Edge Integration

Station Controller (sc)

User's Guide

May 14, 2009
Version 2.5

Table of Contents

Revision Notes

Versions

Version 2.5 (reflected in sc 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.

Version 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. Hit 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 -h* 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 preprocessor directives are used in "my_program"

```
$ 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 -c "read my_program" $ sc -c "read \"./src/my_program\""
-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. $ -d 3 -c "read my_program"
-h	Used to show the options $ sc -h
-i	Run 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.
-s *service*	Used to specify an interface file socket, *service*, when starting the sc. This socket is used to communicate with the running sc. The 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" -s sockets/s.sc_tool
-v	Used to get the current version of the sc. $ sc -v
-x *program*	Used to run the *program* syntax checker. $ sc –x program
-I *path*	Define a include *path* to search for include files. See (#include)
-D *name*	Define a name in a program (see #ifdef).
-D *name*=*value*	Substitute all occurrences of *name* with *value* in a program. $ sc –c "read –D DEBUG –D TOOL=Nova program"
-P	Enable the preprocessor. By default, the preprocessor is enabled if the –I or –D options are used. $ sc –c "read –P program"

sc Commands

In this section we discuss the sc commands.

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

sc Command	Description
help	Used to get a list of all the commands. sc> help

debug *level*	Used to set the debugging level. The *level* ranges from 0 (turns debugging off) to 5. sc> debug 3
dump dump *variable*	Used to dump variables and other internal information. Any *variable* or partial *variable* is dumped. The results are written to a file called "dump.out". sc> dump
event *label* event *label arguments*	Used to jump to an label. The *label* is defined in the program. Any *arguments* are converted into temporary variables. Jumps to the label start: sc> event start Jumps to the label start and assigns the temporary variable recipe to bake_me: sc> event start recipe=bake_me
kill	Used to terminate the program . sc> kill
list	Used to dump a listing of the loaded program. The results are written to a file called program.list. sc> list
logging logging *name value*	Used to view or change the current logging. With no parameters, the current level of logging is shown. Otherwise, the logging level of the *name* is modified. The *name* value is from the "name=*name*" used in the open statement. A *value* of 0 turns of logging. The higher the *value*, the more verbose the logging. (see the open statement) sc> logging sc> logging equipment 3
ping	Used to ping the to make sure it is running. sc> ping
read *program* load *program*	Used to read, load, and start executing the program file. The *read* and *load* commands are the same. sc> read my_program_file
set *name value*	Used to create or assign the global program variable *name* to *value*. Equivalent to "glet *name*=*value*" sc> set reset_flag 1
tokens	Used to dump the tokenized program file. The results are written to a file called program.tokens. sc> tokens
version	Used to get the current version of the . sc> version

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.

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

begin
   statements
end
```

However, if we uncomment 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 ARGx, 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 and- ing and or- ing 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[1234567890123456789] 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 know 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*'s 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 tmp 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 tmp 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 a 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 whle. 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	Description
after *label*	Defines a label. Same as a label. after loop Is the same as label loop

	Note you can include special chars like 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. begin main end

break	Used to halt execution.
	
	If ($return eq "error")
	print "Error detected"
	break # Stop, don't continue
	else
	end_if

break_loop	Used to exit a for or while loop. for I=1 to 10 If ($I == 5) break_loop # Exit loop, and continue end_if next i
close name=*name* close client=*client*	Used to close a connection. The *name* is the name that was used in the open statement. open socket_client name=equipment… close name=equipment As a server, you can also close a connected *client*. *Client* is the fd of the connected client. close client=$client
continue	Used to continue execution within a for or while loop for i=1 to 10 if ($x[$i] eq "") continue end_if next I
define *var* define *var* = *expr*	Used to define a program variable and optionally assign it to a value. Once a variable has been defined, it cannot be modified. define x define x = 100 define x = 5 * 100
define_array *var* = *expr1*, *expr2*, …	Used as a convience method to create an emulation of an array. The name of the array is *var*. Its values are *expr1*, *expr2*, etc. The array *var* is indexed as follows: *var*[*x*], where x starts at 1, and continues for each *expr*. The variable *var[0]* contains the number of elements in the array. define_array days = Mon, Tue, Wed, Thur, Fri, Sat, Sun The variables below are assigned: $days[0] is 7 $days[1] is Mon $days[2] is Tue $days[3] is Wed $days[4] is Thur $days[5] is Fri $days[6] is Sat $days[7] is Sun

define_xfer *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: $color[red] is FF0000 $color[FF0000] is red $color[white] is FFFFFF $color[FFFFFF] is white $color[blue] is 0000FF $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] and produce the much more user friendly value of red.
delete "*var*" delete "*var**"	Used to delete program variables and release any associated storage. The astrisk may only appear at the end. delete "x" delete "data[LOT1]*"
end	Used to define the end of a subprogram. begin main end
exit	Terminates the sc and stops it from running. Equivalent to the 'C' exit (0).
for-next for *var*=*expr* to *expr* next *var* for *var*=*expr* to *expr* by *expr* next *var*	Used for looping. Also supports the continue and break commands. for x=1 to 10 next x for x=1 to $max by 2 next x Note that the *next* uses the variable, x, not its value, $x
glet *var* = *expr*	Used to assign a global variable to an expression. A global variable is a variable created outside any subprogram let x = 100 Assigns the global variable x to 100 begin alternate_main # Re-assign the global variable x 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 alternate_main, 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 begin-end, then outside the context of the begin-end
gosub *label* gosub *label arg1*, *arg2*, ...	Used to jump to a label. A return is expected. gosub sub func_1 label 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 result is passed by reference (the variable *result* must exist prior to the *gosub*): let result = 0 # Initialize the result let arg1 = 100 let arg2 = 200 gosub add $arg1, $arg2, result
goto *label*	Used to jump to a label. No return is expected. goto loop label loop

if-end_if
if (*expr*)
end_if
 	
if (*expr*)
else_if (*expr*)
else
end_ if
	Used for conditional flow
	
	if ($x == 100)
	end_if
	
	if (($x == 100) or ($y == 200))
	else_if ($z == 300)
	else
	end_if

label *label* label *label arg1*, *arg2*, ...	Used to define a label within a program or subprogram. The *label* can be a quoted string. label loop label "jump here" If the *label* is used by a *gosub*, or *goto*, arguments may be passed. gosub add $arg1, $arg2, result # *result* will contain the sum of *arg1* and *arg2* # Assign c to a + b label add a, b, c let c = $a + $b print "The sum is " $c 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*.
let *variable* = *expr*	Used to assign a variable to an expression. If the variable does not exist, it is created. let x = 100
local_var *var* local_var *var* = *expr*	Defines a variable local to a label. It only makes sense to use this within a label. You can optionally assign the variable to *expr*. By default, the variable is initialized to blank. In the example below, the variable *i* inside the label and the variable *i* 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 …
open *type* name=*name* proto=*proto* and device=*device* or local_name=*local_name* or host=*host* service=*service*	Open a connection for sending and receiving messages. The name is used to identify the opened connection. It also corresponds to a label in the file where control flow will resume whenever a message is received on this connection. The types (*type*) include:

and logfile=*logfile* and logging=*logging* and options=*options*	pipe – Unix named pipe port - Serial port on the host socket_client - Connect to a socket (typical for host simulator uses) socket_server - Listening socket (typical for equipment simulator uses) The proto's (*proto*) include: commands – sc commands standard – Simple text messages SECS – Semiconductor SECS protocol HSMS – Semiconductor HSMS protocol The *logfile* value defines a log file for this connection. All logged messages are written to this file. The (*logging*) value defines the level of protocol logging. Values include: 0 – Turns off logging 1 – One line message summary 2 – Complete message (default) 3 – Protocol details The *device* is a system file device such as "/dev/tty01" or Unix named pipe. The *local_name* is a file socket. The *host* and *service* are the remote host and port names. *host* can be an IP address or it can be a host name. If it is a host name it cannot have any special characters in it. Dashes would be an example. If there are special characters you must bound the value of host with "". The *options* are protocol options for the connection. The value of *options* may be a string or a concatenation of strings (expression).
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 chars as {hh}. This applies to the *print* statement and *dump* command. By default, non-printing chars 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 continues 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 tmp variables from received secs messages.
print *string*	Used to print text. String concatenation can be used to construct the string. print "This is a test"
Return	Used to return from a previous gosub.
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

send name=*name* options=*options message* send name=*name* client=*client* options=*options message*	specific. send name=equipment message The options defined in the open command can be temporarily overridden with the values defined by *options* 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
set_program *name*	Used to activate a subprogram. The subprogram *name* becomes the active program. set_program main Note: Use set_program default to return to the default program.
sleep *seconds*	Used to sleep for a number of seconds. The sleep is an inline sleep. All sc processing is suspended. 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.
start_timer timer=*timer_id* seconds=*seconds* handler=*label* msg=*msg*	Used to start a timer. There can be up to 6 timers running. The *timer_id* (0-5) identifies the timer. The *seconds* are the number of seconds to wait, and the *label* is where execution will start when the timer expires. The *msg* is converted to program tmp variables when the timer times-out. start_timer timer=1 seconds=5 handler=loop label loop # Jump here when timer 1 times-out
stop_timer timer=*timer_id*	Used to stop a timer. The *timer_id* identifies the timer to stop. No error is produced a timer that is not running is stopped. stop_timer timer=1

while (*expr*)
end_while
	While loop. Executes commands within the while and end_while if *expr* is true (non zero). Also supports the continue and break commands.
	
	i = 1
	while (1)
	if ($x[$i] eq "")
	let i = $i+ 1
	continue
	else
	break
	end_if
	end_while

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 sc command line options: –P, -D, or –I must be included.

Directive	Description
#define *name* #define *name=value*	Define a name within a program. #define NAME=John print "My name is " . NAME For the #define *name* without a *value*, see #ifdef below
#include *filename*	Include another program into the current program. #include defs.h The location of defs.h can be defined by the -I sc command line option.
#ifdef *name* #endif #ifdef *name* #else #endif	Define a conditional for the program #ifdef DEBUG print "Debug….. #endif The *name* can be defined by either a #define, or the -D command line option.

Comments

Comments are allowed by the methods below.

Comments	Description
#	Single line
/*… */	Block

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)

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	== !=	Equal and not equal
5	&	Bitwise AND
6	|	Bitwise OR
7	^	Bitwise exclusive OR
8	and	Logical AND
9	or	Logical or

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

Unary Operators

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

Math operator	Description
+	Plus
-	Minus
~	One's Complement

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

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

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

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

Constants

Valid constants for variables

Constants and Examples
Whole numbers: 100, 500, -6 Floating point numbers: 1.2, 5.10 Exponential numbers: -6.12e+12, 5.6E-3 Strings: "This is a string" String characters: \xhh – Hex value 00->FF \n – New line \r – Carriage return

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 name to "John Smith"
let name = "John" : "Smith"           # Assigns name to "John Smith"
```


Keywords

The following is a list of keywords that are reserved and should not be used in the sc program. 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	Itoi	return	to
define_array	export	Label	send	while
define_xfer	for	Let	send_reply	
delete		local_var		

Functions

The following functions can be used in any expression:

Function	Description
abs(*value*)	Absolute value. Returns the absolute value for *value*. (Build 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. 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 ($fd)
file_open(*file*, *mode*)	Opens the file, *file*, with the *mode*, and returns a file handle. See the table of valid modes below. let fd = file_open ("data", "r") # Open the file "data" as read only, and assigns fd to the file handle
file_orc(*file*, *mode*) file_orc(*file*, *mode*, *string*)	Opens the *file* and reads and returns the desired line. *Mode* specifies the line to read, 1 to x. If *mode* is –1, then the last line is returned. If the line can't be found, a EOF is returned If a *string* is specified, the line containing the *string* is returned. If *mode* is 1, then the first occurrence is returned. If *mode* is –1, then the last occurrence is returned. If no line is found, an EOF is returned. let line = file_orc("file", 1) # Return the first line let line = file_orc("file", 3) # Return the third line let line = file_orc("file, -1) # Return the last line let line = file_orc("file", 1, "hello") # Return the first line containing "hello" let line = file_orc("file, -1, "hello") # Return the last line containing "hello"
file_owc(*file*, *mode*, *data*)	Opens the *file* with the *mode*, writes the *data*, and closes the file. See the table of valid modes below. A new line character is added to the end of the data. file_owc ("data", "w+", "Hello to you") # Writes "Hello to you" to a "data" file.
file_read(*fd*)	Read from a previously opened file referenced by the fd. File_read() returns the string "EOF" when an end of file is encountered. let line = file_read ($fd) # Assigns line to the value read.
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")
format(*str*, *fmt*)	Format a string. Used to format the *str* using the specified *fmt*. Equivalent to the 'C' sprintf(buf, %*fmt*s, *str*) or sprintf(buf, %*fmt*g, *str*) depending upon the data type of str. If it is a number, %*fmt*g is used, otherwise %*fmt*s 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.
get_tmp_var(*variable*) get_tmp_var("")	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

get_tmp_var(*variable*, *default*)	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 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) is 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.
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. cnt = 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 if the temporary *variable* exists in the program, otherwise, returns 0. if (is_tmp_var(num_wafers)) .... end_if Remember to use the variable name, not its value (num_wafers, not $num_wafers)
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 str is next converted to a binary str (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 hex conversion. Returns the integer *value* as a hex number. The conversion is specified using the fmt value. This is equivalent to the 'C' sprintf(str, "%fmto", value). The str value is returned. let x = itoh (12, 2.2) # Assigns "0C" 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 str value is returned. let i = itoo(12, 3) # Assigns 014 to i
mkprint(*data*)	Converts the *data* to printable characters.
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. Assigns to the variable *data[]*, the tokens parsed from the *str* delimitated by any of the characters contained within *delimiters*. Parse() returns the number of data variables assigned. The array starts with the index 1. If the *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 cnt = 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
save_tmp_var(*data*)	Saves all the tmp variables to the variable *data*. Any variables previously saved as *data*, will be remove. If "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
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 build upon the 'C' strftime() function. See the table of valid *fmts* 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 *str*. let x = strlen ("ABCDEFG") # Assigns x to 7
substr(*str*, *index*, *cnt*)	Get a substring. Returns *cnt* number of characters from the string, *str*, starting with the character at *index* (index starts a 1). 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 re-directed to stdout. Both stderr and stdout are received by the 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 include, 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.
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.

Valid entries 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 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
%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 1st 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.
%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).

%%	The percent (%) character.

Valid *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+	truncate 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

Protocols

This section will be addressed in another revision.

Command Protocol

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.

Options

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.

Standard Protocol

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" …
```

Options

The options for the standard 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.
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.
tmp_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.

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" …
```

Options

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.

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.v
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.

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 Act&#211;r</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>
```


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 temporay 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

Values are XML elements that fall between two tags. These are appended with a .value and are created as shown below:

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	title
	xml.1.1.1.value	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	name
	xml.1.1.2.1.1.value	Ms. Coder
<actor>Onlivia Actora</actor>	xml.1.1.2.1.2.tag	actor
	xml.1.1.2.1.2.value	Onlivia Actora
</character>		
<character>	xml.1.1.2.2.tag	character
<name>Mr. Coder</name>	xml.1.1.2.2.1.tag	name
	xml.1.1.2.2.1.value	Mr. Coder
<actor>El ActÓr</actor>	xml.1.1.2.2.2.tag	actor
	xml.1.1.2.2.2.value	El ActÓr
</character>		
</characters>		
<plot>	xml.1.1.3.tag	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.	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.
</plot>		
<rating type="thumbs">7</rating>	xml.1.1.4.tag	rating
	xml.1.1.4.attr.type	thumbs
	xml.1.1.4.value	7
<rating type="stars">5</rating>	xml.1.1.5.tag	rating
	xml.1.1.5.attr.type	stars
	xml.1.1.5.value	5
</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.

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

HSMS options

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.
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.

Annotated SECSII Message

This section discusses the format of the SECSII message.

The annotated SECSII message has the following format:

```
SstreamFfunction 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"
B	Binary	00 to FF, entered in Hexadecimal

BO	Boolean	T or F
U1, U2, U4, U8	Unsigned one, two, four, eight byte(s)	
I1, I2, I4, I8	Signed one, two, four, eight byte(s)	
F4, F8	Float four, eight bytes	Can be entered in hex format. Ex: '\xhhh..hh' (Use single quotes)

The '<>' are used to delimited the data types and their values, <*data_type value*>.

A typical S2f41 SECS message

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

Arrays can be entered by repeating the data value.

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

Arrays can also be entered using 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)

(1) Requires the pragma MORE_SECS_TMP_VARIABLES to be enabled.

The remaining 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	1
<U4 11 12 13 14 15>	S6F11.1.1	ARRAY
	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

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

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

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

Getting the sc version

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 script file. Whenever there is input on this channel sc will start script 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 temp variables. When using the SECS (SECS or HSMS) protocols there are several temp variable 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 temp 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 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 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_connection1 proto=hsms logging=2 host=10.100.32.1402 service=50003 options="t3=5 t6=10
t7=10 t8=10 timeout_msg=TIMEOUT"

# All communications from the host start here

label host_connection1

  let sxfy= get_tmp_var(SXFY)  4

  goto $sxfy5

# 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 S1F136
  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 script 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 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 temp variable 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 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_connection1 proto=hsms logging=2 host=10.100.32.1402 service=50003 options="t3=5 t6=10
t7=10 t8=10 timeout_msg=TIMEOUT"

label tool_connection1
   let sxfy = get_tmp_var(SXFY) 4
   goto $sxfy5


# S1F13 - Establish Communications, CommunicationState is COMMUNICATING
# This is what the response to an S1F13 is when returned from a "HOST"
label S1F136
   send_reply name=tool_connection S1F14 0
      <L
       <B 00>
       <L
       >
      >.


label est_com7
   send name=tool_connection S1F13 0 W
      <L>.

label hello7
   send name=tool_connection S1F1 0 W
      <L>.

label query7
   send name=tool_connection S1F3 0 W
      <L
```

```
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 recipe file
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;
int i, var, nest, nest_var;
#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 futher 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

