client`*	Used to close a connection on a channel that is acting as a server. *`client`* is the fd (file descriptor) of the connected client. FIXME `close client=$`*`client`*

`continue`	Used to continue execution when executing within a `for` or `while` loop.
	
	`For i = 1 to 10`
	`  if ($x[$i] eq “”)`
	`  continue`
	`  end_if`
	`next i`
	
	This operates in the same manner as most programming languages.

`define` *`var`*
`define` *`var=expr`*
	Used to define a program variable and optionally assign a value to it. `define`d variables are also used with preprocessor directives. Refer to the documentation for `#ifdef` FIXME above.
	
	`define x`
	`define x = 100`
	`define x = 5 * 100`
	
	Note: Once a variable is `define`d, it cannot be modified. For “normal” variable creation refer to `glet` and `let` detailed below.

Program Statement (keyword)	Description

`define_array` *`var`*`=` *`expr1`*`,` *`expr2`*`, ...`	Used as a method of convenience to create an **emulation** of an array. FIXME The name of the array is *`var`*. Its values are what expr1, expr2, etc. resolve to. The array *`var`* is indexed as follows *`var`*`[x]` where x starts at 1, and continues to increment by 1 for each *`expr`*x provided. The variable *`var`*`[0]` contains the number of elements in the array. Consider the following:
	
	`define_array days = Mon, Tue, Wed, Thu, Fri`
	
	This would result in 6 variables being created. The variable names would be `days[0]`, `days[1]`, `days[2]`, `days[3]`, `days[4]`, `days[5]`. The values stored in the variables are shown below.
	
	The value of the variable named `day[0]` is `5`.
	The value of the variable named `day[1]` is `Mon`.
	The value of the variable named `day[2]` is `Tue`.
	The value of the variable named `day[3]` is `Wed`.
	The value of the variable named `day[4]` is `Thu`.
	The value of the variable named `day[5]` is `Fri`.
	
	Remember, in order to get the value of the variable you need to prepend a $ (dollar) sign to the beginning of the variable name. For example the statement:
	
	`print $day[3]`
	
	would print out `Wed`.

Program Statement (keyword)	Description

`define_xref var = expr1, expr2, ...`	Used to create a cross reference variable. Multiple instances are used to create a cross reference table. When the *`var`* is index by *`expr1`*, *`expr2`* is returned, and when index by expr2, expr1 is returned.
	
	`define_xref color = red, FF0000`
	`define_xref color = white, FFFFFF`
	`define_xref color = blue, 0000FF`
	
	The variables below are assigned:
	
	The value of `color[red]` is `FF0000`
	The value of `color[FF0000]` is `red`
	The value of `color[white]` is `FFFFFF`
	The value of `color[FFFFFF]` is `white`
	The value of `color[blue]` is `0000FF`
	The value of `color[0000FF]` is `blue`
	
	This is a very handy feature. Say you have an incoming event that has the value `FF0000` in it. In the example above you could print the value of `color[FF0000]` (`$color[FF0000]`) and produce the much more user friendly value of red.

`delete "`*`var`*`"`	Used to delete program variable *`var`* and release any associated storage. This example deletes the program variable `x` and releases associated storage. `delete "x"`
`delete "var*"`	Used to delete **all** program variables that begin with the letters *`var`* and release any associated storage. This example deletes all of the program variables that begin with the characters `data[LOT1]` and releases their associated storage. `delete "data[LOT1]*"` Note: The asterisk may only appear at the end of the string.

Program Statement (keyword)	Description

`end`	Used to define the end of a subprogram.
	
	`begin subprogram1`
	`.`
	`.`
	`.`
	`end`

`exit`	Terminates the sc and stops it from running. Equivalent to the 'C' `exit(0).`

for loop construct	Used for looping. The `continue` and `break` commands are supported. There are two forms of the for-next loop construct. They are:
	
	`for x=1 to 10`
	`.`
	`.`
	`.`
	`next x`
	
	and
	
	`let max = 10`
	
	`for x=1 to $max by 2`
	`.`
	`.`
	`.`
	`next x`
	
	Note: The syntax for the `next` statement is `next` *`var`* (in this case `next x`), NOT `next $x`. Also, the for statement could look something like `for x=$start to $end by $step`.

Program Statement (keyword)	Description

`glet var = expr`	FIXME (verify) Used to assign a global variable a value. A global variable is created outside any subprogram.
	
	`let x = 100`
	
	`begin my_subprogram`
	`    # Change the global variable x's value to 200`
	`     glet x=200`
	`end`
	
	If a `let` was used instead of the `glet`, a new variable `x`, would be created that is only visible while program flow is within `my_subprogram`, and the global variable `x` would be unchanged. As an item of note sc will look 2 places for `x`, first within the context of the section bounded by the `begin-end` pair then outside the context of the the `begin-end` pair, but, not within the context of another `begin-end` pair.

Program Statement (keyword)	Description

`gosub` *`label`*
`gosub` *`label arg1, arg2, ...`*
	Used to jump to (call) a subroutine. A return statement is expect at the end of the subroutine. When subroutine encounters a `return` statement control flow resumes at the first statement following the `gosub` statement.
	
	`gosub sub_func_1`
	`.`
	`.`
	`.`
	
	`label sub_func_1`
	`.`
	`.`
	`.`
	`return`
	
	Arguments, `arg1`, `arg2`, `etc.` can be passed. Arguments may be passed by value, or by reference. In the following example, arg1 and arg2 are passed by value, while the variable `result` is passed by reference.
	
	`let result = 0 # Initialize the result`
	`let arg1 = 100`
	`let arg2 = 200`
	
	`gosub add_numbers $arg1, $arg2, result`
	`print "The result is ". $result`
	`.`
	`.`
	`.`
	
	`# Assign c the result of $a + $b`
	`label add_numbers a, b, c`
	`let c = $a + $b`
	`return`
	
	Note: The variable `result` must exist prior to the `gosub` statement.
	
	FIXME I need an example of using the passed in arguments.

Program Statement (keyword)	Description

`goto` *`label`*	Use to cause the program execution to start executing the first statement following the label define in the program.
	
	`goto turn_on_blue_light`
	`.`
	`.`
	`.`
	
	`label turn_on_blue_light`
	`.`
	`.`
	`.`
	
	FIXME reference the label explanation section

`Goto with arguments?`	FIXME

if then statement	Used for conditional program flow control.
	
	`if ($x == 100)`
	`.`
	`.`
	`.`
	`end_if`
	
	This reads as, if the value of the variable `x` is equal to `100` then execute the following statements up to the `end_if` statement.

if then else statement	Used for conditional program flow control.
	
	`if ($x <= 100)`
	`.`
	`.`
	`.`
	`else`
	`.`
	`.`
	`.`
	`end_if`
	
	This reads as, if the value of `x` is less than or equal to `100` then execute the statements between the `if` statement and the `else` statement, otherwise, execute the statements between the `else` statement and the `end_if` statement.

Program Statement (keyword)	Description

if then else if statement	Used for conditional program flow control.
	
	`if (($x == 100) or ($x == 200))`
	`.`
	`.`
	`.`
	`else_if ($x == 300)`
	`.`
	`.`
	`.`
	`else`
	`.`
	`.`
	`.`
	`end_if`
	
	This reads as, if the value of `x` is 100 or 200 then execute the statements between the `if` statement and the `else_if` statement, if the value of x is 300 then execute the statements between the `else_if` statement and the `else` statement, otherwise, execute the statements between the `else` statement and the `end_if` statement.

`label` *`label`*	Define a named point (label) within a program where control flow may be directed; usually as the result of a `goto` or `gosub` statement.
	
	`goto "my first label"`
	`.`
	`.`
	`.`
	`label "my first label"`
	`.`
	`.`
	`.`
	
	Note: If control flow is transferred to the label by a `gosub` statement there needs to be a `return` statement somewhere in the control flow after the label.
	
	FIXME reference the detailed explanation of labels
	l

Program Statement (keyword)	Description

`label` *`label arg1, arg2, ...`*	Define a named point (label) within a program where control flow may be directed. It the label is the target of a goto or gosub statement arguments may be passed.
	
	`let result = 0 # Initialize the result`
	`let arg1 = 100`
	`let arg2 = 200`
	
	`gosub add_numbers $arg1, $arg2, result`
	`print "The result is ". $result`
	`.`
	`.`
	`.`
	
	`# Assign c the result of $a + $b`
	`label add_numbers a, b, c`
	`let c = $a + $b`
	`return`
	
	Local variables a and b will be assigned to the values of arg1 and arg2. The variable c is assigned to the reference to result. FIXME what are a and b local to?
	
	Note: If control flow is transferred to the label by a gosub statement there needs to be a return statement somewhere in the control flow after the label.

`let` *`var=expr`*	Used to assign a variable to the value of an expression. If the variable does not already exist, it is created. `let x = 100`

Program Statement (keyword)	Description

`local_var` *var*
`local_var` *var=expr*
	Creates / defines a variable local to a label and optionally assigns it to the value of an expression. It only makes sense to use this within a label. By default, the variable is initialized to blank. FIXME (empty string?) In the example below, the variable `i` inside the label and the variable `i` before / outside the label are different:
	
	`let i = 200 # Assign i to a value of 200`
	`gosub do_it`
	`.`
	`.`
	`.`
	
	`label do_it`
	`local_var i # Define a local variable i`
	`let i = 300 # Assign the local i to 300`
	`.`
	`.`
	`.`
	
	FIXME can the outside variable be referenced at all in this case?

Program Statement (keyword)	Description
`open` **`port`** **`device=`*`device`* `name=`*`name`* `proto=`*`proto`* `logfile=`*`logfile`* `logging=`*`logging`* `options=`*`options`*	Open a connection for sending and receiving messages. In this form of the `open` statement `device` needs to be a a system file device such as "`/dev/tty01`" (a serial port). `name` is the name of the connection or channel. This name is use for a couple of things: 1. The value assign to `name` needs to correspond with a `label` in the running program. Whenever a message is received on this connection or channel, program flow will start at the first line after the `label` statement that has the same value as what was entered for the value of the `name` argument. 2. Any time the program needs to use any `send` statements it must use the value of the `name` argument for the connection or channel that the message needs to go out on. `proto` – The protocol to be used. The value of this argument must be one of 4 values: 1. `commands` – sc commands 2. `standard` – Simple text messages 3. `SECS` – Semiconductor Equipment Communication Standard SECSII messages transported over a SECSI layer. 4. `HSMS` - Semiconductor Equipment Communication Standard SECSII messages transported over a HSMS layer `logfile` – This provides the file path for a log file that will be used in writing out logging related messages for this connection. `logging` – This sets the logging level for this connection. The acceptable values for this argument are as follows: 0 - Turns off logging 1 - Prints out a one line message summary. 2 - Prints out the complete message content (default) 3 – Includes protocol details with the complete messages

Program Statement (keyword)	Description
open **pipe** **device=*device*** name=*name* *proto=proto* logfile=*logfile* logging=*logging* options=*options*	Open a connection for sending and receiving messages.
open **socket_client** name=*name* **host=*host*** **service=*service*** proto=*proto* logfile=*logfile* logging=*logging* options=*options*	Open a connection for sending and receiving messages.
open **socket_client** name=*name* **local_name=*local_name*** proto=*proto* logfile=*logfile* logging=*logging* options=*options*	Open a connection for sending and receiving messages.
open **socket_server** name=*name* **host=*host*** **service=*service*** proto=*proto* logfile=*logfile* logging=*logging* options=*options*	Open a connection for sending and receiving messages.
open **socket_server** name=*name* **local_name=local_name** proto=*proto* logfile=*logfile* logging=*logging* options=*options*	Open a connection for sending and receiving messages.

Program Statement (keyword)	Description
`pragma` *`pragma`*	Pragmas are used to control the operation of sc. Current pragmas include: `LOG_CPU_USAGE` – Logs a cpu usage message when sc runs. Default, no message is logged. `SHOW_NON_PRINTING_CHARS_IN_BRACES` – Prints non-printing characters as {hh}. This applies to the `print` statement and `dump` statement. By default, non-printing characters are printed as \xhh. `NO_BLOCK_ON_SLEEP` – Modifies the `sleep` command so that it will not block sc processing while sleeping. This means that sc uses an internal timer for the "sleep" and does not block. When the timer expires, execution resumes just as before. `NO_SOURCE_CODE_LINE_NUMBERS` – Source code filenames and line numbers are not shown when an error or warning is detected. `ALGEGRAIC_PRECEDENDE` – Enables algebraic precedence when solving equations. `MORE_SECS_TMP_VARIABLES` – Creates `.HEADER`, `.BODY`, and `.SYSTEM_BYTES` temporary variables from received secs messages.
`print` *`string`*	Used to output text. String concatenation may be used in the construction of the *`string`*. `print "The value of i is " . $i . " at the moment."` Notice that the "`.`" outside of the quoted strings is used as a concatenation operator.
`return`	Used to return execution flow to just after the prior `gosub` statement.
`send name=`*`name message`*	Used to send a message to an opened connection. The name is the name defined in the open statement. The message data is protocol specific. `send name=equipment_connection message`

Program Statement (keyword)	Description
`send name=`*`name`* `options=`*`options`* *`message`*	
`send name=`*`name`* `client=`*`client`* `options=`*`options`* *`message`*	The *`options`* defined in the open command can be temporarily overridden with the values defined by the *`options`* in this format of the send statement. The destination of the message can be to a connected client defined by the client. When you are a "server", "clients" connect to you. But suppose you wanted to send them a message (not a reply). To whom do you send the message? There may be several connected. You first need to get the client's id (`get_client()`). Then you can use that returned name (id value) to send a message. All of this is necessary because you could have multiple clients connected.
`send_ reply name=`*`name message`*	Used to send a reply message. The reply message is sent to the client connected to the connection. The message data is protocol specific. `send_reply name=name message  FIXME ident name`
`set_program` *`name`*	Used to activate a subprogram. The subprogram name becomes the active program. `set_program alternate_main` Note: Use `set_program default` to return to the default program. See the `begin` statement.
`sleep` *`seconds`*	Used to sleep for a specified number of seconds. The sleep is an inline sleep. All sc processing is suspended. This is equivalent to a 'C' sleep(). `sleep 20 # Sleep for 20 seconds` The pragma, "`NO_BLOCK_ON_SLEEP`", can be used to have sc not block or suspend processing while sleeping. FIXME...what does this mean? Refer to sleep down below

Program Statement (keyword)	Description

`start_timer timer=`*`timer_id`* `seconds=`*`seconds`* `handler=`*`label`* `msg=`*`msg`*	Used to start a timer. There can be up to 6 timers running at the same time. The *`timer_id`* identifies the timer. The *`seconds`* are the number of seconds (whole seconds) to wait, and the *`label`* is where execution will start when the timer expires. The *`msg`* is converted to a program temporary variable when the timer times-out. FIXME what does this msg look like and how is it used?
	
	`start_timer timer=1 seconds=5 handler=timer_1_timeout`
	`.`
	`.`
	`.`
	
	`label timer_1_timeout`
	`.`
	`.`
	`.`
	
	Note: Valid timer IDs are 0, 1, 2, 3, 4, 5.
	FIXME refer to the detailed explanation of timers

`stop_timer timer=`*`timer_id`*	Used to stop a timer. The *`timer_id`* identifies the timer to stop. No error is produced if timer that is not running is stopped. *stop_timer timer = 1* Note: Valid timer IDs are 0, 1, 2, 3, 4, 5.

`while(`*`expr`*`)`
`.`
`.`
`.`
`end_while`
	Your basic while loop. Executes the commands between the while and the end_while statements as long as the expression(*`expr`*) evaluates to true (non-zero). The while loop construct supports the `continue` and `break` statements.
	
	`let i = 1`
	`while($i <= 100)`
	`  if ($x[$i] eq “”)`
	`  continue`
	`  else if ($x[$i] eq “stop”)`
	`  break`
	`  else`
	`  print “x[“ . $i . “] is “ . $x[$i]`
	`  end_if`
	
	`  let i = $i + 1`
	`end_while`

Table 3: Program Statements

Preprocessor Directives

Preprocessor commands are resolved when the sc program is initially read and loaded. The `#` char must appear in the first column. One of the following sc command line options: `-P`, `-D`, or `-I` must be included.

Preprocessor Directive	Description
`#define` *`aname`*	Define "*`aname`*" within the program. See the `#ifdef` description below.

`#define` *`aname=value`*	Creates an unchangeable variable with the name of *`aname`* and assigns it the specified value.
	
	`#define SIRNAME=Jones`
	`print "My last name is " . SIRNAME`
	
	This will print out the following:
	
	"`My last name is Jones`"
	
	Note: It will be printed out without the double quotes. Also, the "`.`" in the `print` statement is a string concatenation operator.

`#include` *`filename`*	Include another program or parts of a program into the current program. `#include local_values.sc` The location of `local_values.sc` can be defined by the `-I` sc command line option. Note: The file extension specified ("`.sc`") is neither required, nor expected, nor assumed.

`#ifdef` *`aname`*
`.`
`.`
`.`
`#endif`
	If "*`aname`*" has been previously defined then all of the program's statements between the `#ifdef` and the corresponding `#endif` statement are considered part of the program and are executable.
	
	"*`aname`*" may be defined with a `#define`

Preprocessor Directive	Description
	statement as documented above or with the `-D` command line option when sc is started. Refer to table Table 1: Startup Command Line Options on page 9.

`#ifdef` *aname*
.
.
.
`#else`
.
.
.
`#endif`
	This is an if-then-else form of the `#ifdef` statement documented above.

Table 4: Preprocessor Directives

Comments

Comments are allowed by the methods below.

Comments	Description
#	Single line...to end of line
/* … */	Block ... everything between the /* and the */ are considered to be a comment and thus not executable. This comment form may span multiple lines.

Table 5: Comment Characters

Expressions

sc uses infix notation for expression evaluation. This is where the operator is place between 2 operands. In sc, all operators have equal precedence (there is no implied order of precedence). During expression evaluations, expressions are evaluated from right to left.

Always use parentheses for precedence. The table below shows this by example:

Expression to Solve	Result	Implied Parentheses
`5-4+1`	`0 (not 2)`	`5-(4+1)`
`6/2+1`	`2 (not 4)`	`6/(2+1)`

Table 6: Expressions

However, if the `pragma ALGEBRAIC_PRECEDENCE` is defined, expressions are solved using the precedence defined in the table below:

Precedence	Operator	
`1`	`* /`	Multiplication and division
`2`	`+ -`	Addition and subtraction
`3`	`< <= < <=`	Comparisons
`4`	`== !=`	Equality and non-Equality
`5`	`&`	BitWise AND
`6`	`|`	Bitwise OR
`7`	`^`	Bitwise exclusive OR
`8`	`and`	Logical AND
`9`	`or`	Logical OR

Table 7: Precedences

Math Operators

Math operations are used to perform simple arithmetic, add subtract, multiply, and divide. There is no operator precedence during evaluation of equations. All Operators have equal precedence.

Math Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division

Table 8: Math Operators

Unary Operators

Unary operations are used to modify a value prior to evaluation.

Math operator	Description
+	Plus
−	Minus
~	One's complement

Table 9: Unary Operators

Relational Operators

Relational operators are used in expressions (if, else_if). Relational operations return either true or false. True is non-zero, and false is zero.

Numeric

Used with number values

Numeric relational operator	Description
`expression1 == expression2`	Returns true if expression1 is equal to expression2
`expression1 != expression2`	Returns true if expression1 is not equal to expression2
`expression1 > expression2`	Returns true if expression1 is greater than expression2
`expression1 < expression2`	Returns true if expression1 is less than expression2
`expression1 >= expression2`	Returns true if expression1 is greater than or equal to expression2
`expression1 <= expression2`	Returns true if expression1 is less than or equal to expression2

Table 10: Numeric Relational Operators

String conditionals

Used with string variables

String relational operator	Description
`string1 eq string2`	Returns true if string1 is identical to string2
`string1 ne string2`	Returns true if string1 is not identical to string2

Table 11: String conditionals

Boolean Operators

Boolean operators are used to test for either true for false. True is non-zero and false is zero.

Boolean operator	Description
`expression1 and expression2`	Returns true if expression1 and expression2 are true.
`expression1 or expression2`	Returns true if either expression1 or expression2 are true.
`!expression`	Returns true if expression is false, or false if expression is true.

Table 12: Boolean operators

Bitwise Operators

Bitwise operators perform bitwise operations on the arguments.

Bitwise operator	Description
`value1 & value2`	Bitwise and- ing of value1 and value2
`value1 | value2`	Bitwise or- ing of value1 and value2
`value1 ^ value2`	Bitwise xor- ing of value1 and value2

Table 13: Bitwise operators

Constants

Valid constants for assignment to variables:

Example constants
Integer numbers: `100`, `500`, `-6`, etc.
Floating point numbers: `1.2`, `5.1`, etc.
Strings: "`This is a string.`"
String characters:
`\xhh` – Hexidecimal value from `00` through `FF`
`\n` – new line character
`\r` – carriage return character

Table 14: Examples of constants

String Concatenation

Strings may be concatenated using the '.' and ':' characters. The '.' appends the strings, while the ':' appends with a single white space between the strings. The following are equivalent.

```
let name = "John" . " " . "Smith"  # Assigns "John Smith" as the value of name
let name = "John" : "Smith"        # Assigns "John Smith" as the value of name
```

Keywords

The following is a list of keywords that are reserved and should not be used in the sc program as variable names, etc. However, they can be used in double quoted strings.

Keywords				
`after`	`else`	`format`	`ne`	`set_program`
`and`	`else_if`	`glet`	`next`	`sleep`
`begin`	`end`	`gosub`	`open`	`start_timer`
`break`	`end_if`	`goto`	`or`	`step`
`break_loop`	`end_while`	`if`	`pragma`	`stop_timer`
`close`	`eq`	`import`	`print`	`system`
`continue`	`error`	`is_label`	`proto`	`then`
`define`	`exit`	`ltoi`	`return`	`to`
`define_array`	`export`	`label`	`send`	`while`
`define_xfer`	`for`	`let`	`send_reply`	
`delete`		`local_var`		

Table 15: Language Reserved Words

Functions

The following functions can be used in any expression:

Function	Description
`abs(value)`	Absolute value. Returns the absolute value for value. Built on the 'C' `abs()` function)
`atoi(char)`	Converts the char to its integer equivalent. If a string is passed, only the first char is converted.

Function	Description
	`let x = atoi("A") # Assigns x to 65`
`btoi(value)`	Binary to integer conversion. Returns the binary value as an integer. (Built on the 'C' `strtol()` function) `let i = btoi (000101) # Assigns i to 5`

`checksum(type, data)`	Returns a checksum for the passed data. Allowed types are:
	`ADD` – Sums all the bytes in data
	`ADD1` – Sums all the bytes in data. Returns a one byte value.
	`ADD2` – Sums all the bytes in data. Returns a two byte value.
	`ADD4` – Sums all the bytes in data. Returns a four byte value.
	`XOR` – Exclusive OR of all the byes in data
	`CRC` – Calculates the CRC checksum for data
	
	`let x = checksum (ADD, $data)`
	`let x = checksum (XOR, $data)`

`dump()`	Works like the dump command.
	
	`dump()`
	`dump(x)`

`export (file, mode, data)`	Writes the data to the file. The file is open with the mode options. Modes are defined below. If the mode contains a 'b', the data is assumed to be hexadecimal and is converted back to binary when written. `export("recipe", "wb+", $buf)`

`file_close(fd)`	Closes the previously opened file referenced by fd. Fd was returned by a `file_open()` statement.
	
	`let fd = file_open ("data", "r")`
	`file_close ($fd)`

`file_open(file, mode)`	Opens the file, file, with the mode, and returns a file handle. See the table of valid modes below.
	
	`# Open the file "data" as read only, and assigns fd to the file handle`
	
	`let fd = file_open ("data", "r")`

Function	Description

`file_orc(file, mode)`
`file_orc(file, mode, string)`
	Opens the file and reads and returns the desired line. Mode specifies the line to read, 1 to x. If mode is –1, then the last line is returned. If the line can't be found, a EOF is returned.
	
	If a string is specified, the line containing the string is returned. If mode is 1, then the first occurrence is returned. If mode is –1, then the last occurrence is returned. If no line is found, an EOF is returned.
	
	`# Return the first line`
	`let line = file_orc("file", 1)`
	
	`# Return the third line`
	`let line = file_orc("file", 3)`
	
	`# Return the last line`
	`let line = file_orc("file, -1)`
	
	`# Return the first line containing "hello"`
	`let line = file_orc("file", 1, "hello")`
	
	`# Return the last line containing "hello"`
	`let line = file_orc("file, -1, "hello")`

`file_owc(file, mode, data)`	Opens the file with the mode, writes the data, and closes the file. See the table of valid modes below. A new line character is added to the end of the data.
	
	`# Writes "Hello to you" to a "data" file.`
	`file_owc ("data", "w+", "Hello to you")`

`file_read(fd)`	Read from a previously opened file referenced by the fd. File_read() returns the string "EOF" when an end of file is encountered. `# Assigns line to the value read.let line = file_read ($fd)`

`file_write(fd, data)`	Writes the data, data, to a previously opened file referenced by the fd.
	
	`let fd = file_open ("data", "w")`
	`file_write ($fd, "This is a test")`

Function	Description

`format(str, fmt)`	Format a string. Used to format the `str` using the specified `fmt`. Equivalent to the 'C' `sprintf(buf, %fmts, str)` or `sprintf(buf, %fmtg, str)` depending upon the data type of str. If it is a number, `%fmtg` is used, otherwise `%fmts` is used. `format()` returns the formatted string (the value of `buf`).
	
	`let x = format ("123", "05") # Assigns "00123" to x`
	`let x = format ("1.23", "8.3") # Assigns " 1.23" to x`
	
	If the fmt string contains a "%" character, the fmt is used as defined. This is equivalent to the 'C' sprintf(buf, fmt, str), and buf is returned.
	
	`Let x = format(2.2345678", "%2.3f") # Assigns 2.235 to x`
	`Let x = format(2.3456e-03", %3.3f") # Assigns 0.002 to x`
	`Let x = format("John", "Hello %s") # Assigns "Hello John" to x`

`ftoh(value)`	Returns the float value as hex. `let i = ftoh(1.1) # Assigns 3F8CCCCD to i`
`get_client(name)`	Returns the client connected to name. If multiple clients are connected, the last client to send a message is returned.

`get_env(variable)`	Get an environment variable. Returns the environmental variable's value of variable. If the variable is not defined, a blank ("") value is returned.
	
	`let x = get_env (TERM)`
	`# Assigns x to "hpterm", or whatever TERM has been`
	`# assigned to by the shell.`

Function	Description

`get_tmp_var(variable)`
`get_tmp_var("")`
`get_tmp_var(variable, default)`
	Get a temporary variable. Returns the value of the temporary variable. If the variable is "" or blank, all temporary variables are returned. If the tmp variable does not exist, a blank value is returned. However, if the optional default value is included, that value will be returned.
	
	`let x = get_tmp_var (ARG0)`
	or
	`let x = get_tmp_var ("ARG0") # Assigns x to the value of ARG0`
	
	`print get_tmp_var ("") # Prints all of the temporary variable`
	
	`let x = get_tmp_var(num_wafers, 24) # Assigns x to the tmp variable num_wafers or 24 if it does not exist`

`get_tmp_var_type(variable)` `get_tmp_var_type("")`	Get the data type of a temporary variable. Returns the data type of a SECS temporary variable. For all other (non SECS) temporary variables, nothing is returned. `let x = get_tmp_var_type(S1F4.1.2)`
`get_version()`	Get the current version of sc. `let version = get_version()`

`htof(value)`
`htof(value, fmt)`
	Hex to float conversion. Returns the hex value as a float. If a fmt is supplied, the returned float if formatted.
	
	`let f = htof(3F8CCCCD) # Assigns 1.100000 to f`
	`let f = htof(3F8CCCCD, %4.3f) # Assigns 1.100 to f`

`htoi(value)`	Hex to integer conversion. Returns the hex value as an integer. (Built on the 'C' `strtol()` function)
	
	`let i = htoi (FF) # Assigns 255 to i`
	`let i = htoi (7FF) # Assigns 2047 to i`
	`let i = htoi(80000000) # Assigns -2147483648 to i`

`htou(value)`	Hex to unsigned integer conversion. Returns the hex value as an unsigned integer. (Built on the 'C' strtol() function) let i = htou(FFFF) # Assigns 65535 to i let i = htou(FFFFFFFF) # Assigns 4294967295 to i
`is_number(value)`	Returns `1` (true) if the value is a valid number. Otherwise it returns `0` (false). `if (is_number($x)) # Continue only if x is a valid number that can be used in an expression.`

Function	Description
`is_print(char)`	Returns `1` (true) if the char is printable, otherwise returns `0` (false). `if (is_print("\x00")) # is_print would return 0 (false)`
`import (file, mode, var)`	Reads the `file` into `var`. The file is open with the mode options. Modes are defined below. If the mode contains a '`b`', the data is assumed to be binary and is converted to hexadecimal when read. Import returns the number of bytes read. `count = import("recipe", "rb+", buf)`

`instring (str1, str2)`	String within a string. This is used to determine if string `str2` is contained within string `str1`. If found, `instring ()` returns the index in `str1` where `str2` is found (starting at position 1). `instring()` returns `0` if `str2` is not contained within `str1`.
	
	`let x = instring("my name is john", "name") # Assigns x to 4`
	
	`let x = instring("my name is john", "bill") # Assigns x to 0`

`is_label(label)`	Used to determine if a label exists. Returns `1` if the `label` exists in the program, otherwise, returns `0`.
	
	`if (is_label (xxx))`
	`  goto xxx # Jumps to the label xxx only if it exists.`
	`end_if`

`is_timer(timer)`	Returns the time remaining (in seconds) for the given timer or `0` if the timer has expired.
	
	`# Get the time remaining for timer 2`
	`let time_remain = is_timer(2)`

`is_tmp_var(variable)`	Used to determine if the temporary variable exists. Returns `1` (true) if the temporary variable exists in the program, otherwise, returns `0`(false).
	
	`if (is_tmp_var(num_wafers))`
	`....`
	`end_if`
	
	Remember to use the variable name, not its value (i.e. `num_wafers`, not `$num_wafers`)

Function	Description

`is_var(variable)`	Used to determine if the variable exists. Returns 1 if the variable exists in the program, otherwise, returns 0.
	
	`if (is_var(lot_id))`
	`....`
	`end_if`
	
	Remember to use the variable name, not its value (`lot_id`, not `$lot_id`)

`itoa(value)`	Integer to ASCII conversion. Returns the printable ASCII character equivalent of value. If there is no printable equivalent, a "`?`" is returned.
	
	`let c = itoa(65) # Assigns 'A' to c`
	`let c = itoa(20) # Assigns '?' to c`

`itob(value, fmt)`	Integer to binary conversion. Returns the integer value as a binary number. Internally, the value is first converted to a hex value, and then converted to a binary value. The conversion is specified using the `fmt` value. This is equivalent to the 'C' sprintf(str, "%fmtX", value). The hex string is next converted to a binary string (four binary characters for each hex character) and then returned. `let x = itob (255, 4.4) # Assigns "0000000011111111" to x`
`itoh(value, fmt)`	Integer to binary conversion. Returns the integer value as a binary number. Internally, the value is first converted to a hex value, and then converted to a binary value. The conversion is specified using the `fmt` value. This is equivalent to the 'C' `sprintf(str, "%fmtX", value)`. The resulting hex string is next converted to a binary string (four binary characters for each hex character) and then returned. `let x = itob (255, 4.4) # Assigns "0000000011111111" to x`
`itoo(value, fmt)`	Integer to octal conversion. Returns the integer value as an octal number. The conversion is specified using the `fmt` value. This is equivalent to the 'C' `sprintf(str, "%fmto", value)`. The string value is returned. `let i = itoo(12, 3) # Assigns 014 to i`
`mkprint(data)`	Converts the `data` to printable characters.

Function	Description
`otoi(value)`	Octal to integer conversion. Returns the octal value as an integer. (Built on the 'C' `strtol()` function.) `let i = otoi(77) # Assigns 63 to i`

`parse(str, delimiters, data)`
`parse(str, delimiters)`
	Parse a string into tokens. The results are assigned to the variable `data[]`, the tokens parsed from `str` delimited by any of the characters contained within `delimiters`. `parse()` returns the number of `data` variables assigned. The array starts with the index 1. If `data` is omitted, no variables are assigned. This similar to the `define_array` command, but populated by a parsing effort and is typically used to break up something like a csv file.
	
	`let count = parse ("aaa.bbb.ccc", ".", "data")`
	`# Assigns data[1] to "aaa"`
	`# Assigns data[2] to "bbb"`
	`# Assigns data[3] to "ccc"`
	`# Assigns cnt to 3`

`pow(x, y)`	Returns the value of `x` raised to the power of `y`. `let x = pow (2,3) # Assigns x to 8`
`rand([seed])`	Returns a random number between `0` and `65535`. The optional seed is used to initialize the list or random numbers. `rand()` is based on the 'C' `srand()` and `rand()` functions.

Function	Description

`save_tmp_var(my_data)`
check this out
	Saves all of the temporary variables to the variable `my_data`. `my_data`'s previous value will be replaced.
	
	If the text:
	`"3 colors first=red second=white third=blue"`
	is received, the following variables are created with the values shown:
	
	`my_data[ARGS] = "3 colors first=red second=white third=blue"`
	`my_data[ARG1] = "3"`
	`my_data[ARG2] = "colors"`
	`my_data[first] = "red"`
	`my_data[second] = "white"`
	`my_data[third] = "blue"`
	
	Also, the variable "`my_data`" is created. This contains a comma-separated list of all the names of the variables created. **DO NOT** alter this. **sc** first removes this list of variables before creating new ones.
	
	`my_data = "my_data[second],my_data[ARG1],my_data[ARG2],my_data[ARGS],..."`

`scan(str, fmt)`	Scan a string. Used to scan a string, `str`, for text using the supplied `fmt`. Equivalent to a 'C' `sscanf(str, fmt%s, buf)`. The value of `buf` is returned. In the example below, we want to get the data following `"bobo t:"` `let x = scan ("bobo t:99", "t:") # Assigns "99" to x`

Function	Description
`set_system_options(options)`	Sets options for the `system()` and `system_shell()` functions. These options allow for a timeout period, a message returned upon a timeout, and characters not to parse from the returned value. Allowed options are listed below (in the form <name>=<value>): `timeout` - Seconds to wait for a reply `no_parse` – Don't parse these characters from the returned value `timeout_msg` - Message to return upon a timeout `set_system_options("timeout=30 no_parse=\",\" timeout_msg=TIMEOUT")` (see `system()` and `system_shell()`)
`set_tmp_var(variable, value)`	Create and assign a temporary variable to value. This temporary variable is valid until the next message is received. set_tmp_var(num_wafers, 25)

`sqrt(value)`	Get the square root. Returns the square root of `value`.
	
	`let x = sqrt (16) # Assigns 4 to x`
	`let x = sqrt (15) # Assigns 3.87298 to x`

`strftime (fmt)`
`strftime(fmt, value)`
	Format time. Return a formatted time string using the format defined by `fmt`. The `strftime` function is built upon the 'C' `strftime()` function. See the table of valid `fmt`s below. The value is optional. If supplied, the `strftime` function uses the value instead of the system time. The value can be initially generated with the `time()` function.
	
	`let x = strftime ("%D %T")`
	`# Assigns x the date and time formatted as:`
	`"mm/dd/yy hh:mm:ss"`
	`# using current time`
	`let t = time()`
	`let x = trftime(""D %T", $t)`
	`# Assigns x the date and time formatted`
	`# as: "mm/dd/yy hh:mm:ss" using the value of t for time`

`strlen(str)`	Get the length of a string. Returns the length of the string `str`. `let x = strlen ("ABCDEFG") # Assigns x to 7`

Function	Description
`substr(str, index, count)`	Get a sub-string. Returns `count` number of characters from the string, `str`, starting with the character at `index` (index starts at 1). In this example, beginning with the 7th character return 2 characters. `let x = substr("Hello to you", 7, 2) # Assigns x to "to"`

`system (command)`
`system (command, no_parse)`
	Executes a shell command. The `system()` function is built upon the 'C' `popen()` and `pclose()` functions. The command is executed as a shell command. `stderr` is redirected to `stdout`. Both `stderr` and `stdout` are received by sc and that data is converted into sc temporary program variables.
	
	`let x = system ("echo name=john")`
	`# Assigns 0 to x, and creates a temporary variable, "name",`
	`# assigned to "john"`
	
	If the optional `no_parse` string is included, sc will not parse those characters using the parsing rules. (see `set_system_options()`)

`system_shell (command)`	Executes a shell command. This is implemented using the 'C' system() function. The exit status of the command is returned. The system_shell() differs from the system() in that the system() is build upon the 'C' popen() function while the system_shell() is build upon the 'C' system() function. No temporary variables are created. . (see set_system_options()) let cmd = "my_shell". arg1 : arg2 let x = system ($cmd) # Arg1 and arg2 are passed to the my_shell script. # The exit status of my_shell is assigned to x let x = system_shell ("xxx") # Assigns 127 to x if the script xxx does not exist let x = system_shell ("exit 10") # Assigns 10 to x # Note that the system_shell() is better suited for those cases where commands are executed in the background.

Function	Description
`test(str)`	Tests and evaluates the str. Build on the Unix "test" command. Returns either True or False (0). if (test("-r my_file")) # Returns true if my_file exists and is readable.
`time()`	Get the current time. Returns the system time as an integer. Uses the 'C' time() function. let t = time()
`tolower(str)`	String conversion. Converts str to lower case. Returns the converted string. let x = tolower ("AbCdEfG") # Assigns x to "abcdefg"
`toupper(str)`	String conversion. Converts str to upper case. Returns the converted string. let x = toupper ("AbCdEfG") # Assigns x to "ABCDEFG"
`trace(0|1)`	Turn on and off tracing while the program is running. trace(0) # Turns off tracing trace(1) # Turns on tracing
`t_parse(str, delimiters, data)`	Parse a string into tokens. Works like parse(), except temporary variables are assigned. Note: All temporary variables are valid until the next message is received, so be sure and copy them off as soon as possible.

Table 16: Functions

Valid values for `fmt`:

Table 17: Valid values for fmt

Format	Description
%a	Locale's abbreviated weekday name.
%A	Locale's full weekday name.
%b	Locale's abbreviated month name.
%B	Locale's full month name.
%c	Locale's appropriate date and time representation
%C	The century number (the year divided by 100 and truncated to an integer) as decimal

Format	Description
	number [00-99].
%d	Day of the month as a decimal number [01,31].
%D	Equivalent to the directive string %m/%d/%y.
%e	Day of the month as a decimal [1,31]; a single digit is preceded by a space.
%h	Equivalent to %b.
%H	Hour (24-hour clock) as a decimal number [00,23].
%I	Hour (12-hour clock) as a decimal number [01,12].
%j	Day of the year as a decimal number [001,366?].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%n	The New-line character.
%p	Locale's equivalent of either AM or PM.
%r	The time in AM and PM notation; in the POSIX locale this is equivalent to %I:%M:%S: %p.
%R	The time in 24 hour notation (%H:%M).
%S	Second as a decimal number[00,61].
%t	The Tab character.
%T	The time in hours, minutes, and seconds (%H:%M:%S).
%u	The weekday as a decimal number [1(Monday),7].
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
%V	The week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing January 1 st has four or more days in the new year, then it is considered week 1; otherwise, it is week 53 of the previous year, and the next week is week 1.
%w	Weekday as a decimal number [0(Sunday),6].
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.
%x	Locale's appropriate date representation.

Format	Description
%X	Locale's appropriate time representation.
%y	Year without century as a decimal number [00,99].
%Y	Year with century as a decimal number.
%Z	Time zone name (or by no characters if no time zone exists).

Table 18: Valid entries for fmt

Valid values for `modes`:

Mode	Description
`r`	open for reading
`w`	truncate to zero length or create for writing
`a`	append; open for writing at end of file, or create for writing
`rb`	open binary file for reading
`wb`	truncate to zero length or create binary file for writing
`ab`	append; open binary file for writing at end-of-file, or create binary file
`r+`	open for update (reading and writing)
`w+`	truncate to zero length or create for update
`a+`	append; open or create for update at end-of-file
`r+b` or `rb+`	open binary file for update (reading and writing)
`w+b` or `wb+`	runcate to zero length or create binary file for update
`a+b` or `ab+`	append; open or crate binary file for update at end- of-file

Table 19: Valid values for modes

Protocols

This section will be addressed in another revision.

Command Protocols

The commands protocol is used to send sc commands to the sc. See the section on sc Commands for a complete list of commands supported by the sc.

Typically the sc receives commands via a file socket connection. This connection can be defined in two ways: at sc startup time using the -s option, and with a program `open` command.

To used the sc startup option, use the -s option to specify the file socket.

```
sc –n main –d 0 –s sockets/s.main –c "read program" >> log 2>$1 &
```

The file socket, sockets/s.main, is created and ready to receive sc commands. Use sc_talk to connect to this socket.

```
sc_talk –s sockets/s.main
```

You can also use the program to define a file socket to receive sc commands.

```
open socket_server proto=commands...
```

It is simpler to use the –s option instead of the open command.

The options for the command protocol are as follows:

Standard Options	
`crlf=yes | no`	Converts the newline char to a carriage return linefeed. This can be used when interfacing to sc via a socket connection using a telnet or hyperterm application.

Table 20: Command Protocol Options

Standard Protocols

The standard protocol is used to send messages to other sc's or other servers connected to a socket, file

socket, or port connection.

The sc can act as a client, or server in order to send and receive messages.

As a client, the sc connects to an established connection, sends a message, and optionally waits for a reply.

open socket_client name=recipe_server proto=standard ...

Connects as a client to a recipe_server

As a server, the sc establishes the connection and waits for a message from a connected client, and may or

may not reply.

open socket_server name=recipe_server proto=standard ...

Creates the recipe_server socket connection

As a client, the name parameter is the destination for the message in the send command.

open socket_client name=recipe_server proto=standard ...

send name=recipe_server

As a server, the name parameter is the label where the received message is handled

open socket_server name=recipe_server proto=standard ...

label recipe_server

Messages from the client come here

send_reply name=recipe_server ...

The connection (or type of open) can be a: pipe, socket, a file socket, or a port for both the client and server.

For a socket connection, specify the host machine and service name for the socket.

open ... name=recipe_server proto=standard host=digi service=2100 ...

For a file socket connection, specify the local socket file name.

open ... name=recipe_server proto=standard local_name=sockets/s.recipe_server …

For a port device file, specify the device file.

```
open port name=scanner proto=standard device=”/dev/tty0p0” ...
```

For a Unix named pipe, specify the filename.

```
open pipe name=scanner proto=standard device=”host” …
```

Standard Options	
`error=label`	Defines a label to jump to if there is a protocol error.
`flush=time`	Defines a time period, in seconds, to wait and discard all incoming data. This is used when servers initially connect to a device, such as a terminal server, and want to ignore initialization messages or any other characters to be discarded.
`no_parse= str`	Defines a list of characters, str, that are used to override the parsing rules (see the section on parsing rules). When the parser detects one of these characters,

	it does not follow the rules for parsing, but simply includes that character in the token. Assume the reply contained "data=-100", normally a temporary variable "data" is assigned the value "-", and ARG0 would be assigned "100". However, by specifying a str value of "- ", the temporary variable "data" would be assigned "-100".
`rcv_eol=str`	Defines a string, str, used to detect the end of messages received. This is how the sc determines the end of a message.
`raw=no|yes`	Does no interpretation of the bytes being sent or received. If no, default, non-printing bytes are converted to a 2 byte hex value.
`timeout_msg=str`	Defines a string, str that will be returned if a time-out is detected. In the event of a time- out, the sc returns the str as if it was the actual reply message. This allows the sc program to test for time-outs when sending messages and waiting for replies. Any partially returned message is included with the reply. The same applies while receiving unsolicited messages. The str and the partially received message are sent to the message handler.
`timeout=time`	Defines a time-out period, time, in seconds to wait for a reply message. This is used with the send command. When a message is sent, the sc waits for a reply message. If the time-out period has elapsed, sc no longer waits for the reply. If not specified, a default value of 60 seconds is used.
`mp_var=tolower|toupper`	This option will convert the names of tmp variables to the case specified. This applies to both unsolicited messages and replies. Conversion only applies to tmp variables of the name=value format. It does not apply to names such as ARG0, ARG1, etc.
`wait_for_ reply=no`	Instructs the send command to not wait for a reply message. If not specified, the default is to wait for a reply.

xmt_eol=str	Defines a string, str, to append to the message sent via the send command. The str will not be part of any name=value tmp variables, but it will be included in the ARGS tmp variable.

Table 21: Standard Protocol Options

Example

The following example is a driver program used to interface a serial barcode scanner to a host program. The scanner is connected to the host system via a serial cable. The scanner can operate in two modes:scan-on-command (scanner is sent a scan command and returns the barcode), or scan-on-detect (scanner automatically detects a barcode and sends the barcode to the driver).

If the driver receives a "read" command from the host, it will read the barcode and return the value. If a barcode is placed in front of the reader, a read event, containing the barcode, is sent to the host.

```
# Open a serial connection to the scanner

open port name=scanner proto=standard device=”/dev/tty0p0” \

options=”baud=9600 rcv_eol=\”\r\” xmt_eol=\”\r\” timeout=10
timeout_msg=\”TIMEOUT\””

# Open a server connection to wait for commands from the host

open socket_server name=driver proto=standard \

local_name=”sockets/s.driver_command” options==”rcv_eol=\”\n\””

# Open a client connection to the host to send event messages to the host

open socket_client name=host proto=standard \

local_name=”sockets/s.driver_event” options==”xmt_eol=\”\n\” wait_for_reply=no”

# Wait for a command from the host. Upon a read command, send a scan command to the
barcode scanner and wait for a reply.

label driver

let cmd = get_tmp_var (cmd) # Expecting a “cmd=read\n” message

if ($cmd == “read”)

# Send a “scan” command to the scanner. The scan command is “\r” terminated

send name=scanner “scan”

# Wait 10 seconds for the reply. The reply comes in the format “value/r”.

# ARG0 is set to either “TIMEOUT” or the barcode scanned.
```

```
let return = get_tmp_var (ARG0)
if ($return eq "TIMEOUT")
# Time-out has been detected, so handle the time-out here.
# Send an error to the host "error error_msg="timeout detected"\n".
# The host creates two temporary variables ARG0 (set to error), and error_msg.
send_reply name=driver "error error_msg=\"timeout detected\""
else
# Send the barcode to the host as "success barcode=value\n".
# The host creates two temporary variables ARG0 (set to success), and barcode.
send_reply name=driver "success barcode=" . $return
end_if
end_if
break
```

Wait for an automatically detected barcode scan from the scanner

```
label scanner
let scan = get_tmp_var (ARG0) # Get the scanned value
# Send the barcode to the host. No reply is expected.
# The host creates two temporary variables, ARG0 (set to autoscan), and barcode.
send name=host "autoscan barcode=" . $scan
break
```

XML Protocol

The xml protocol is used to send messages to other sc's or other servers connected to a socket, file socket, or port connection.

The sc can act as a client, or server in order to send and receive messages in xml format.

As a client, the sc connects to an established connection, sends a message, and optionally waits for a reply.

```
open socket_client name=recipe_server proto=xml ...
# Connects as a client to a recipe_server
```

As a server, the sc establishes the connection and waits for a message from a connected client, and may or may not reply.

```
open socket_server name=recipe_server proto=xml ...
```

```
# Creates the recipe_server socket connection
```

As a client, the name parameter is the destination for the message in the send command.

```
open socket_client name=recipe_server proto=standard ...
send name=recipe_server
```

As a server, the name parameter is the label where the received message is handled

```
open socket_server name=recipe_server proto=xml ...
label recipe_server
# Messages from the client come here
send_reply name=recipe_server ...
```

The connection (or type of open) can be a: pipe, socket, a file socket, or a port for both the client and server.

For a *socket connection*, specify the host machine and service name for the socket.

```
open ... name=recipe_server proto=xml host=digi service=2100 ...
```

For a *file socket connection*, specify the local socket file name.

```
open ... name=recipe_server proto=xml local_name=sockets/s.recipe_server ...
```

For a *port device file*, specify the device file.

```
open port name=scanner proto=xml device="/dev/tty0p0" ...
```

For a Unix named *pipe*, specify the filename.

```
open pipe name=scanner proto=xml device="host" …
```

The options for the xml protocol are as follows (see also port options):

Standard Options	
`error=label`	Defines a label to jump to if there is a protocol error.
`flush=time`	Defines a time period, in seconds, to wait and discard all incoming data. This is used when servers initially connect to a device, such as a terminal server, and want to ignore initialization messages or any other characters to be discarded.
`timeout_msg=str`	Defines a string, str that will be returned if a time-out is detected. In the event of a time- out, the sc returns the str as if it was the actual reply message. This allows the sc program to test for time-outs when sending messages and waiting for replies. Any partially returned message is included with the reply. The same applies while receiving unsolicited messages. The str and the partially received message are sent to the message handler.
`timeout=time`	Defines a time-out period, time, in seconds to wait for a reply message. This is used with the send command. When a message is sent, the sc waits for a reply message. If the time-out period has elapsed, sc no longer waits for the reply. If not specified, a default value of 60 seconds is used.
`wait_for_ reply=no`	Instructs the send command to not wait for a reply message. If not specified, the default is to wait for a reply.

Table 22: XML Protocol Options

Serial Port Options

When using the open port device file, you can specify the following port options. By default, the port settings are 8N1 (8 bit, no parity and 1 stop bit). The baud has no default and needs to be defined. These port options are optionally used by the standard and xml protocols.

Standard Options	
`baud=value`	Defines the baud rate. Valid values are: 300, 600, 1200, 2400, 4800, 9600, 19.2k, 57.6k, 115.2k, 128k, 230.4k, and 256k. There is no default baud rate value.
`parity=value`	Defines the parity. Valid values are: even or odd. The default is no parity.
`bits=bits`	Defines the number of data bits, either 7 or 8 (bits=7 or bits=8). The default is 8 bits.
`sbits=sbits`	Defines the number of stop bits, either 1 or 2 (sbits=1 or sbits=2). The default is 1 stop bit.

Table 23: Serial Port Options

Variables in XML Replies and Unsolicited Messages

This section discusses how to handle the XML replies and events messages.

Whenever the sc receives a XML reply message or an unsolicited message, the sc converts that message in temporary variables. These variables are accessed via the get_tmp_var () function. The following XML example is used in the following discussion:

XML Item
<movies>   <movie>   <title>PHP: Behind the Parser</title>   <characters>   <character>   <name>Ms. Coder</name>   <actor>Onlivia Actora</actor>   </character>   <character>   <name>Mr. Coder</name>   <actor>El ActÓr</actor>   </character>   </characters>   <plot>   So, this language. It's like, a programming language. Or is it a scripting language? All is revealed in this thrilling horror spoof of a documentary.   </plot>   <rating type="thumbs">7</rating>   <rating type="stars">5</rating>   </movie> </movies>

Table 24: XML Item

The entire XML message is converted into temporary variables. Each item in the XML message is named according to its position within the XML layout, (like xml.1.1). And are appended with either a tag name, attribute name, or value, depending upon the context of the tag.

Tag items, like <movies> and <movie> are named as shown below (appended with a . tag). Closing tags, like </movie> and </movies> are not converted to temporary variables.

XML Item	Temporary Variable	Value
<movies>	xml.1.tag	movies
<movie>	xml.1.1.tag	movie
</movie>		
</movies>		

Tags with attributes, like <rating type="thumbs">, are converted into temporary variables as shown below. Each attribute item is named according to its name, in the name=value pair (attribute_name=value). In the example below, two temporary variables are created: one for the tag, and one for the attribute.

XML Item	Temporary Variable	Value
<rating type="thumbs">	xml.1.1.4.tag	rating
	xml.1.1.4.attr.type	thumbs

Tags with attributes, like <rating type="thumbs">, are converted into temporary variables as shown below. Each attribute item is named according to its name, in the name=value pair (attribute_name=value). In the example below, two temporary variables are created: one for the tag, and one for the attribute.

XML Item	Temporary Variable	Value
<plot>So, this language. It's like, a programming language. Or is it a scripting language? All is revealed in this thrilling horror spoof of a documentary</plot>	xml.1.1.3.value	So, this language. It's like, a programming language. Or is it a scripting language? All is revealed in this thrilling horror spoof of a documentary.

Below, is shown how the entire sample XML message is converted into temporary variables:

XML Item	Temporary Variable	Value
<movies>	xml.1.tag	movies
<movie>	xml.1.1.tag	movie
<title>PHP: Behind the Parser</title>	xml.1.1.1.tag xml.1.1.1.value	title PHP: Behind the Parser
<characters>	xml.1.1.2.tag	characters
<character>	xml.1.1.2.1.tag	character
<name>Ms. Coder</name>	xml.1.1.2.1.1.tag xml.1.1.2.1.1.value	name Ms. Coder
<actor>Onlivia Actora</actor>	xml.1.1.2.1.2.tag xml.1.1.2.1.2.value	actor Onlivia Actora
</character>		
<character>	xml.1.1.2.2.tag	character
<name>Mr. Coder</name>	xml.1.1.2.2.1.tag xml.1.1.2.2.1.value	name Mr. Coder
<actor>El ActÓr</actor>	xml.1.1.2.2.2.tag xml.1.1.2.2.2.value	actor El ActÓr
</character>		
</characters>		
<plot> So, this language. It's like, a programming language. Or is it a scripting language? All is	xml.1.1.3.tag xml.1.1.3.value	plot So, this language. It's like, a programming language. Or is it a

XML Item	Temporary Variable	Value
`revealed in this thrilling horror spoof of a documentary.`		`scripting language? All is revealed in this thrilling horror spoof of a documentary.`
`</plot>`		
`<rating type="thumbs">7</rating>`	`xml.1.1.4.tag` `xml.1.1.4.attr.type`	`rating` `thumbs`
`<rating type="stars">5</rating>`	`xml.1.1.5.tag` `xml.1.1.5.attr.type`	`rating` `stars`
`</movie>`		
`</movies>`		

SECS Protocol

This section discusses how to send and receive SECS messages.

The SECS protocol is used to send SECS messages to SECS compliant equipment.

To send or receive a message, the sc can act as a client, a server, or as both.

As a client, the SC connects to an established connection, sends a message, and optionally waits for a reply.

```
open socket_client name=tool proto=secs ...
```

As a server, the SC establishes the connection and waits for a message and may or may not reply.

```
open socket_server name=tool proto=secs ...
```

For example:

```
open socket_server name=tool proto=hsms logging=2 host=10.100.32.140 service=5001
options="t3=5 t6=10 t7=10 t8=10 timeout_msg=TIMEOUT"
```

In this example *host* is the IP address of the machine that sc is running and service represents the port that it will listen for communications on.

As a client, the `name` parameter is the destination for the message in the send command.

```
send name=tool ...
```

As a server, the `name` parameter is the label where the received message is handled.

```
open socket_server name=tool proto=secs …
```

```
label tool # Messages from the client come here
send_reply name=tool ...
```

In general, we always connect to SECS equipment as a client.

The connection can be a socket, a file socket, or a port.

The file socket would typically never be used, and the socket connection would typically be to a terminal server.

For a *socket connection*, specify the host machine and service name for the socket. This would typically be the terminal server and port where the SECS equipment is connected.

```
open ... name=tool proto=secs host=digi service=2100 ...
```

For a *file socket connection*, specify the local socket file name. Again, this is not typical.

```
open ... name=tool proto=secs local_name=sockets/s.recipe_server ...
```

For a port device file, specify the device file. Here the SECS equipment would be connected directly to the host machine.

```
open ... name=tool proto=secs device="/dev/tty0p0" ...
```

SECSI

Interleaving Messages

The SC does not support interleaving of messages. Interleaving is where you can send or receive more than one multi-block SECS message at a time. Interleaving means that the SECSI blocks of multiple messages can be mixed or interleaved with each other.

This is not the same as opened transactions, where the sc may send a primary message, but receive a primary message instead of the reply. The initial primary message is considered opened until either the reply is received, or a time-out is detected. The sc does support opened transactions. This means that the tool does not need to reply to a message immediately, but may reply later.

SECSI Options

The options for the SECS protocol are as follows:

SECSI Options	
baud=value	Defines the baud rate. Only valid when using device files. Valid values are: 300, 600, 1200, 2400, 4800, and 9600.
error=label	Defines a label to jump to if there is a protocol error.
flush=time	Defines a time period, in seconds, to wait and discard all incoming data. This is used when servers initially connect to a device, such as a terminal server, and want to ignore initialization messages or other data.
arrays=yes|no	Defines if tmp variables are created for each item in a secs array (arrays=yes), or one tmp variable is created for the array items (arrays=no).
rbit=value	The rbit (reverse bit) value is the direction bit. The value is either 0 (messages are being sent to the equipment) or 1 (messages are being sent to the host). When developing a driver, the value is always 0.
retry=value	The retry value specifies the number of retires upon a SECSI t2 timeout.
secsI_ logging=off	This turns off the SECSI logging of the SECSI protocol and block messages. The default is secsI_logging=on. **The feature is better controlled with the `logging` parameter used in the open statement. A value of 3 turns this on, a value of 2 or less, turns this off.**
t1=time	Defines the t1 time in seconds. T1 is the time period between receiving SECSI characters.
t2=time	Defines the t2 time in seconds. T2 is the time period between receiving SECSI protocol characters.
t3=time	Defines the t3 time in seconds. T3 is the time period between primary and secondary SECS messages.
t4=time	Defines the t4 time in seconds. T4 is the time period between SECSI message blocks.
timeout_msg=str	Defines a string, str, that will be returned if a time-out is detected. In the event of a time-out, the sc returns the str as if it was the actual reply message. This allows the sc program to test for time-outs when sending messages and waiting for replies.

Table 25: SECSI Options

SECSI Contention

This section discusses how the sc detects and handles SECS contention.

What is SECS contention? This is when both the sc and the equipment try to talk at the same time.

Who is the slave, and who is the master? In most configurations, the equipment is master. The sc is always the slave. This sc cannot be configured to be the master.

So the rule is: when contention is detected, the master proceeds, and the slave backs down.

The sc handles contention by first processing the event, and then resuming where it left off. The following shows the order in which the statements are executed when contention is detected at step 2.

```
statements…  # 1. Start here
send name=tool 0 W S1F1 <L>. # 2. Contention  # 6. Execute again
statements…   # 7. Execute these statements.
```

Note, at this point, all the `tmp` variables from above were overwritten.

```
label tool   # 3. Jump to here
statements…  # 4. Execute these statements
             # 5. Jump to 6
```

HSMS

The options for the SECS protocol are as follows:

HSMS Options	
error=label	Defines a label to jump to if there is a protocol error. Used for example by a client to detect if server terminates. This is different than a disconnect (see below).
flush=time	Defines a time period, in seconds, to wait and discard all incoming data. This is used when servers initially connect to a device, such as a terminal server, and want to ignore initialization messages.
hsms_ logging=off	This turns off the HSMS logging of the HSMS protocol. The default is hsms_logging=on.
t3=time	Defines the t3 time in seconds. T3 is the time period between a primary and secondary secs messages. A typical value is 30.
t5 = time	Defines the t5 time in seconds. T5 is the time period between a failed "linktestrequest" and a subsequent retry. A typical value is 10.
t6=time	Defines the t6 time in seconds. T6 is the time period between a "linktest request" and linktest response". A typical value is 10.

HSMS Options	
t7=time	Defines the t7 time in seconds. T7 is the time period between a TCP/IP accept and "select request". A typical value is 10.
t8=time	Define the t8 time in seconds. T8 is the time period between data bytes. A typical value is 10.
timeout_msg=str	Defines a string, str, that will be returned if a time-out is detected. In the event of a time- out, the sc returns the str as if it was the actual reply message. This allows the sc program to test for time-outs when sending messages and waiting for replies.
disconnect=label	Defines a label where sc will jump if a disconnect (closed connection) is detected. This applies to the client side of the connection only. A disconnect is detected when the server side of the connection closes the connection. However the server would still be running thus allowing for the client to re-connect. This does not apply if the server issues an hsms separate request. When this condition is detected, sc will resume execution at the specified `label`.

Table 26: HSMS Options

Annotated SECSII Messages

This section discusses the format of the SECSII message.

The annotated SECSII message has the following format:

Sstream**F**function device_id [W] body .

The stream and function are the SECSII stream and function. The device_id is the SECSII device id. The 'W' is the optional wait-for-reply flag. The body is the SECSII message body that contains all the data types and their values. The message is terminated with a '.' period.

The body is defined in the format typically shown in most equipment's SECS manual. It uses the following abbreviations to define the SECS data types:

Data Type	Description	Comments
L	List	
A	ASCII	enclosed in double(") or single(') quotes

Data Type	Description	Comments
B	Binary	00 to FF, entered in Hexadecimal
BO	Boolean	T or F
U1, U2, U4, U8	Unsigned one, two, four, or eight byte integer(s)	
I1, I2, I4, I8	Signed one, two, four, or eight byte integer(s)	
F4, F8	Four or eight bytes floating point numbers	Can be entered in hex format. Ex: '\xhhh..hh' (Use single quotes)

Table 27: SECS Data Types

Angle brackets ,<>, are used to delimited the data types and their values, i.e. `<data_type value>`.

A typical S2f41 SECS message follows:

```
S2F41 0 W
<L
  <A "START">
  <L>
>.
```

Arrays can be specified by providing additional data values.

```
<U2 100 200 300>
```

Arrays can also be entered using the value of a variable, where the variable contains the items.

```
let data = "100 200 300"
<U2 $data>
```

Binary values can be entered with or without space delimiters. However, they must be entered as 2 hex characters for each binary value.

```
<B 010203FEFF>
<B 01 02 03 FE FF>
```

Variables in SECSII Replies and Unsolicited Messages

This section discusses how to handle the SECSII replies and events messages.

Whenever the sc receives a SECSII reply message or an unsolicited message, the sc converts that message in temporary variables. These variables are accessed via the get_tmp_var () function.

Some common SECSII variables include:

Temporary Variable	Description
`DEVICE_ID`	SECS message device id
`WAIT_BIT`	SECS message wait bit
`SXFY`	Stream and function of the received SECS message
`HEADER`	The secs message header (1)
`BODY`	The secs message body (1)
`SYSTEM_BYTES`	System bytes for the header (1)

Table 28: Common SECSII Variables

(1) Requires the `pragma MORE_SECS_TMP_VARIABLES` to be enabled.

The remains of the message is converted into temporary variables. Each data item is named according to its position within the list. List items are assigned to the number of elements in the list.

SECS Item	Temporary Variable	Value
`S6F11`		
`<L`	`S6F11.1`	`3`
`  <U4 100>`	`S6F11.1.1`	`100`
`  <U4 200>`	`S6F11.1.2`	`200`
`  <L`	`S6F11.1.3`	`2`
`    <U4 200>`	`S6F11.1.3.1`	`200`
`    <L`	`S6F11.1.3.2`	`3`
`      <U2 100>`	`S6F11.1.3.2.1`	`100`
`      <U2 200>`	`S6F11.1.3.2.2`	`200`

Arrays are handled in one of two ways. This is controlled via the "arrays=yes|no" options parameter, with the default being "arrays=yes".

For "`arrays=yes`", the temporary variable for the item is assigned to a value "`ARRAY`". Other temporary variables, index as [1], [2], [3], etc., are assigned to each array element. Index [0] is assigned to the number of array elements. Binary elements, in addition to indexes [1], [2],. [3], etc, contain an index, [], that contains the entire binary array.

SECS Item	Temporary Variable	Value
S6F11		
<L	S6F11.1	
<U4 11 12 13 14 15>	S6F11.1.1	
	S6F11.1.1[0]	5
	S6F11.1.1[1]	11
	S6F11.1.1[2]	12
	S6F11.1.1[3]	13
	S6F11.1.1[4]	14
	S6F11.1.1[5]	15

Table 29: arrays=yes Example

For "arrays=no", the single temporary variable contains all the array elements.

SECS Item	Temporary Variable	Value
S6F11		
<L	S6F11.1	1
<U4 11 12 13 14 15>	S6F11.1.1	11 12 13 14 15

Table 30: arrays=no Example

For SECS reply messages, these calls to `get_tmp_var ()` belong immediately following the send statement.

```
send name=name S2F41 0 W
<L
  <A "START">
  <L>
>.
# Handle S2F42 reply here
```

For unsolicited SECS messages, these calls belong following the label statement for the opened connection. All unsolicited messages from that connection are handled at that label.

```
open name=tool ...
# All unsolicited messages from tool come here
label tool
# Process the SECS message
send_reply name=tool ...
```

```
break
```

It is best to have the `send_reply` as the last command. This is because the reply may encounter contention causing the sc to handle the contentious message first. When the sc returns, the temporary variables would have been over written.

For/Next Loops

When creating SECS messages, there are times when the number of elements may vary. To handle this, the annotated SECSII message allows the use of for/next constructs. This is best shown by an example:

In the following S2F41, the number of optional CPNAME/VALUE pairs is constructed by the for/next loop.

```
S2F41 0 W
<L
  <A PP_SELECT>
  <L
    <L
      <A NUM_WAFERS>
      <U2 $num_wafers>
    >
    for i=1 to $num
    <L

      <A $cpname[$i]
      <A $cpvalue[$i]
    >
    next I
  >
>.
```

The number of CPNAME/VALUE pairs is determined by $num. The $num, $cpname[], and $pcvalue[]

variables would have been previously assigned.

In this example, we send the `S2F37` enable events message. The list of ceids are defined in a ceid [] array, and num_ceid is the max number of ceids.

```
S2F37 0 W
<L
  <BO TRUE>
  <L for i=1 to $num_ceid
    <U4 $ceid[$i]>
  next I
  >
>.
```

Handling SECS Events

This section discusses how to handle the SECS event messages from the program file.

From before, the sc jumps to the label statement associated with the opened connection. However, statements must be added at that label to properly handle the event.

First, test to determine if a SECS reply is required. This is done by testing the wait bit if the incoming SECS message. Use the `get_tmp_var ()` function. If the wait bit is set, a reply is expected.

```
if (get_tmp_var (WAIT_BIT) == 1)
  # A reply is expected
else
  # No reply expected
end_if
```

It is suggested not to have this code in the actual driver, but simply reply or not by convention.

Second, determine what the reply should be. Use the `get_tmp_var ()` function to get the stream and function of the SECS message.

```
if (get_tmp_var (SXFY) eq "S1F1")
  # Reply with a s1f2
else_if (get_tmp_var (SXFY) eq "S6F11")
  # Reply with a s6f12
else_if (get_tmp_var (SXFY) eq "S5F1")
  # Reply with a s5f2
end_if
```

If you don't know the SECS device id of the incoming message, use the `get_tmp_var ()` function to get it. Then the device id can be used in the subsequent send_reply statement.

```
let device_id = get_tmp_var (DEVICE_ID)
send_reply $device_id ......
```

Obviously, other handling is required once an event is received, such as handling the different types of equipment S6F11 SECS events.

Dynamic SECS Body

In the previous sections, the SECS body is constructed in the secs message. However, is it possible to

create a variable with the appropriate SECS message and then send it in the secs message body. You cannot mix the data variable with other SECS elements in the message body. The entire SECS message needs to be included in the variable, except the ending ".". Include the "." with the send statement.

```
send name=tool S1F1 0 $data .
```

The following two S1F2 reply messages are equivalent:

Example 1	Example 2
`label S1F2`	`label S1F2`
`send_reply name=tool S1F2 0`	`let data = '<L <A "demo"> <A "demo"> >'`
`<L`	`send_reply name=tool S1F2 0 $data .`
`  <A "demo">`	
`  <A "demo">`	
`>.`	

SECS Pass-Through

There is another format of send command which allows the received secs message to be passed-through with or without any interpretation. When the MORE_SECS_TMP_VARIABLES pragma is enabled, the .HEADER and .BODY tmp variables contain the received secs message. These can be used to build a new send message to be sent to a host or client.

```
label S1F1
let header = get_tmp_var(S1F1.HEADER)
let body = get_tmp_var(S1F1.BODY)
send name=<name> $header $body # or send name=<name> client=<client> $header $body
```

You can use just the header portion and add you own body.

```
send name=<name> $header
<L
  <A “xxxx”>
  <A “yyyy”>
>.
```

The follow is an example of a program that would reside between an host and the equipment. Each message is simply received from one connection and passed to the other.

```
pragma MORE_SECS_TMP_VARIABLES
# Assumes all the messages have the Wait Bit set
# Connecton for the host (host will connect to here)
open socket_server name=pass_through proto=hsms logging=2 host=127.0.0.1
service=5000 options="t3=120 t6=10 t7=10 t8=10 timeout_msg=TIMEOUT"


# Connection to the tool
open socket_client name=tool proto=hsms logging=2 host=127.0.0.1 service=5001
options="t3=120 t6=10 t7=10 t8=10 timeout_msg=TIMEOUT"


# ------------------------------------------------------------------------


label pass_through # Messages from the host
    let client = get_client(pass_through)
```

```
    let sxfy = get_tmp_var(SXFY)
    let header = get_tmp_var($sxfy . ".HEADER")
    let body = get_tmp_var($sxfy . ".BODY")

    # Send the message to the tool
    send name=tool $header $body .

    let sxfy = get_tmp_var(SXFY)
    let header = get_tmp_var($sxfy . ".HEADER")
    let body = get_tmp_var($sxfy . ".BODY")

    # Send the reply back to the host
    send_reply name=pass_through $header $body .

# ------------------------------------------------------------------------

label tool # Messages from the tool
    let sxfy = get_tmp_var(SXFY)
    let header = get_tmp_var($sxfy . ".HEADER")
    let body = get_tmp_var($sxfy . ".BODY")

    # Sent the message to the host
    send name=pass_through client=$client $header $body .

    let sxfy = get_tmp_var(SXFY)
    let header = get_tmp_var($sxfy . ".HEADER")
    let body = get_tmp_var($sxfy . ".BODY")

    # Send the reply back to the tool
    send_reply name=pass_through $header $body .
```

Hints

You can check the version of sc using the Unix "what" or "strings" commands.

```
what sc
strings sc | grep '@(#)'
```

Examples

Using sc as a semiconductor equipment simulator

The following bit of sc code represent a very simplistic equipment simulation. Please refer to the callouts for a more detailed explanation of what is going on.

1. "*host_connection*" is the name of this connection. In environments where multiple connections are required it is used to distinguish between them. The value of name (in this case "*host_connection*") is also the value of a `label` in the program file. Whenever there is input on this channel sc will start processing at the statement immediately following this `label`.
2. The value of `host` can be either an IP address or it can be a host name that can be resolved via the host file or DNS. Since this script example is that of a server (think equipment simulator) the IP address would be the same as the machine that sc is running on. `localhost` (127.0.0.1) can be used instead.
3. The value of `service` represents the port that the connection will use. The value of 5000 is fairly typical.
4. Whenever a message is received on the "`host_connection`" channel/connection sc packages the contents of the message into temporary variables. When using the SECS (`secs` or `hsms`) protocols there are several temporary variables you can expect to have good data in them. In this case the `SXFY` variable will contain the stream and function of the message just received. For example, if the incoming message was a stream 1 function 13 message the value of the temporary variable `SXFY` would be S1F13.
5. Most semiconductor equipment simulators will be set up in this manner. In simple terms, whenever a message is received the stream and function of the message are retrieved and this dynamic `goto` is executed to direct the simulation to the appropriate message handler. In the situation mentioned above, this line would resolve to a `goto S1F13` which would cause the program to start executing at the line immediately following `label S1F13`.
6. This is a very simple handler to deal with an incoming S1F13 message. It generates the appropriate S1F14 message header and then includes the information specified and sends the reply back on the "`host_connection`" connection. In this example the values of the variables `E5Model` and `E5SoftwareVersion` are retrieved and sent back as part of the response.
7. In this example, this block of code is executed in response to an interactive command line input from the user. This code simulates the start and ending of a job that lasts 10 seconds. In order to execute this code block the user would type `event run_job` or `e run_job` from the command line and press enter. As a result the two S6F11 messages would be sent with a 10 second pause between them. One thing to note in this example is that sc will be asleep for 10 seconds and will not be able to process other messages, an incoming S1F13 for instance,

until it wakes up from its sleep and starts processing again. There are methods to deal with this situation in the event this is not acceptable. They are explained elsewhere.

```
let client = -1
let E5Model = "EqModel"
let E5SoftwareVersion = "1.2.14"
let sc_version = get_version()
print "SC Version is <" . $sc_version . ">"

open socket_server name=host_connection 1 proto=hsms logging=2

host=10.100.32.140 2 service=5000 3 options="t3=5 t6=10 t7=10 t8=10
timeout_msg=TIMEOUT"

# All communications from the host start here

label host_connection 1
    let client = get_client(host_connection)

    let sxfy= get_tmp_var(SXFY) 4

    goto $sxfy 5

# S1F1 - Hello from the host
label S1F1
    send_reply name=host_connection S1F2 0
    <L
        <A $E5Model>
        <A $E5SoftwareVersion>
    >.

# S1F13 - Establish Communications, CommunicationState is
COMMUNICATING

label S1F13 6
    send_reply name=host_connection S1F14 0
    <L
        <B 00>
        <L
            <A $E5Model>
```

```
            <A $E5SoftwareVersion>
        >
    >.

# S7F19 - Request PPID List
label S7F19
    send_reply name=host_connection S7F20 0
    <L
        <A '/Test/Recipe001'>
        <A '/Test2/Recipe002'>
        <A 'SZ41T_8a76A'>
        <A 'EMPTY RECIPE'>
    >.


label run_job 7
    # send job started event
    send name=host_connection client=$client S6F11 0 W
    <L
        <U4 23>
        <U4 5458>
        <L
            <L
                <U4 106>
                <L
                    <A 'CJ001'>
                >
            >
        >
    >.

    sleep 10
    # send job completed event
    send name=host_connection client=$client S6F11 0 W
    <L
        <U4 24>
        <U4 5459>
        <L
            <L
                <U4 106>
                <L
                    <A 'CJ001'>
                >
            >
```

```
        >
    >.
```

Using sc as a semiconductor host or Equipment Interface simulator

The following bit of sc code represent a very simplistic equipment simulation. Please refer to the callouts for a more detailed explanation of what is going on.

1. "`tool_connection`" is the name of this connection. In environments where multiple connections are required it is used to distinguish between them. The value of name (in this case "`tool_connection`") is also the value of a label in the script file. Whenever there is input on this channel sc will start the program running at the statement immediately following this label.
2. The value of `host` can be either an IP address or it can be a host name that can be resolved via the host file or DNS. Since this script example is that of a client (think Equipment Interface or host controller simulator) the IP address would be that of the equipment (or emulator/simulator) that you want to communicate with.
3. The value of `service` represents the port that the connection will use. The value of 5000 is fairly typical.
4. Whenever a message is received on the "`tool_connection`" channel/connection sc packages the contents of the message into temp variables. When using the SECS (secs or `hsms`) protocols there are several temporary variable syou can expect to have good data in them. In this case the `SXFY` variable will contain the stream and function of the message just received. For example, if the incoming message was a stream 1 function 13 message the value of the temp variable `SXFY` would be `S1F13`.
5. Most semiconductor host controllers or Equipment Interface simulators will be set up in this manner. In simple terms, whenever a message is received the stream and function of the message are retrieved and this dynamic `goto` is executed to direct the simulation to the appropriate message handler. In the situation mentioned above, this line would resolve to a `goto S1F13` which would cause the script to start executing at the line immediately following `label S1F13`.
6. This is a very simple handler to deal with an incoming S1F13 message. It generates the appropriate S1F14 message header and then sends the reply back on the "`tool_connection`" connection.
7. These are example of some of the commands you might find in a file that was simulating and Equipment Interface or host.

```
open socket_client name=tool_connection 1 proto=hsms logging=2

host=10.100.32.140 2 service=5000 3 options="t3=5 t6=10 t7=10 t8=10
timeout_msg=TIMEOUT"


label tool_connection 1

    let sxfy = get_tmp_var(SXFY) 4

    goto $sxfy 5

/*
S1F13 - Establish Communications, CommunicationState is COMMUNICATING
This is what the response to an S1F13 is when returned from a "HOST"
*/


label S1F13 6
    send_reply name=tool_connection S1F14 0
    <L
        <B 00>
        <L
        >
    >.


label est_com 7
    send name=tool_connection S1F13 0 W
    <L>.


label hello 7
    send name=tool_connection S1F1 0 W
    <L>.


label query 7
    send name=tool_connection S1F3 0 W
    <L
FIXME
    >.
```

```
label offline 7
    send name=tool_connection S1F15 0 W .

label online 7
    send name=tool_connection S1F17 0 W .
```

Using sc to upload a binary recipe and compare it to a recipe in a file

This example shows how to upload a binary recipe from the tool and then compare it to a recipe file.

```
# Connect to the tool
open socket_client name=tool proto=secs local_name="./sockets/tool"
logging=2 options="baud=9600 t1=2 t2=2 t3=45 rbit=0 retry=3
timeout_msg=T3_TIMEOUT_DETECTED"
# Upload the recipe from the tool
send name=tool S7F5 0 W
<L
    <A "my_recipe">
>.

# Add all the uploaded bytes together to create one variable with the data
let size = get_tmp_var("S7F6.1.2[0]")
print "size <" . $size . ">"
let recipe = "" # Initialize to blank
for i=1 to $size
    let b = get_tmp_var(S7F6.1.2[$i])
    let bb = itoh($b, "2.2") # Convert the decimal value to hexadecimal
    print "b <" . $b . "> bb <" . $bb . ">"
    let recipe = $recipe . $bb
next I

# Read in the master recipe file (from disk)
import("t.binary_recipe_small", "rb", recipe.old)

# Compare the recipe file with the uploaded recipe
if ($recipe.old eq $recipe)
    print "YES match"
else
    print "NO match"
end_if
```

More sophisticated variable usage

This section will be addressed in a later revision.

Maintenance

This section discusses how to maintain the sc source code.

Debugging sc

All the source code contains DEBUG statements. Debug statements can be added anywhere in the source code. The format of a debug statement is as follows:

```
DEBUG (<level>, DEBUG_HDR, <format>, <args>);
```

<level> - Can be any number, and is controlled by the –d <level> option on the command line.
<format> - Defines the constant portion.
<args> - Defines the arguments.

Each module should also contain:

```
#undef NAME
#define NAME "<function_name)()"
```

Example:

```
static void look(struct buffer *token, struct buffer *name, int *idx1, int *idx2)
{
    char *data = (char *)NULL;
    int i = 0;
    int var = 0;
    int nest = 0;
    int nest_var = 0;

#undef NAME
#define NAME "look()"

    DEBUG (3, DEBUG_HDR, "Look called, token <%s>\n", bufdata(token));
    var = nest = nest_var = 0;
    data = bufdata(token);
    bufcpy(name, "");
    *idx1 = *idx2 = 0;
```

```
.............. deleted code ..............

    DEBUG (3, DEBUG_HDR, "Returning name <%s>, idx1 <%d>, idx2 <%d>\n",
    bufdata(name), *idx1, *idx2);

    return;
}
```

Debugging can also be controlled by an sc.debug file. This file contains enteries containing filenames and modules. If the entry is present, debug statements will be printed. If an entry is ommitted, commented-out ("#"), debug statements will not be printed.

```
break.c::cmd_break_loop()
buffer.c::buffree()
buffer.c::bufprefix()
buffer.c::bufncat()
buffer.c::bufcat()
buffer.c::bufcpy()
buffer.c::bufncpy()
buffer.c::bufnlcpy()
buffer.c::buffilecpyread()
.........
```

The sc.debug file can easily be created using the following script:

```
strings sc.exe | grep ".c::" > sc.debug
```

The sc.debug file and the –d <level> command line option work together. The level is used to further specify which debug statements are printed.

The sc.debug file is optional and my be omitted, thus only the –d <level> will be used to control the printing of debug statements.

Important Concepts You Need to Understand

Arrays

There are no arrays in sc. There is only an emulation of arrays. Consider the following program snippet:

```
let day[1] = "Monday"
let day[2] = "Tuesday"
let day[3] = "Wednesday"
```

The result of executing the above 3 statements is that 3 variables have been created and have been assigned values. (If they already existed, the values of them would be assigned as indicated above.) The variables created would have the names `day[1]`, `day[2]`, `day[3]`. Yes, that is correct, the names of the variables actually have the left square bracket "`[`", the number ("`1`" for example), and the right square bracket "`]`" as part of their names. Let's say we want to print out the values of the variables. You might do it like this:

```
for i = 1 to 3
    print $day[$i]
next i
```

This would result in `Monday`, `Tuesday`, and `Wednesday` being printed out each on their own line. Basically what is happening is that the `$i` is being converted to its value, let's say `1` in this case. Now the `print` statement looks like `print $day[1]`. This now translates into print the value of the variable with the name of `day[1]`. The `$` (dollar) sign tells sc to substitute the variable's value.

Oh, this is so hokey and silly. Not so fast. How about a contrived, but, not too far fetched example. Let's say we are communicating with a machine and it sends an unsigned 16-bit value that conveys important information that the program needs to act on. The machine can send one of 65536 values (0 – 65535). However, through analysis we know that there are only a few values that it will actually send. Here is an example of a way to implement the program.

```
let jump_table[0] = "Turn_on_Green_Light"
let jump_table[32767] = "Turn_on_Blue_Light"
let jump_table[32768] = "Turn_on_Yellow_Light"
let jump_table[65533] = "Turn_on_Red_Light"
let jump_table[65534] = "Help_Me_Mommy"
let jump_table[65535] = "Kill the Witnesses"
```

code to receive message from machine

```
goto $jump_table[$value_from_machine]

label  Turn_on_Green_Light
```

code to turn on the green light

```
label  Turn_on_Blue_Light
```

code to turn on the blue light

```
label  Turn_on_Yellow_Light
```
code to turn on the yellow light

```
label  Turn_on_Red_Light
```
code to turn on the red light

```
label  Help_Me_Mommy
```
code to … its getting bad

```
label "Kill the Witnesses"
```
use your imagination here

As described above, in the `goto` statement this is what happens. The value of the variable `value_from_machine` (let's say it is 32767) is retrieved thus changing the `goto` statement into:
`goto $jump_table[32767]`

Next the value of the variable `jump_table[32767]` is retrieved changing the `goto` statement into:
`goto  Turn_on_Blue_Light`

At this point the program flow jumps to the label `Turn_on_Blue_Light` and begins executing the statement after the `label`. (As explained in the Labels section below the program will stop executing instructions when it encounter the next label statement, in this case `label Turn_on_Yellow_Light`. The will (normally) go back up and await the next message from the machine.)
So, what did we do? By using an "array" we were able to create a vector table that was able to use the datum from the machine in order to directly access the appropriate response. We did not have to create a "normal" array that would have needed the space for 65536 entries. We only needed 6 "entries."

Since the array concept is only emulated in sc the following is totally okay and does not produce some kind of array boundary issues:

```
let array[12345678909876543321] = 1
```

Consider also that the array "index" does not have to be a number. This is possibly too:

```
let tasks[Monday] = "Water Plants"
let tasks[Tuesday] = "Wash Clothes"
let day = "Monday"
```

```
print "Today's task is " .  $tasks[$day]
```

This would print out "`Today's task is Water Plants`".

One last item of note is yes, a quoted label may contain spaces as in the label statement:

```
label "Kill the Witnesses"
```

Labels

Labels are named locations in a program that may be reached via some statement that can redirect control flow, a `goto` statement for instance. Labels in an sc program are probably different than you may be used to. In an sc program a label that is reached without an explicit directive to go to it will cause the execution of the sc program to stop. In the normal case, the program does not "die" it just returns to waiting for incoming messages, expired timers, etc. Below is an example, the text in **bold** identifies the program statements that are actually executed:

```
let a = 1
goto add_2_to_a

let b = 2

label add_2_to_a
    a = $a + 2

label add_2_to_b
    b = $b + 2
```

Multi-threading

Sleep(ing)

Timers

sc has 6 timers. The ID for the timers are the numbers 0, 1, 2, 3, 4, 5. A timer expiration will not interrupt the current processing. This means if sc is caught in a "compute bound" loop it might be a while before expired timer handlers are activated. When sc reaches a "quiet" state (usually awaiting a response "message" or an unsolicited "message") it will process any timers that have expired. The "compute bound" condition is pretty rare do to sc being pretty fast, but, it is something to keep in mind if delayed timeout handling is causing issues in your application. As mentioned above, if the `NO_BLOCK_ON_SLEEP pragma` is not used a `sleep` statement will cause sc to not handle any timeout until sc "awakens" from the sleep statement.

License

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you

distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest

your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for

making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying

the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the

integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of

this License, you may choose any version ever published by the Free Software

Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER

PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License

```
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
    USA


Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) year name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show
w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than `show w' and `show c'; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

   Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  `Gnomovision' (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.