

GSEOS STOL User Manual

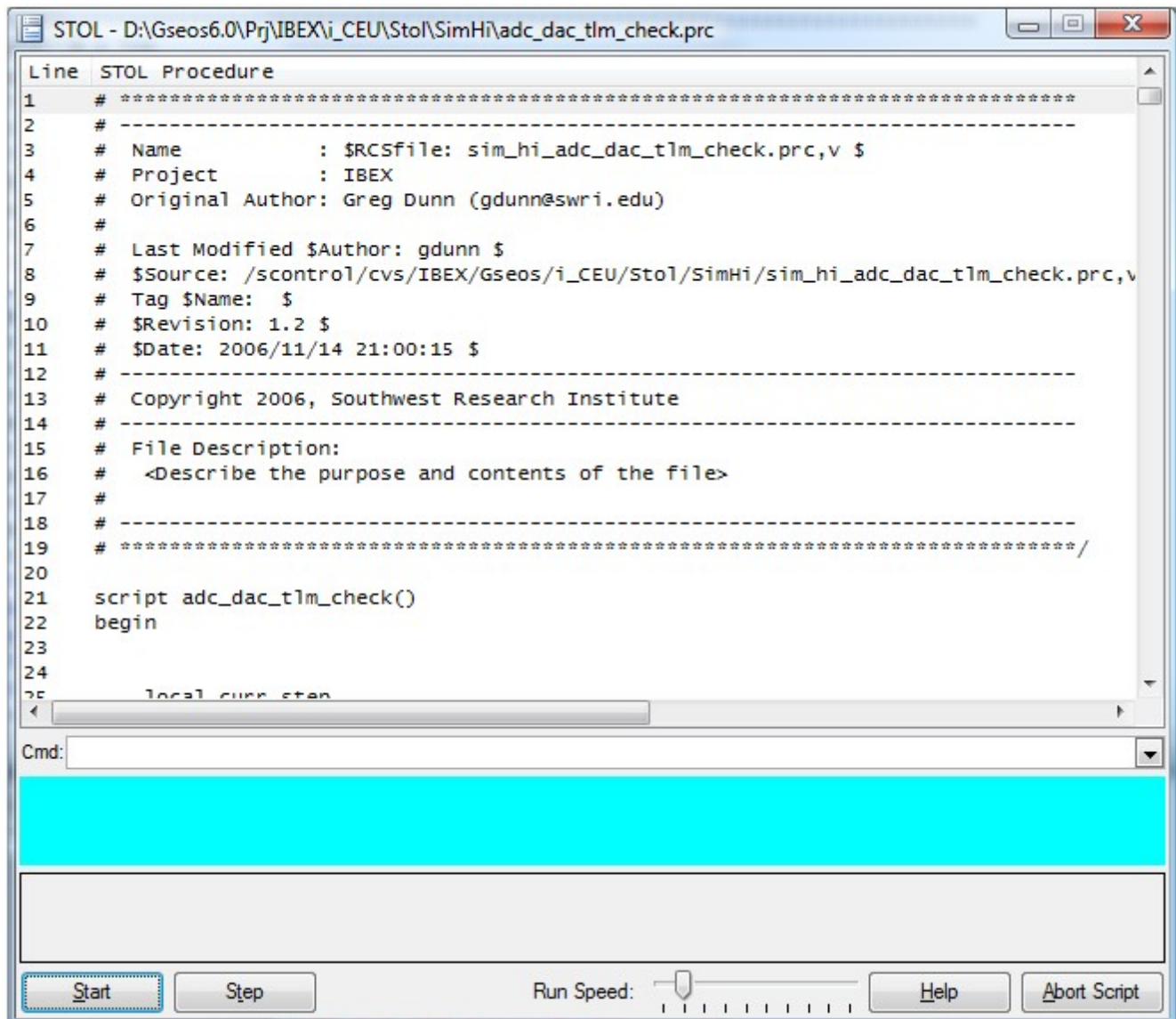
www.gseos.com

GSE Software, Inc. 1998 - 2017

1 GSEOS STOL User Manual

GSEOS STOL User Manual

GSE Software, Inc. 2017



1.1 Introduction

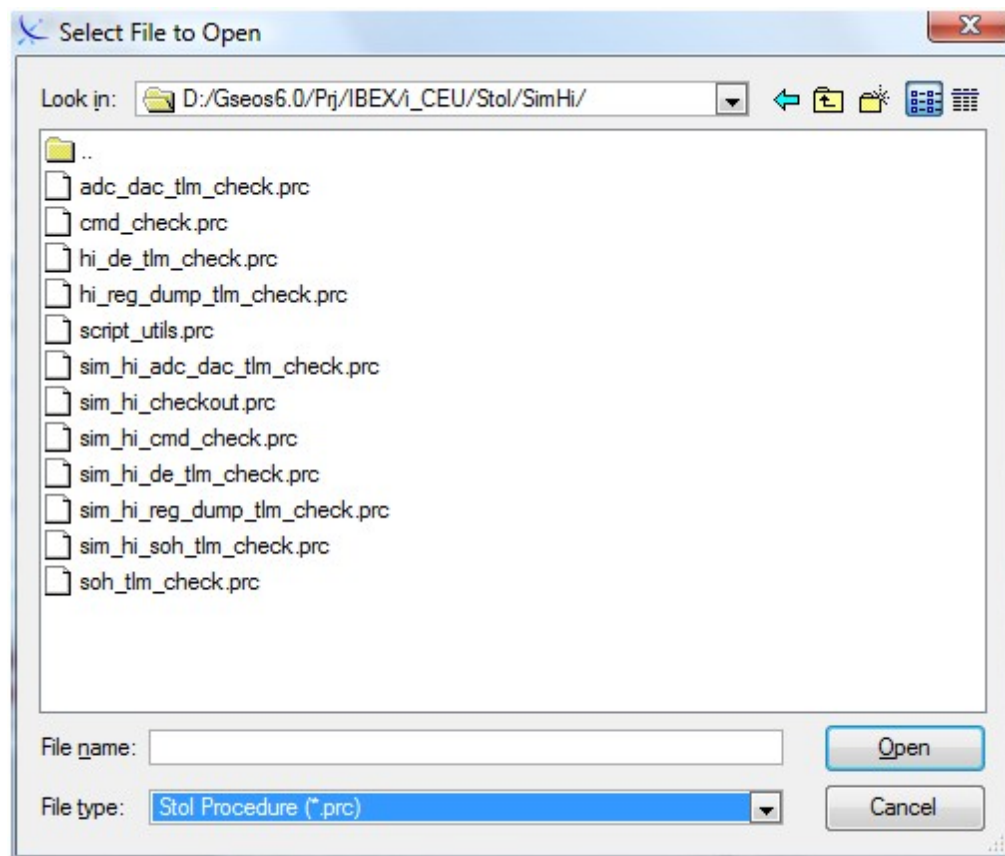
The STOL interpreter allows you to write command and verification scripts using a BASIC like procedural language. GSEOS STOL is loosely based on several of the industry common STOL implementations.

There are several reasons to use STOL over the GSEOS built-in scripting language Python:

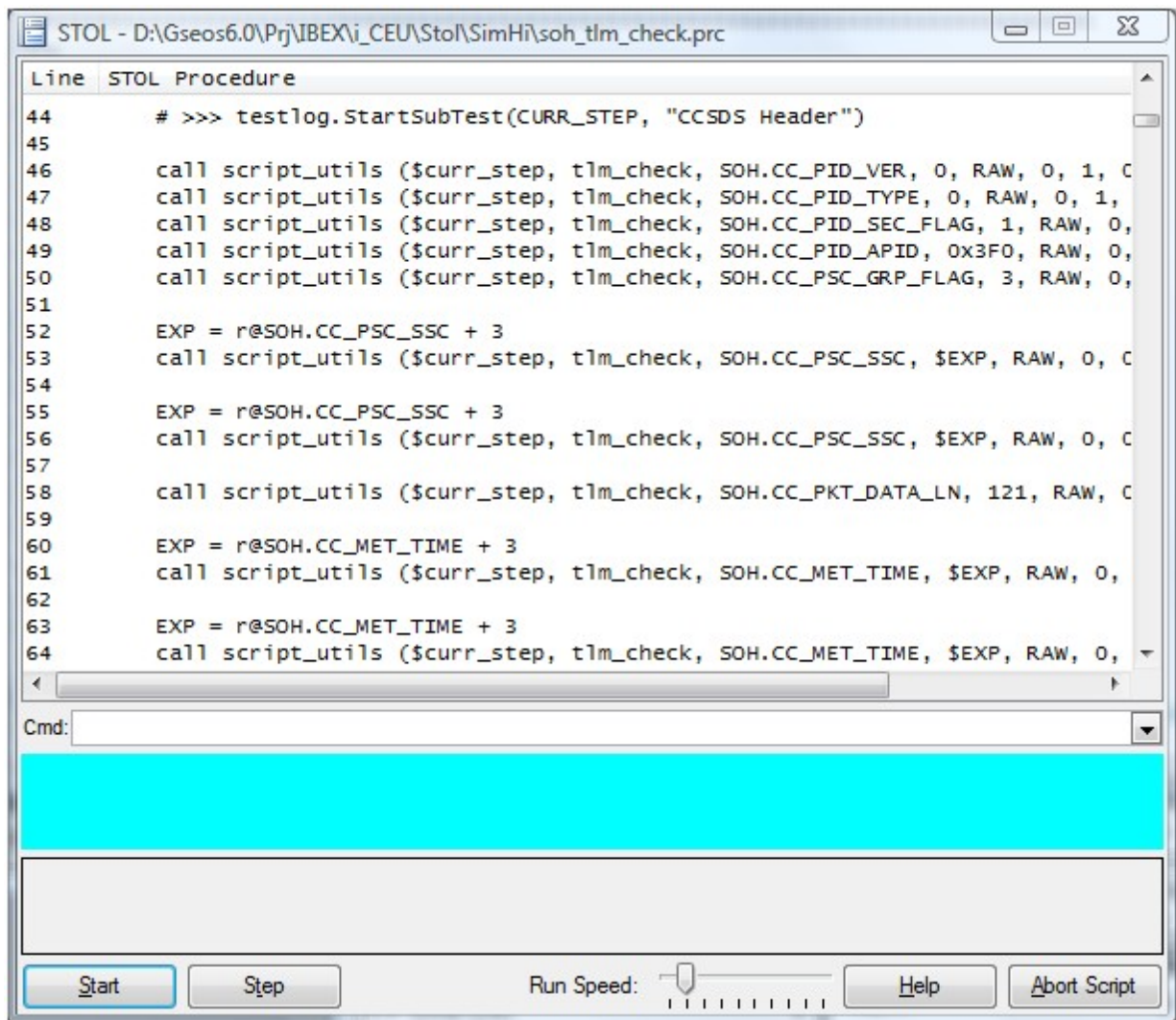
- Simple syntax, less complexity
- Easy access to TLM data points and commands
- Develop scripts that can run on the ground system
- Two-way test system to verify procedures in bench-test configuration

Quick Start

A STOL script is a plain text file that conforms to the syntax described in the STOL Language section. The file extension for STOL procedures is .gstol. To load a STOL procedure from GSEOS open the File/Open Menu and select the file type: GSEOS STOL Procedure (*.gstol):



Opening a STOL procedure will open the STOL Interpreter Dialog that displays your procedure and allows you to run it:



1.2 Installation

The GSEOS Stol interpreter is implemented in the GseosStol.pyd module. You simply have to import this module to have the STOL interpreter functionality:

```
import GseosStol
```

This can be easily done by including the module in the gseos.ini [PyStartup] section:

```
[PyStartup]
Import = GseosStol
```

Alternatively you can load the module through the [Config] section:

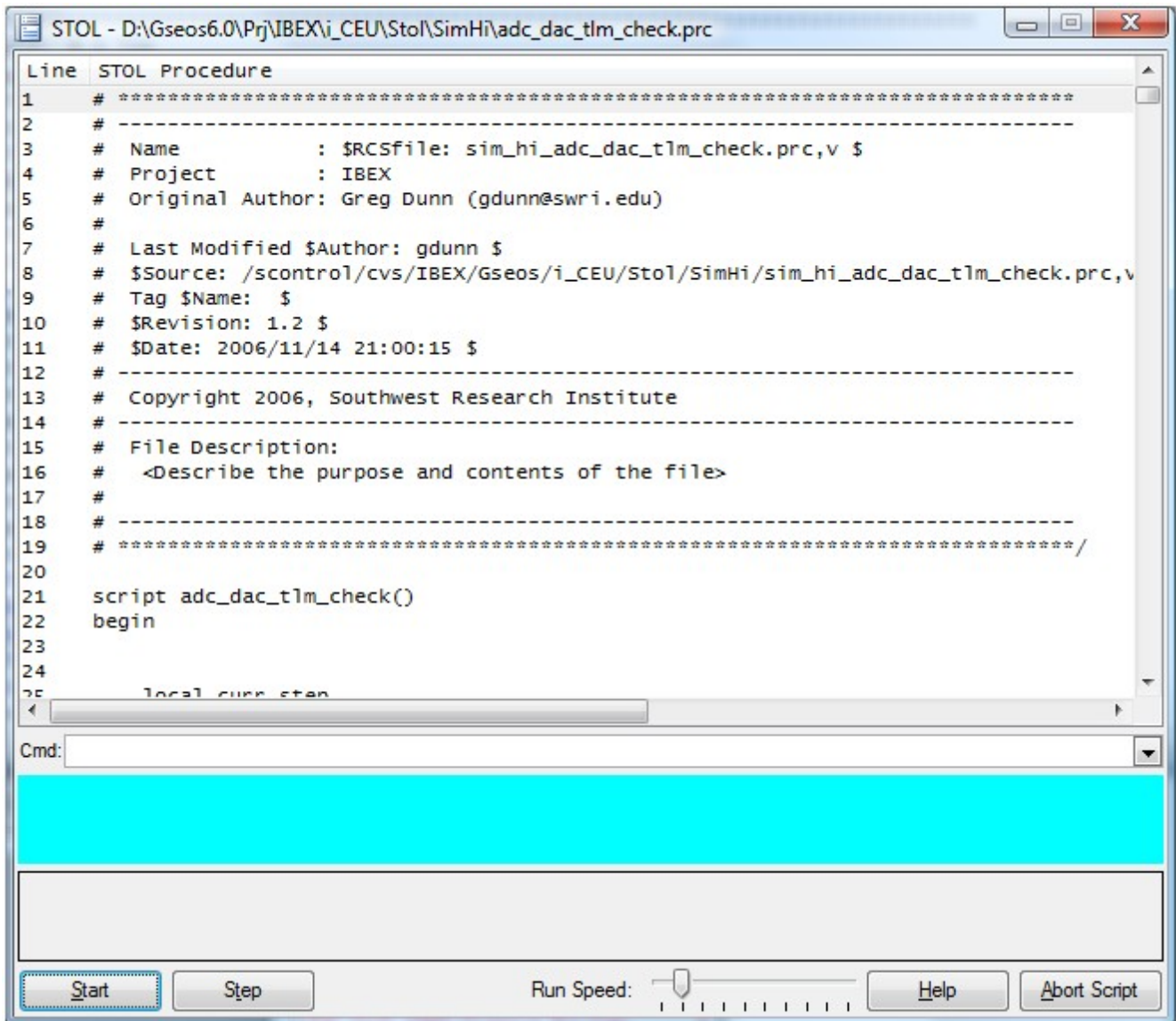
```
[Config]
Load = GseosStol.pyd
```

The module should reside in the GSEOS root folder for these imports to succeed. If you place the GseosStol.pyd module in a separate folder you will need to include the path to the module in the Load

entry. The documentation should be in the 'Doc' folder which is underneath the GSEOS root folder. For Windows the file "GSEOS STOL User Manual.chm" should be copied into this folder, for Linux the file "GSEOS STOL User Manual.pdf" should be copied into the 'Doc' folder.

1.3 STOL Dialog

The user interface of the STOL emulator consists of a dialog box that displays the currently executing procedure's source code as well as several user controls:



To run a STOL procedure simply open a .gstol file from a Python script or the GSEOS user interface. You can run multiple scripts side by side. However, keep in mind that they all issue commands to the single attached hardware (instrument) so there is usually not a good reason to run multiple STOL procedures concurrently unless they address separate subsystems.

Source Code Page

The source code pane shows the currently executing procedure with the current line being highlighted.

Although you can select any line within the source code this doesn't actually change the execution point. You can jump to any line (within limits, it is not possible to jump into and out of control statements) using the STOL **goto** directive using the **Cmd** line editor.

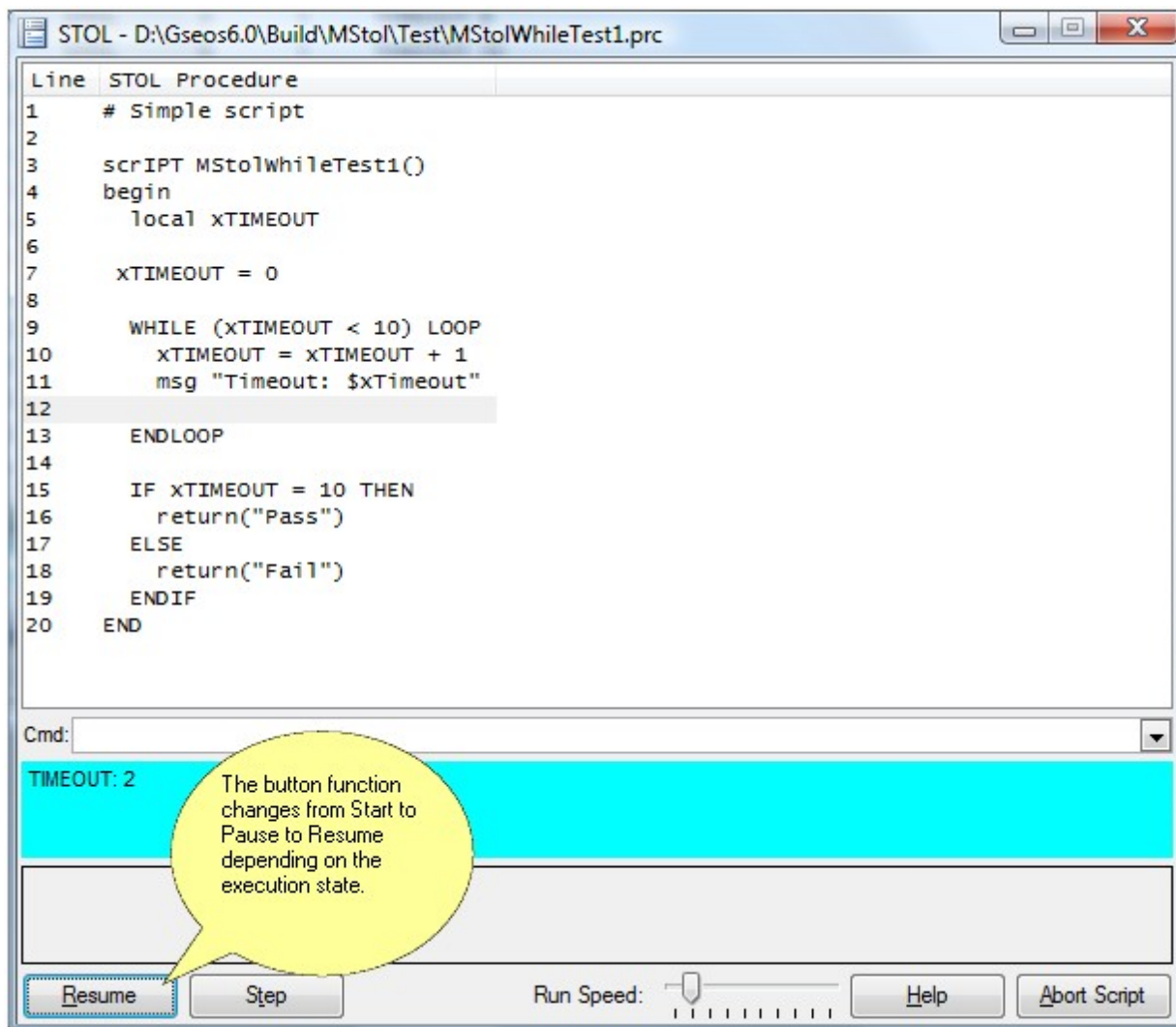
When your script calls a nested procedure the source code window loads the script of the nested procedure and restores the caller script upon return of the subroutine.

Cmd Line Editor

You can issue arbitrary STOL commands using the **Cmd** line editor. A command history is kept so you can quickly repeat often used commands. In general it is only possible to issue simple statements like: goto, msg, ask and so on. Control statements like: while, for will cause an error.

Start/Pause/Resume

To start a procedure you press the Start button. While a procedure is running you the button text changes to **Pause**. If you click the Pause button while the script is running execution is paused. The button then changes to **Resume**. To resume your script from where you paused it click the Resume button.



Single Step Mode

Instead of clicking the Start button you can execute the script in single step mode by clicking the **Step** button. This executes the procedure one line at a time. You can also single step from the paused state.

Run Speed

In addition to single step mode you can change the execution speed of your procedure. The **Run Speed** slider allows you to set the desired execution speed.

Help


The **Help** button opens this help file.

Abort Script

You can either simply close the dialog or click the **Abort Script** button to terminate the script. This will end the entire call stack (if there are nested procedures) and close the dialog. It will also report the termination to the console window.

msg

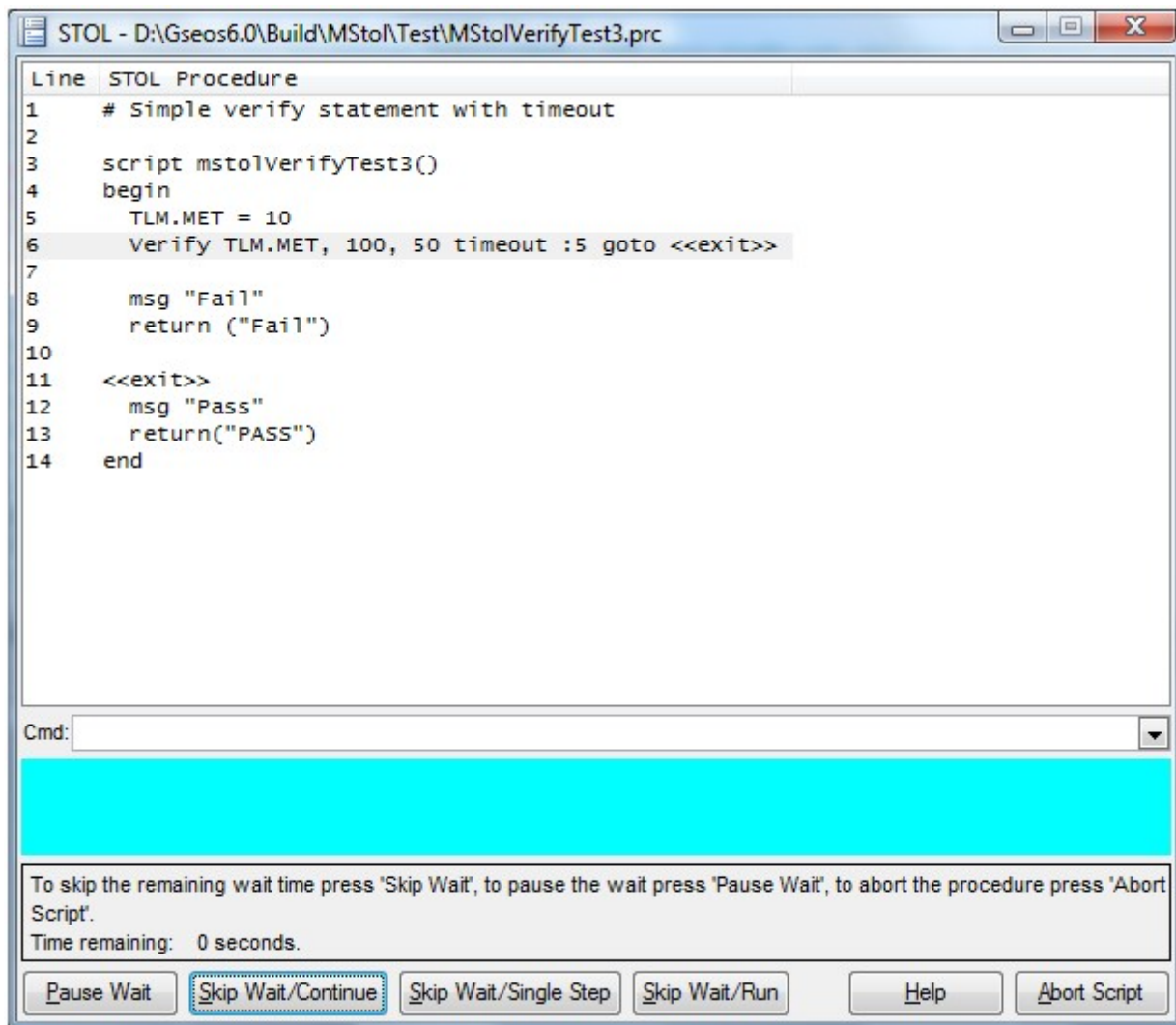
The **msg function** displays the message text in the message section of the STOL dialog and also writes the message to the GSEOS Message window.



VAR1: 55

Wait

The Wait directives can put a STOL procedure into a wait state. When a procedure enters a wait state the button bar on the bottom of the dialog changes as displayed in the picture below:



At this point you have several options:

You can pause the wait using **Pause Wait**. This will suspend the wait timer until you resume the wait and essentially keep the system in a wait state until you continue. You can skip the wait using the **Skip Wait/Continue**, or **Skip Wait/Single Step**, or **Skip Wait/Run** buttons. The Skip Wait/Continue button skips over the remaining wait time and continues in the mode it was in when it hit the wait statement. The Skip Wait/Single Step does the same but enters single step mode, and Skip Wait/Run aborts the wait and continues in regular run mode.

1.4 STOL Language Reference

There are many different implementations of STOL (Spacecraft Test and Operating Language) in use. The GSEOS implementation orients itself loosely on several of the more common versions. This chapter describes the language syntax and how you can interact with your telemetry data and commands. Besides using pure STOL you can also embed Python script within a STOL comment. This allows you more complex control without violating the general STOL syntax. Typically this Python script will only run in GSEOS and be ignored if you run your STOL script on the ground system. The

chapter STOL and Python describes details of the Python integration.
The following chapters give an overview of the STOL syntax and semantics:

STOL Procedure Basics

Data Types

Variables

Expressions

Telemetry Data

Commands

Procedure Elements

Flow Control Directives

Built-in Functions

1.4.1 STOL Procedure Basics

A procedure is a plain text file containing a sequence of the following procedure elements: Comments, Labels, Statements, and Flow control directives. A procedure has the following layout:

```
script ScriptName([Arguments])  
begin  
    Procedure Elements  
end
```

The keyword **script** starts a procedure declaration. if arguments are supplied the formal argument names are specified in the list following the script name. There is a one-to-one relation between a procedure file and a script. A procedure file must have exactly one procedure definition. The procedure directives are enclosed between the **begin** and **end** keywords.

A procedure can be invoked from within a calling procedure with the call directive. The return directive allows you to return data from a STOL procedure. As with other data elements in STOL a procedure doesn't contain any explicit type information, so neither the return type nor the argument types are defined in the procedure declaration.

STOL is **case sensitive**. All language directives are lower case.

1.4.2 Data Types

GSEOS STOL supports the following data types: Integer, Real, Boolean, String, Time. STOL is a loosely typed language and automatic type conversion takes place as appropriate.

Integer values can be represented as hex, octal, or decimal (default) literals. To indicate a hexadecimal value you must prefix the literal with 0x, for an octal number you must specify a leading zero, all other literal integers are interpreted as decimal. The following are valid integer literals:

```
0xed  
007  
077  
123  
0xABCD
```

600000
2

A real value must contain a decimal point and have at least one digit after the decimal point. Exponents are allowed and can be positive and negative. The following are valid real values:

1.2
0.4e44
1e+4
1e-3
.6

The Boolean values are: true, and false.

String data must be delimited by double quotes:

"This is a STOL string."

To embed a literal double quote in a string use the backslash escape:

"This is a STOL string that contains a double quote (\")."

Time data can be either absolute or relative. The time format is:

[[[YYYY/]DDD:]HH:]MM:]SS or [[[[YYYY/]MM/]DD:]HH:]MM:]SS

2009/02/23:15:02:33, 04:33, :03

String Formatting

You have control over the output formatting of strings using the % operator. This is similar to the C programming language. Each string can have at most one formatting operator. The argument is separated from the format string with the '%' character:

"This print an integer in decimal: %d" % 44

"This print an integer in hex: %04X" % 44

"This inserts a string: %s" % "String"

The following format characters are available:

| | |
|---|---|
| d | Signed integer decimal. |
| i | Signed integer decimal. |
| o | Unsigned octal |
| u | Unsigned decimal. |
| x | Unsigned hexadecimal (lowercase) |
| X | Unsigned hexadecimal (uppercase) |
| e | Floating point exponential format (lowercase) |
| E | Floating point exponential format (uppercase) |
| f | Floating point decimal format |

| | |
|---|--|
| F | Floating point decimal format |
| g | Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise |
| G | Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise |
| c | Single character (accepts integer or single character string) |
| s | String |

Besides the format character you can also specify width and precision if applicable.

1.4.3 Variables

The declaration of variables is optional. If variables are declared this should be done at the top of the script within the begin...end sequence. They can be either local or global. Global variables have a scope that encompasses all procedures, local variables only have local scope within the current procedure. Although you can 'hide' global variables with locals with the same name this practice should be avoided. If you don't declare a variable a local variable is created for you on the fly at the first assignment. The declaration doesn't specify a data type but you can optionally initialize a variable:

```
script VarDecl(flArg1, flArg2, dwArg3)
begin
    local wCnt=0, wTrigger
    global wShutter
end
```

The above example defines five local variables and one global. The three procedure arguments are local variables as are the explicitly defined locals.

Before variables are accessed they must be assigned to. This can be a literal value or the result of an expression:

```
wCnt = 88
strCmdState = "NOP"
flGMT = LocalTime - 08:00:00
```

Procedure arguments are initialized by the calling procedure. For example the following call will initialize the arguments flArg1, flArg2, and dwArg3:

```
start VarDecl(23, flGMT, wCnt)
```

The arguments are passed literally. That is variable names are NOT dereferenced. If you intend to pass the 'value' of the flGMT variable into the script you have to use the dereference operator (\$). The above example would set the arguments to the following values:

```
flArg1: 23
flArg2: "flGMT"
dwArg3: "wCnt"
```

To pass the actual values you would write:

```
start VarDecl(23, $flGMT, $wCnt)
```

Dollar Escape

A \$ character in a script will cause the variable following the \$ to be substituted by the variable's value. This allows a pre-processor like behavior. See the example below:

```
Instrument = "PEPSI"
```

```
goto <<Start$Instrument>>
```

```
<<StartPEPSI>>
```

```
<<StartLORRI>>
```

The interpreter use the entire identifier after the dollar character. If you want to control when to stop interpreting the characters as an identifier you can use the curly brackets to limit the identifier. The following example sets the variable voltage to 4. If you want to print say the value of the voltage together with the unit: 4V you would use \$voltageV, however, the interpreter will search for a variable named voltageV which it can't find and issue an error. If you limit the replacement it will only replace the value in braces: \${voltage}V.

```
voltage = 4
```

```
; Error
msg "$voltageV"
```

```
; Print value with unit
msg "${voltage}V"
```

1.4.4 Expressions

Expressions can contain arithmetic, relational, or boolean operators (logical and bitwise). The following operators are supported:

Operators (listed in order of precedence):

Arithmetic: *, /, mod (modulo), rem (remainder), +, -,

Bitwise: ! (not), ^ (xor), | (or), & (and)

Relational: ==, !=, <, <=, >, >=

Logical: ! (not), || (or), && (and)

Although STOL has built-in precedence rules for operators it is highly recommended to use parentheses to control the order of evaluation.

The **assignment operator** is a single equal sign: '='.

Examples

```
wMask = (wVar1 & 0x34) | wVar2
bOk = wVar2 == 223
bNotOk = !bOk
Result = 2 + 3*cos(2.2)
```

```
bTrue = 4.5 < 5.4
```

1.4.5 Telemetry Data

You can access real-time telemetry data from your STOL scripts.

You use the general GSEOS naming syntax to access telemetry points: Block.Item. If the item is an array item you must also specify the index you want to access. You can't use slicing syntax (e.g.: TLM.Data[:30]) to access an array of times. The resulting value must always be a scalar:

```
if TLM.ApID == 0x233 then
    msg "Received PHA data."
endif
```

The TLM.ApID variable will change depending on the data arriving in the system. So if you run this statement in a loop it would read the current value. Keep in mind that this technique is not sufficient to check for a particular telemetry value, since the data might change at a time when you are not currently executing the conditional statement. Use the wait statement for this purpose.

You also can access converted telemetry values (engineering units). In this case you have to make sure the telemetry point has a conversion function named 'EU' defined. To request engineering units prefix the telemetry point with the EU directive: EU(SOH.LAST_OPCODE). The default (if no EU() is specified is the raw data value.

1.4.6 Commands

Commands are issued using the built-in directive: **send**. The **send** directive takes the command mnemonic to issue and any arguments.

```
send Mnemonic([Arg1, Arg2, ..., ArgN])
```

Commas ',' are used as a delimiter between arguments. The arguments are enclosed in parentheses. If no arguments are required you still have to supply the parentheses (e.g.: NOP()).

Examples

```
send EGSE_TIMESTAMP_ENABLE(OFF)
```

```
send CEU_LO_IF_HVPS_CTRL SYNC_CLK_EN(PAC_DIS, MCP_DIS, LV_EN)
```

```
send CEU_SSR_FLUSH_BUFFER (4+$Level)
```

1.4.7 Procedure Elements

The main elements of a STOL procedure are: Comments, Labels, Statements, and Flow Control Directives.

Comments

Comments are started with the semicolon ';', hash '#', or double hyphen characters '--'. Comments extend through the end of the line.

Examples

```
send NOP() ; This command does nothing.
```

```
; Turn instrument power on  
send PEPSI_INSTR_A_PWR_ON()
```

Labels

Labels are used as targets for goto directives. A label can be either enclosed in '<<' '>>' or have a colon ':' as the last character.

Example

```
goto Step5
```

```
<<Step5>>
```

```
goto End
```

```
End:  
  return
```

1.4.7.1 Flow Control Directives

1.4.7.1.1 begin

The begin directive marks the start of a procedure body. This must be the first directive in a STOL procedure. The procedure is terminated with the end directive.

1.4.7.1.2 call

The call directive is used to start execution of a nested STOL procedure (subroutine). The format of the call directive is:

```
RetVal = call ProcedureName([Arg1, Arg2, ... ArgN])
```

The arguments are enclosed in parentheses and separated by comma. The calling procedure stops executing until the callee returns. The return value of the called procedure can be used as the return value of the call directive. The ProcedureName is the name of a procedure without the .prc extension. So in order to call the script PowerOn.prc you would issue:

```
call PowerOn()
```

In general it is assumed that all STOL procedures remain in the same directory. However, it is possible to refer to procedures in different directories by prefixing the procedure name with the relative path:

e.g.: `call ../PowerOn()`

Argument passing

Arguments can be arbitrary expressions. The arguments are evaluated before being passed to the callee. This is also true for telemetry points. If you want to pass a symbolic telemetry point (i.e. by reference) you can simply pass the telemetry point as a string and de-reference it using the \$ replacement syntax on the callee side. This will be most likely used for functions that use the 'wait for arrival' statement. See the example below:


```
call foobar("TLM.Item1")
```

```
script foobar(TLMPoint)
begin
    wait for arrival $TLMPoint
end
```

1.4.7.1.3 end

The end directive closes a procedure body and must be the last directive of a STOL procedure. It is paired with the opening begin directive.

1.4.7.1.4 for

The **for** directive is used in conjunction with the loop directive. This construct is used to repeat a block of statements a given number of times. See also the while directive.

```
for Var = StartValue to EndValue loop
    Statements
endloop
```

```
for Var = StartValue to EndValue step StepValue loop
    Statements
endloop
```

The for directive uses a step of 1 by default. You can specify a different step value (including a negative value) using the step directive as indicated above.

Example

```
for wCnt=0 to 10 loop
    send NOP()
endloop
```

1.4.7.1.5 goto

The goto directive transfers execution to a line number or label within the current procedure. In general this leads to unstructured code and is to be avoided except for jumps to termination code within a procedure.

```
goto LineNumber | Label
```

Example

```
goto Finalize
```

```
Finalize:
  msg "End of Procedure"
```

1.4.7.1.6 if

The **if/else/elseif/endif** directive allows to conditionally execute script code. The if directive takes a boolean expression that if evaluated to TRUE will execute the code between the if and else/endif directives. If the expression evaluates to FALSE the code between else and endif will be executed if an else clause is specified.

```
if (BooleanExpression) then
  Statements
endif
```

```
if (BooleanExpression) then
  Statements
else
  Statements
endif
```

```
if (BooleanExpression) then
  Statements
elseif (BooleanExpression) then
  Statements
else
  Statements
endif
```

Example

```
prev_cnt = LO_ENG.CC_PSC_SSC
wait 4
curr_cnt = LO_ENG.CC_PSC_SSC

if (prev_cnt != curr_cnt) then
  msg "$CURR_STEP: LO_ENG.CC_PSC_SSC = $CURR_CNT (prev count = $prev_cnt)"

else
  msg "TFAIL: $CURR_STEP: LO_ENG.CC_PSC_SSC = $CURR_CNT (prev count = $prev_cnt)"
endif

if (convert == "T_RAW") then
  tlm_mnemonic = "$tlm_mnemonic"

elseif ((convert == "T_EU") || (convert == "T_TEXT")) then
  tlm_mnemonic = "EU($tlm_mnemonic)"

else
  msg "TFAIL: $CURR_STEP: Invalid conversion type specified"
```

```
# >>> test.TestLogFail(CURR_STEP, "Invalid conversion type specified")
goto EOP
endif
```

1.4.7.1.7 return

The **return** directive terminates the procedure immediately. The return value is returned to the calling procedure.

return(ReturnValue)

1.4.7.1.8 start

The **start** directive starts a new STOL procedure. This is very similar to the call directive. However, start spawns off the new procedure which executes side by side with the calling procedure. This is different from the call directive which blocks the calling procedure.

start ProcName([Arg1, Arg2, ..., ArgN])

1.4.7.1.9 wait

The **wait** directive allows to wait for an expression to become true. This is usually only meaningful if the expression contains one or more telemetry points that can change during the wait. There are a number of different uses of the wait statement:

a) Unconditional wait.

wait [timeout Expression]

Samples:

wait

wait timeout 20

b) Conditional wait

wait Expression [timeout Expression] **then**
Statements

[else
Statements]

endwait

A conditional wait waits until the conditional expression evaluates to true. The wait can have an optional timeout associated, the default is no timeout. The according 'if' statement would be the equivalent of a timeout of 0. If no timeout is specified the function will wait indefinitely for the condition to become true. If the wait condition evaluates to true the statements between the wait and else are executed. The optional else clause is executed if a timeout occurs or the user aborts the wait.

Samples:

```
#  
# No timeout. Timeout clause may be entered if the user aborts the wait.  
#  
wait TLM.Item <= 7 then  
  msg "Data is in acceptable range"  
  
else  
  msg "Timeout or aborted by user"  
  
endwait
```

```
wait EU(Data.Heater) == 3.2 timeout 22 then  
  log info MyLog "Heater ok"  
  
else  
  log err MyLog "Timeout"  
  
endwait
```

```
wait (3.2-1) < EU(Data.Heater) < (3.2+1) timeout 10 then  
  ... condition true  
  
else  
  ... timeout  
  
endwait
```

```
wait 1.2 < EU(Data.Heater) < 4.2 timeout 4 then  
  ... condition true  
  
else  
  ... timeout  
  
endwait
```

c) Wait for arrival of data item

The conditional wait statement evaluates an conditional expression that you provide. Usually this expression will contain one or more telemetry points. However, the wait statement doesn't actually 'wait' for the data item to arrive, it merely waits for the condition to become true. So it will operate on stale data. This might lead to undesirable script execution. The 'wait for arrival' statement lets you ensure that telemetry data is flowing before entering a conditional wait statement:

wait for arrival TLM Mnemonic **timeout** Expression **then**
Statements

[else
Statements]

endwait

Note that the 'wait for arrival' statement doesn't take a condition, it just waits for the block arrival (although you specify an item name, only the block is really relevant).

If a timeout occurs the (optional) else clause is executed. If you need to ensure multiple data items (from different blocks) to not be stale you can use multiple 'wait for arrival ...' statements.

If the block you are waiting for is periodic the worst case wait should be one period (in case you just missed the last block before entering the wait statement).

Samples:

```
wait for arrival TLM.Item timeout 40 then
  msg "Received TLM data"
else
  msg err "Timed out waiting for TLM data"
endwait
```

1.4.7.1.10 while

The while directive is used in conjunction with the loop directive. This construct is used to repeat a block of statements until the end condition is met.

```
while (BooleanExpression) loop
  Statements
endloop
```

Example

```
while (wRet != 0) loop
  send NOP()
  wRet = call GetCmdSuccess()
endloop
```

1.4.7.2 Built-in Functions

STOL has a number of built-in functions. They can be categorized into different groups:

Algebraic Functions:

abs, max, min, mod, sqrt

abs() take a single integer or real argument and returns the absolute value.

min(), and max() take two arguments and return the smaller, and larger value respectively.

mod() takes two integer arguments and returns the modulus (the remainder of the division of the two integers).

sqrt() takes a single integer or real argument and returns the square root of the value.

Examples:

```
x= abs(-4)
x=min(4, 50)
x=max(-3, 4)
x=mod(20, 10)
x=sqrt(44)
```

Trigonometric Functions:

acos, asin, atan, atan2, cos, sin, tan

sin(x) takes one integer or real argument and returns the sine of x radians.

cos(x) takes one integer or real argument and returns the cosine of x radians.

tan(x) takes one integer or real argument and returns the tangent of x radians.

asin(x) takes one integer or real argument and returns the arc sine of x in radians.

acos(x) takes one integer or real argument and returns the arc cosine of x in radians.

atan(x) takes one integer or real argument and returns the arc tangent of x in radians.

atan2(x, y) takes two integer or real arguments and returns the arc tangent of x in radians of the quotient of the two arguments.

Logarithmic Functions:

ln, log10, log2

The natural logarithm (base e) is computed using the function ln(). The function log() is used for logging.

ln(), log10(), and log2() take one integer or real argument and return the logarithm to the base e, 10, and 2 respectively.

Time Functions:

current_time, year, month, day, hour, min, secs

These functions take a time value as their argument and return the according part of the time value (as an integer value). The current_time() function doesn't take any arguments and returns a time object with the current local time.

Misceallaneous Directives:

ask, beep, log, msg, msgbox, playsound

1.4.7.2.1 ask

The **ask** function prompts the operator for a value. A pop-up box asks the operator to enter a value which is then returned by the function. Optionally you can specify the **buttons** or **menu** keywords and supply a list of values. The ask dialog then renders the values as a drop down list (there is no difference between the button and menu keywords).

ReturnString = **ask** "PromptText" [**buttons**(Arg1, Arg2,..., ArgN) | **menu**(Arg1, Arg2,..., ArgN)]

Examples

```
var = ask "Enter a value"
var = ask "Enter a value" buttons (1, 2, "Testing", var1, $var1)
var = ask "Enter a value" menu(1, 2, "Testing", var1, $var1, 1.23, TRUE)
var = ask "Enter a value" menu(1, 2, "Testing", var1, EU(TLM.MET))
```


1.4.7.2.2 beep

The **beep** statement sounds a simple audible alarm. You have to make sure the file 'boing.way' is in your GSEOS installation.

1.4.7.2.3 log

The **log** directive logs a text to the specified log file. You can either use pre-configured automatic logs or single file log files (these must end in .log or .html). The first argument to the log directive is the name of the log (either the name of the automatic log or the name of the log file). The remaining arguments are a comma separated list of expressions to log. The individual values are computed and concatenated to a single string. You can optionally specify a level: info, warn, or err. The default if not specified is info. This logs the entry with a different color (for plain ASCII files (the ones ending in .log) the color is not retained in the file).

Examples

log [info|warn|err] Log Arg1, Arg2, Arg3, ...

Samples:

```
log StolLog, "The temperature of heater 1 is ", INSTR_TEMP_CTRL.wHtr1, " Celsius"
```

```
var1 = 77
```

```
log TestRun08.log, "var1: ", var1
```

```
log err ErrorLog "Error code: ", wError
```

1.4.7.2.4 msg

The **msg** directive logs a user message to the GSEOS message window. The arguments are a comma separated list of expressions to log. The individual values are computed and concatenated to a single string. You can optionally specify a level: info, warn, or err. The default if not specified is info. The level is indicated in the message window.

Examples

msg [info|warn|err] Arg1, Arg2, Arg3, ...

Samples:

```
msg "The temperature of heater 1 is ", INSTR_TEMP_CTRL.wHtr1, " Celsius"
```

```
var1 = 77
```

```
msg warn "var1: ", var1
```

```
msg err "Error code: ", wError
```

1.4.7.2.5 msgbox

The **msgbox** directive pops up a message box which needs to be acknowledged by the user before the script continues running. The message box is modeless which means you can still interact with GSEOS while the message box is active. However, that also means that the message box can be obscured by other windows. The arguments to the msgbox statement are the same as for the msg statement. You can optionally specify a level: info, warn, or err. The default if not specified is info.

Examples

msgbox [info|warn|err] Arg1, Arg2, Arg3, ...

Samples:

```
msgbox "The temperature of heater 1 is ", INSTR_TEMP_CTRL.wHtr1, " Celsius"
```

```
var1 = 77
msgbox warn "var1: ", var1
```

```
msgbox err "Error code: ", wError
```

1.4.7.2.6 playsound

The **playsound** directive plays a .wav file. This is similar to the Gseos.PlaySound() function. The playsound statement takes one argument which is the wav file name. If the file can't be found or can't be played the function fails without sound.

Examples

playsound "Error.wav"

1.5 STOL and Python

Since STOL is (on purpose) very limited it is possible to embed Python code within STOL comments. These will be executed in GSEOS and ignored if the script is run in other environments.

Embedding Python

It is possible to invoke Python statements directly from the STOL script. The invocation is hidden in a STOL comment. The escape sequence to indicate a Python statement is >>>. So the following would be a valid embedded Python statement:

```
# >>> print 'Hello, STOL'
```

Be aware that none of the embedded Python code will be recognized by the ground system (if you plan on deploying your scripts) and if you plan on running your scripts on both systems you should keep the amount of embedded Python code to a minimum. However, in some test cases embedding Python code might be helpful.

```
#
#
# Testing embedded Python code
# >>> def f(a):
```

```
# >>> print 'This is output from an embedded Python function: ', a #
# >>> f('Hello, World.') # Call it here
#
#

value = ask "Please enter a number"

#
# >>> f('Another test')
#
```

The indentation has to be consistent within a block. A block is defined as any consecutive number of lines with embedded Python code. Indentation between different blocks can vary. The individual code blocks are just being fed to the Python interpreter as they are encountered. So you can write some embedded Python code, like a function definition and call it later in the STOL script.

One restriction: A line with embedded Python code cannot contain any Stool commands. That means it must be a comment line only.

Wrong:

```
send EMBOX_OFF() # >>> print 'Turn box off'
```

Ok:

```
# >>> print 'Turn box off'
send EMBOX_OFF()
```

Variables

STOL variables are actually converted into Python variables with the same name. So by declaring local TESTNUMBER you effectively define a Python variable. STOL nor Python variables are typed. If you assign to STOL variables from your embedded Python code you will need to make sure to use the proper type conversion. I.e. in the example below the GSEOS.InputDialog box simply returns a string. You will need to convert the string to the appropriate data type, in this case an integer. You can do that with the Python eval() statement, this will make sure to properly convert decimal, as well as hexadecimal input.

The example also shows that you can directly assign the result to the STOL variable TESTNUMBER which is accessible from Python as explained above.

String variables are handled as their own data type in STOL. If you need to transfer a STOL string variable to Python you have to explicitly convert it to a Python string before you can access it as a string:

```
local MacRev = "Rev 1.0 MMS EM Basic Test Config"
# >>> fMyProcessing(str(MacRev))
```

The STOL string type has the `__str__` conversion function defined. So if you just want to print the contents of the variable you don't need to convert it (although it doesn't hurt either).

1.6 Custom STOL Error Handler

You can install an error handler that gets called when errors are encountered in the script or the user aborts the script. Your error handler takes one argument which is the exception object.

To register the error handler you call:

```
GseosStol.SetErrorHandler(fMyErrorHandler)
```

where `fMyErrorHandler` is your error handler. It has to have the following signature:

```
def fMyErrorHandler(oX):
    ....
```

There are three different return values that are recognized:

| | |
|---------|--|
| None/0: | The normal error prompt will be presented. |
| 1: | No prompt will be presented and the script will continue (the error is ignored). If the user aborted the script the script will be aborted without a prompt. |
| 2: | No prompt will be issued and the script will be aborted. |

If your handler raises an exception the error will be logged in the message window.

Example

```
def fStolErrHandler(oX):
    Gseos.Message('Stol error: %s' % oX)
    return None

GseosStol.SetErrorHandler(fStolErrHandler)
```

1.7 Custom STOL Event Handler

Like the error handler you can also install an event handler. The following events are supported:

```
GseosStol.STOL_EVENT_SCRIPT_STARTED
GseosStol.STOL_EVENT_SCRIPT_FINISHED
GseosStol.STOL_EVENT_STOL_CLOSED
```

When you register a custom event handler you will get notified of all events. Your function must take two arguments: The event and an optional parameter that varies from event to event.

The `STOL_EVENT_SCRIPT_STARTED` and `_FINISHED` will provide the STOL file name as the second argument, the `STOL_EVENT_STOL_CLOSED` will provide `None`.

To register the error handler you call:

```
GseosStol.SetEventHandler(fMyEventHandler)
```

where `fMyEventHandler` is your custom event handler. It has to have the following signature:

```
def fMyErrorHandler(wEvent, oArg):
    ....
```

If your handler raises an exception the error will be logged in the message window.

Example

```
def fStolEventHandler(wEvent, oArg):
    if wEvent == GseosStol.STOL_EVENT_SCRIPT_STARTED:
        Gseos.Message('The Stol script: %s started.' % oArg)

GseosStol.SetErrorHandler(fStolEventHandler)
```

1.8 Example STOL scripts

The following samples show the general structure and elements of a STOL procedure. (They are not meant to be executed on your system).

1.8.1 Sample 1

```
# Header ====={{{
# -----
# Name:      iftp.gstol
# Project:   14085-HOPE
# Author(s): Greg Dunn
# Created:   9/2/2009 4:48:38 PM
# -----
# SVN Header
#
# $Author: gdunn $
# $URL: http://folger/svn-rbsp/trunk/GroundSoftware/Gseos/Instruments/ECT/HOPE/Scripts/Stol/
# IFTP/iftg.gstol $
# $Revision: 1921 $
# $Date: 2010-06-17 12:02:59 -0500 (Thu, 17 Jun 2010) $
# -----
# Copyright 2009, Southwest Research Institute
# -----
# File Description:
# Implements a basic instrument functional test procedure for the HOPE
# electronics unit.
#
# -----
# =====
# -----
# Header }}}--
# Imports -----{{{--
# >>> from Instruments.ECT.HOPE.Util import testlog as testlog
# Imports }}}--
# Script -----{{{--
script iftp()
begin

    # >>> testlog.StartNewTestLog("IFTP")
    # >>> testlog.StartTestSeries("Instrument Functional Test Procedure", "$Id: iftp.gstol 1921
2010-06-17 17:02:59Z gdunn $")
    # >>> testlog.StartTestLog("Instrument Functional Test Procedure", "$Id: iftp.gstol 1921
2010-06-17 17:02:59Z gdunn $")

# -----
# Query user for test setup parameters
```

```

# -----
ask_hv_plug_state:
    exp_hv_plug = ask "Expected HV Plug State?" buttons ("Disabled", "Limited", "FullEnable")
    # >>> exp_hv_plug = exp_hv_plug.upper()

ask_cem_target:
    cem_target = ask "Enter CEM target voltage:"

    # Validate the target voltage
    # >>> try:
    # >>>     x = float(cem_target)
    # >>>     if (x<0) or (x>9000):
    # >>>         invalid_cem=1
    # >>>     else:
    # >>>         invalid_cem = 0
    # >>> except:
    # >>>     invalid_cem = 1

    if (invalid_cem == 1) then
        msgbox err "Invalid entry for CEM target: $cem_target"
        # >>> testlog.TestLogWarning(0, "Invalid entry for CEM target: %s" % cem_target)
        goto ask_cem_target
    endif

ask_test_len:
    test_len = ask "Short or Full FTP?" buttons("Short", "Full")

ask_end_state:
    end_state = ask "Script final state?" buttons("PowerOff", "ReadyForScience")

    # -----
    # >>> testlog.StartSubTest(0, "Power On")
    # -----
    call ../Util/power_on()

    # Turn on diagnostic telemetry
    call ../Util/cmd_check("HOPE_SET_PARAMETER", "1, Tlm_Rate_Diag", 0)

    # -----
    # >>> testlog.StartSubTest(0, "ADC Test")
    # -----

    # >>> testlog.TestLogRemark(0, "Temperatures -----")
    # TODO: sane temperature range? This is -25 to 35 C (-45 to 127F)
    start ../Util/tlm_check("p@HOPE_HK.FEE_TEMP"           , 5, 30, 0)
    start ../Util/tlm_check("p@HOPE_HK.LVPS_TEMP"          , 5, 30, 0)
    start ../Util/tlm_check("p@HOPE_HK.HVPS_TEMP"          , 5, 30, 0)
    start ../Util/tlm_check("p@HOPE_HK.HOPE_DB_TEMP2"      , 5, 30, 0)
    start ../Util/tlm_check("p@HOPE_HK.HOPE_DB_TEMP1"      , 5, 30, 0)
    wait timeout 3

    # >>> testlog.TestLogRemark(0, "Thresholds -----")
    cem_thresh_tol = 5 * 0.02 # 2% tolerance

    # DAC and ADC are both 12-bit, so the ADC should mirror the DAC value

```



```
exp_strt_1 = p@HOPE_DG.CEM_1_STRT_THRSH
exp_strt_2 = p@HOPE_DG.CEM_2_STRT_THRSH
exp_strt_3 = p@HOPE_DG.CEM_3_STRT_THRSH
exp_strt_4 = p@HOPE_DG.CEM_4_STRT_THRSH
exp_strt_5 = p@HOPE_DG.CEM_5_STRT_THRSH
```

```
exp_stop_1 = p@HOPE_DG.CEM_1_STOP_THRSH
exp_stop_2 = p@HOPE_DG.CEM_2_STOP_THRSH
exp_stop_3 = p@HOPE_DG.CEM_3_STOP_THRSH
exp_stop_4 = p@HOPE_DG.CEM_4_STOP_THRSH
exp_stop_5 = p@HOPE_DG.CEM_5_STOP_THRSH
```

```
start ../Util/tlm_check("p@HOPE_HK.CEM1_START_LLD_MON" , $exp_strt_1, $cem_thresh_tol, 0)
start ../Util/tlm_check("p@HOPE_HK.CEM2_START_LLD_MON" , $exp_strt_2, $cem_thresh_tol, 0)
start ../Util/tlm_check("p@HOPE_HK.CEM3_START_LLD_MON" , $exp_strt_3, $cem_thresh_tol, 0)
start ../Util/tlm_check("p@HOPE_HK.CEM4_START_LLD_MON" , $exp_strt_4, $cem_thresh_tol, 0)
start ../Util/tlm_check("p@HOPE_HK.CEM5_START_LLD_MON" , $exp_strt_5, $cem_thresh_tol, 0)
wait timeout 3
```

```
start ../Util/tlm_check("p@HOPE_HK.CEM1_STOP_LLD_MON" , $exp_stop_1, $cem_thresh_tol, 0)
start ../Util/tlm_check("p@HOPE_HK.CEM2_STOP_LLD_MON" , $exp_stop_2, $cem_thresh_tol, 0)
start ../Util/tlm_check("p@HOPE_HK.CEM3_STOP_LLD_MON" , $exp_stop_3, $cem_thresh_tol, 0)
start ../Util/tlm_check("p@HOPE_HK.CEM4_STOP_LLD_MON" , $exp_stop_4, $cem_thresh_tol, 0)
start ../Util/tlm_check("p@HOPE_HK.CEM5_STOP_LLD_MON" , $exp_stop_5, $cem_thresh_tol, 0)
wait timeout 3
```

```
# >>> testlog.TestLogRemark(0, "HV - CEM Monitors -----")
```

```
# NOTE: Tolerances set to approximately 0.5% of max
```

```
start ../Util/tlm_check("p@HOPE_HK.CEMB14_VMON" , 0, 45, 0)
start ../Util/tlm_check("p@HOPE_HK.CEMF14_VMON" , 0, 45, 0)
start ../Util/tlm_check("p@HOPE_HK.CEMF14_IMON" , 0, 5, 0)
```

```
start ../Util/tlm_check("p@HOPE_HK.CEMB25_VMON" , 0, 45, 0)
start ../Util/tlm_check("p@HOPE_HK.CEMF25_VMON" , 0, 45, 0)
start ../Util/tlm_check("p@HOPE_HK.CEMF25_IMON" , 0, 5, 0)
```

```
start ../Util/tlm_check("p@HOPE_HK.CEMB3_VMON" , 0, 45, 0)
start ../Util/tlm_check("p@HOPE_HK.CEMF3_VMON" , 0, 45, 0)
start ../Util/tlm_check("p@HOPE_HK.CEMF3_IMON" , 0, 5, 0)
wait timeout 3
```

```
# >>> testlog.TestLogRemark(0, "HV - ESA/TOF Monitors -----")
```

```
# NOTE: Tolerances set to approximately 0.5% of max
```

```
start ../Util/tlm_check("p@HOPE_HK.ESA_NEG_VMON" , 0, 40, 0)
start ../Util/tlm_check("p@HOPE_HK.TOF_NEG_VMON" , 0, 55, 0)
start ../Util/tlm_check("p@HOPE_HK.ESA_V_IN" , 0, 35, 0)
start ../Util/tlm_check("p@HOPE_HK.TOF_V_IN" , 0, 55, 0)
start ../Util/tlm_check("p@HOPE_HK.ESA_POS_VMON" , 0, 40, 0)
start ../Util/tlm_check("p@HOPE_HK.TOF_POS_VMON" , 0, 10, 0)
wait timeout 3
```

```
# >>> testlog.TestLogRemark(0, "LV Monitors -----")
```

```
start ../Util/tlm_check("p@HOPE_HK.AGND" , 0.0 , 0.5, 0)
start ../Util/tlm_check("p@HOPE_HK.POS_5V_CEM_DAC_MON" , 5.0 , 0.5, 0)
start ../Util/tlm_check("p@HOPE_HK.POS_5V_HVPS_DAC_MON" , 5.0 , 0.5, 0)
start ../Util/tlm_check("p@HOPE_HK.POS_5_5V_DAC_MON" , 5.5 , 0.5, 0)
wait timeout 3
```

```

start ../Util/tlm_check("p@HOPE_HK.NEG_12V_MON"           , -12.0 , 1.2, 0)
start ../Util/tlm_check("p@HOPE_HK.POS_5V_MON"           ,   5.0 , 0.5, 0)
start ../Util/tlm_check("p@HOPE_HK.POS_5V_REF_MON"       ,   5.0 , 0.5, 0)
start ../Util/tlm_check("p@HOPE_HK.POS_1_8V"             ,   1.8 , 0.2, 0)
start ../Util/tlm_check("p@HOPE_HK.POS_5V_SW_MON"        ,   5.0 , 0.5, 0)
wait timeout 3

start ../Util/tlm_check("p@HOPE_HK.POS_1_5V_DPF"         ,   1.5 , 0.2, 0)
start ../Util/tlm_check("p@HOPE_HK.POS_1_5V_CEM"         ,   1.5 , 0.2, 0)
start ../Util/tlm_check("p@HOPE_HK.POS_1_5V_SIF"         ,   1.5 , 0.2, 0)
start ../Util/tlm_check("p@HOPE_HK.POS_3_3V"             ,   3.3 , 0.3, 0)
start ../Util/tlm_check("p@HOPE_HK.POS_12V_MON"          ,  12.0 , 1.2, 0)
wait timeout 3

# >>> testlog.TestLogRemark(0, "Door Monitor -----")
start ../Util/tlm_check("p@HOPE_HK.DOOR_FB_PRI_V"        ,   0   , 2   , 0)
start ../Util/tlm_check("p@HOPE_HK.DOOR_FB_SEC_V"        ,   0   , 2   , 0)
wait timeout 3

# -----
# >>> testlog.StartSubTest(0, "Boot to EEPROM")
# -----
# Wait for transition to LVENG
call ../Util/tlm_check("p@HOPE_HK.OP_MODE", "LVENG", 0, 80)

# Turn on diagnostic telemetry
call ../Util/cmd_check("HOPE_SET_PARAMETER", "1, Tlm_Rate_Diag", 0)

# Make sure HV plug is in expected state
ret = call ../Util/tlm_check("p@HOPE_HK.HV_PLUG_ST", $exp_hv_plug, 0, 5)
if (ret != 0) then
    msg "Error: HV Plug not in expected state"
    # >>> testlog.TestLogFail(0, "Error: HV Plug not in expected state")
    wait # indefinite wait until issue is resolved.
endif

# -----
# >>> testlog.StartSubTest(0, "Enable HV")
# -----
call ../Util/cmd_check("HOPE_MODE", "HVENG", 0)
call ../Util/tlm_check("p@HOPE_HK.OP_MODE", "HVENG", 0, 5)

call ../Util/cmd_check("HOPE_HV_EN", "ENABLE", 0)
call ../Util/cmd_check("HOPE_BULK_HVPS_CTRL", "ENABLE, ESA"      , 0)
call ../Util/cmd_check("HOPE_BULK_HVPS_CTRL", "ENABLE, TOF"      , 0)
call ../Util/cmd_check("HOPE_BULK_HVPS_CTRL", "ENABLE, CEM_BK3"   , 0)
call ../Util/cmd_check("HOPE_BULK_HVPS_CTRL", "ENABLE, CEM_BK25" , 0)
call ../Util/cmd_check("HOPE_BULK_HVPS_CTRL", "ENABLE, CEM_BK14" , 0)

call ../Util/tlm_check("p@HOPE_HK.ESA_EN"      , "ON", 0, 0)
call ../Util/tlm_check("p@HOPE_HK.TOF_POS_EN"  , "ON", 0, 0)
call ../Util/tlm_check("p@HOPE_HK.TOF_NEG_EN"  , "ON", 0, 0)
call ../Util/tlm_check("p@HOPE_HK.CEM_BK3_EN"  , "ON", 0, 0)
call ../Util/tlm_check("p@HOPE_HK.CEM_BK25_EN", "ON", 0, 0)
call ../Util/tlm_check("p@HOPE_HK.CEM_BK14_EN", "ON", 0, 0)

```

```

# -----
# >>> testlog.StartSubTest(0, "Ramp CEMs")
# -----
target    = cem_target
step_sz    = 100
dwell     = 2
wait_tmo = ((target/step_sz) * dwell) + 10

cem_dac_tol = (9000/4095) * 2 # 2-bit tolerance
cem_adc_tol = (9000 * .005)  # 0.5% of max

exp_dac = target

plug = p@HOPE_HK.HV_PLUG_ST
if ("FULLENABLE" == plug) then
    exp_adc = target
elseif ("LIMITED" == plug) then
    exp_adc = target / 10
else # DISABLED
    exp_adc = 0
endif

call ../Util/cmd_check("HOPE_CEM_LVL", "BACK_3", $target, $step_sz, $dwell", 0)
call ../Util/tlm_check("p@HOPE_DG.CEM_BK3_LVL" , $exp_dac, $cem_dac_tol, $wait_tmo)
call ../Util/tlm_check("p@HOPE_HK.CEMB3_VMON" , $exp_adc, $cem_adc_tol, 5)

call ../Util/cmd_check("HOPE_CEM_LVL", "BACK_14", $target, $step_sz, $dwell", 0)
call ../Util/tlm_check("p@HOPE_DG.CEM_BK14_LVL" , $exp_dac, $cem_dac_tol, $wait_tmo)
call ../Util/tlm_check("p@HOPE_HK.CEMB14_VMON" , $exp_adc, $cem_adc_tol, 5)

call ../Util/cmd_check("HOPE_CEM_LVL", "BACK_25", $target, $step_sz, $dwell", 0)
call ../Util/tlm_check("p@HOPE_DG.CEM_BK25_LVL" , $exp_dac, $cem_dac_tol, $wait_tmo)
call ../Util/tlm_check("p@HOPE_HK.CEMB25_VMON" , $exp_adc, $cem_adc_tol, 5)

# -----
# >>> testlog.StartSubTest(0, "HVSCI Test")
# -----
call ../Util/cmd_check("HOPE_SCI_PLAN", "7", 0)

call ../Util/cmd_check("HOPE_MODE", "HVSCI", 0)
wait timeout 1

call ../Util/cmd_check("HOPE_SET_PARAMETER", "1, Tlm_Rate_Hk", 0)
call ../Util/cmd_check("HOPE_SET_PARAMETER", "1, Tlm_Rate_Rtsci", 0)

call ../Util/tlm_check("HOPE_DG.PLAN" , , 7, 0, 10)
call ../Util/tlm_check("p@HOPE_HK.OP_MODE", "HVSCI", 0, 10)

# Wait for a HOPE_SI packet to be generated so we have a good SPIN_ID to use
# as a base.

```

```
wait timeout 20
```

```
stim_delay_0 = "_0_NS"
stim_delay_1 = "_25_NS"
stim_delay_2 = "_50_NS"
stim_delay_3 = "_75_NS"
stim_delay_4 = "_100_NS"
stim_delay_5 = "_125_NS"
stim_delay_6 = "_150_NS"
stim_delay_7 = "_175_NS"
```

```
stim_freq_0 = "_2_MHZ"
stim_freq_1 = "_1_MHZ"
stim_freq_2 = "_750_KHZ"
stim_freq_3 = "_500_KHZ"
stim_freq_4 = "_250_KHZ"
stim_freq_5 = "_100_KHZ"
stim_freq_6 = "_75_KHZ"
stim_freq_7 = "_50_KHZ"
```

```
# NOTE: .87 factor assumes nominal acquisition parameters are loaded, which
# creates an acquisition/hv settling ration of .87. If those parameters are
# changed, these counts will need to be tweaked.
```

```
ACQ_RATIO = (4558.0)/(4558+648)
```

```
exp_cnts_0 = 2000000 * ACQ_RATIO
exp_cnts_1 = 1000000 * ACQ_RATIO
exp_cnts_2 = 754700 * ACQ_RATIO # 750kHz stim is actual 754.7kHz
exp_cnts_3 = 500000 * ACQ_RATIO
exp_cnts_4 = 250000 * ACQ_RATIO
exp_cnts_5 = 100000 * ACQ_RATIO
exp_cnts_6 = 75000 * ACQ_RATIO
exp_cnts_7 = 50000 * ACQ_RATIO
```

```
freq = stim_freq_3
# >>> testlog.TestLogRemark(0, "Cycling through TOF at frequency %s" % (freq,))
for dly=0 to 7 loop
    delay = stim_delay_$dly

    # >>> testlog.TestLogRemark(0, "%d/7: Setting Stims to %s, %s" % (dly, freq, delay))
    call ../Util/cmd_check("HOPE_STIM_CTRL", "ENABLE, ENABLE, $delay, $freq, No, 0, 1", 0)
    wait timeout 3
```

```
# make sure all channels are counting as expected.
exp_threshold = $exp_cnts_3 * .001 # 0.1% tolerance
call ../Util/tlm_check("HOPE_RTSCI.PIX1_STARTS", $exp_cnts_3, $exp_threshold, 5)
call ../Util/tlm_check("HOPE_RTSCI.PIX2_STARTS", $exp_cnts_3, $exp_threshold, 5)
call ../Util/tlm_check("HOPE_RTSCI.PIX3_STARTS", $exp_cnts_3, $exp_threshold, 5)
call ../Util/tlm_check("HOPE_RTSCI.PIX4_STARTS", $exp_cnts_3, $exp_threshold, 5)
call ../Util/tlm_check("HOPE_RTSCI.PIX5_STARTS", $exp_cnts_3, $exp_threshold, 5)
call ../Util/tlm_check("HOPE_RTSCI.PIX1_STOPS", $exp_cnts_3, $exp_threshold, 5)
call ../Util/tlm_check("HOPE_RTSCI.PIX2_STOPS", $exp_cnts_3, $exp_threshold, 5)
call ../Util/tlm_check("HOPE_RTSCI.PIX3_STOPS", $exp_cnts_3, $exp_threshold, 5)
call ../Util/tlm_check("HOPE_RTSCI.PIX4_STOPS", $exp_cnts_3, $exp_threshold, 5)
call ../Util/tlm_check("HOPE_RTSCI.PIX5_STOPS", $exp_cnts_3, $exp_threshold, 5)
```

```
# Collect data for 2-3 spins
```

```

curr_spin_id = HOPE_SI.SPIN_ID
wait_spin_id = mod((curr_spin_id + 3), 256)

# >>> testlog.TestLogRemark(0, "Waiting for HOPE_SI.SPIN_ID == %d" % wait_spin_id)
call ../Util/tlm_check("HOPE_SI.SPIN_ID", $wait_spin_id, 0, 40)
endloop

delay = stim_delay_3
# >>> testlog.TestLogRemark(0, "Cycling through frequency at TOF %s" % (delay,))
for frq=0 to 7 loop
    freq = stim_freq_$frq

    # >>> testlog.TestLogRemark(0, "%d/7: Setting Stims to %s, %s" % (frq, freq, delay))
    call ../Util/cmd_check("HOPE_STIM_CTRL", "ENABLE, ENABLE, $delay, $freq, No, 0, 1", 0)

    # make sure all channels are counting as expected.
    exp_threshold = ${exp_cnts_$frq} * .001 # 0.1% tolerance
    call ../Util/tlm_check("HOPE_RTSCI.PIX1_STARTS", ${exp_cnts_$frq}, $exp_threshold, 5)
    call ../Util/tlm_check("HOPE_RTSCI.PIX2_STARTS", ${exp_cnts_$frq}, $exp_threshold, 5)
    call ../Util/tlm_check("HOPE_RTSCI.PIX3_STARTS", ${exp_cnts_$frq}, $exp_threshold, 5)
    call ../Util/tlm_check("HOPE_RTSCI.PIX4_STARTS", ${exp_cnts_$frq}, $exp_threshold, 5)

    call ../Util/tlm_check("HOPE_RTSCI.PIX5_STARTS", ${exp_cnts_$frq}, $exp_threshold, 5)
    call ../Util/tlm_check("HOPE_RTSCI.PIX1_STOPS", ${exp_cnts_$frq}, $exp_threshold, 5)
    call ../Util/tlm_check("HOPE_RTSCI.PIX2_STOPS", ${exp_cnts_$frq}, $exp_threshold, 5)
    call ../Util/tlm_check("HOPE_RTSCI.PIX3_STOPS", ${exp_cnts_$frq}, $exp_threshold, 5)
    call ../Util/tlm_check("HOPE_RTSCI.PIX4_STOPS", ${exp_cnts_$frq}, $exp_threshold, 5)
    call ../Util/tlm_check("HOPE_RTSCI.PIX5_STOPS", ${exp_cnts_$frq}, $exp_threshold, 5)

    # Collect data for 2-3 spins
    curr_spin_id = HOPE_SI.SPIN_ID
    wait_spin_id = mod((curr_spin_id + 3), 256)

    # >>> testlog.TestLogRemark(0, "Waiting for HOPE_SI.SPIN_ID == %d" % wait_spin_id)
    call ../Util/tlm_check("HOPE_SI.SPIN_ID", $wait_spin_id, 0, 40)
endloop

# -----
# >>> testlog.StartSubTest(0, "Ending Test")
# -----
# >>> testlog.TestLogRemark(0, "Transition to HVENG and disable stims")
call ../Util/cmd_check("HOPE_MODE", "HVENG", 0)
call ../Util/tlm_check("p@HOPE_HK.OP_MODE", "HVENG", 0, 0)

call ../Util/cmd_check("HOPE_STIM_CTRL", "DISABLE, DISABLE, _0_NS, _50_KHZ, No, 0, 1", 0)
call ../Util/cmd_check("HOPE_SCI_PLAN", "0", 0)

if (end_state == "PowerOff") then
    # >>> testlog.TestLogRemark(0, "Powering down instrument")

    # >>> testlog.TestLogRemark(0, "Ramp down CEM HV")
    call ../Util/cmd_check("HOPE_CEM_LVL", "BACK_3", 0, 500, 2, 0)

```

```

call ../Util/tlm_check("p@HOPE_DG.CEM_BK3_LVL"      , 0, 5, 35)

call ../Util/cmd_check("HOPE_CEM_LVL", "BACK_14, 0, 500, 2", 0)
call ../Util/tlm_check("p@HOPE_DG.CEM_BK14_LVL"     , 0, 5, 35)

call ../Util/cmd_check("HOPE_CEM_LVL", "BACK_25, 0, 500, 2", 0)
call ../Util/tlm_check("p@HOPE_DG.CEM_BK25_LVL"     , 0, 5, 35)

# >>> testlog.TestLogRemark(0, "Disable HV")
call ../Util/cmd_check("HOPE_BULK_HVPS_CTRL", "DISABLE, CEM_BK14", 0)
call ../Util/cmd_check("HOPE_BULK_HVPS_CTRL", "DISABLE, CEM_BK25", 0)
call ../Util/cmd_check("HOPE_BULK_HVPS_CTRL", "DISABLE, CEM_BK3", 0)
call ../Util/cmd_check("HOPE_BULK_HVPS_CTRL", "DISABLE, TOF"      , 0)
call ../Util/cmd_check("HOPE_BULK_HVPS_CTRL", "DISABLE, ESA"      , 0)
call ../Util/cmd_check("HOPE_HV_EN", "DISABLE", 0)

call ../Util/tlm_check("p@HOPE_HK.ESA_EN"           , "OFF", 0, 0)
call ../Util/tlm_check("p@HOPE_HK.TOF_POS_EN"        , "OFF", 0, 0)
call ../Util/tlm_check("p@HOPE_HK.TOF_NEG_EN"        , "OFF", 0, 0)
call ../Util/tlm_check("p@HOPE_HK.CEM_BK3_EN"        , "OFF", 0, 0)
call ../Util/tlm_check("p@HOPE_HK.CEM_BK25_EN"       , "OFF", 0, 0)
call ../Util/tlm_check("p@HOPE_HK.CEM_BK14_EN"       , "OFF", 0, 0)

# >>> testlog.TestLogRemark(0, "Transition to LVENG")
call ../Util/cmd_check("HOPE_MODE", "LVENG", 0)
call ../Util/tlm_check("p@HOPE_HK.OP_MODE", "LVENG" , 0, 0)

# >>> testlog.TestLogRemark(0, "Power off...")
@SCE_CTRL_DISABLE_INSTR_A_POWER
msgbox "Disable Instrument Power"
else
# >>> testlog.TestLogRemark(0, "Leaving instrument powered and ready for science mode")
endif

# >>> testlog.EndTestLog("IFTP")
# >>> testlog.EndTestSeries()
# >>> testlog.CloseTestLog(testlog.LogFile)
eop:
end

# Script }}}--

```

1.8.2 Sample 2

```

script cmd_check(cmd_mnemonic, params, print_pass)
begin
    goto skip_hdr

# *****
# -----
# Name           : $RCSfile$
# Project        : RBSP-HOPE
#
# -----
# Copyright 2009, Southwest Research Institute

```



```

# -----
# File Description:
#
# Issue a command and verify that it is executed
#
# Parameters:
#   cmd_mnemonic - the command mnemonic to send (string)
#   params       - a string containing the comma-seperated paramter list
#   print_pass   - 0: Don't print messages for successful commands
#                 1: print messages for successful commands
# -----
# *****/

skip_hdr:

# >>> import re
# >>> from Instruments.ECT.HOPE.Util import testlog as testlog

local prev_cmd_exe_cnt = HOPE_HK.CMD_EXE_CNT
local prev_cmd_rej_cnt = HOPE_HK.CMD_REJ_CNT

# -----
# Issue the command

send $cmd_mnemonic($params)

# -----
# Wait for the exe counter to increment
test_label = "cmd_ok_check1"

exp_cmd_exe_cnt = mod((prev_cmd_exe_cnt + 1), 256)
exp_cmd_rej_cnt = prev_cmd_rej_cnt

# Special case for CLR_LATCHED command, when the 2nd parameter is set (clear
# command counters), we need to override the expected command counter values.

# >>> if (      (cmd_mnemonic == "HOPE_CLR_LATCHED")
# >>>      and (re.match("^[^,]*, *1", str(params)))):
# >>>     exp_cmd_exe_cnt = 1
# >>>     exp_cmd_rej_cnt = 0

test_fail = 1
wait (      (HOPE_HK.CMD_REJ_CNT == exp_cmd_rej_cnt) ;;
      && (HOPE_HK.CMD_EXE_CNT == exp_cmd_exe_cnt));;
      timeout 10 then
        test_fail = 0
      else
        test_fail = 1
    endwait

cmd_ok_check1:
    cmd_rej_cnt = HOPE_HK.CMD_REJ_CNT
    cmd_exe_cnt = HOPE_HK.CMD_EXE_CNT

```

```

if ((test_fail == 0) && (print_pass == 1)) then
    msg "HOPE_FPT:: TPASS:: HOPE_HK.CMD_EXE_CNT = $cmd_exe_cnt (exp: $exp_cmd_exe_cnt)"
    msg "HOPE_FPT:: TPASS:: HOPE_HK.CMD_REJ_CNT = $cmd_rej_cnt (exp: $exp_cmd_rej_cnt)"
    # >>> testlog.TestLogOk(0, "%-25s = %s (exp: %s)" % ("HOPE_HK.CMD_EXE_CNT", cmd_exe_cnt,
exp_cmd_exe_cnt))
    # >>> testlog.TestLogOk(0, "%-25s = %s (exp: %s)" % ("HOPE_HK.CMD_REJ_CNT", cmd_rej_cnt,
exp_cmd_rej_cnt))
endif

if (test_fail != 0) then
    msg "HOPE_FPT:: TFAIL:: HOPE_HK.CMD_EXE_CNT = $cmd_exe_cnt (exp: $exp_cmd_exe_cnt)"
    msg "HOPE_FPT:: TFAIL:: HOPE_HK.CMD_REJ_CNT = $cmd_rej_cnt (exp: $exp_cmd_rej_cnt)"
    # >>> testlog.TestLogFail(0, "%-25s = %s (exp: %s)" % ("HOPE_HK.CMD_EXE_CNT", cmd_exe_cnt,
exp_cmd_exe_cnt))
    # >>> testlog.TestLogFail(0, "%-25s = %s (exp: %s)" % ("HOPE_HK.CMD_REJ_CNT", cmd_rej_cnt,
exp_cmd_rej_cnt))
    wait
endif

ret = call opcode_check($cmd_mnemonic, $print_pass)
test_fail = test_fail + ret

eop:
    return(test_fail)
end

```

1.8.3 Sample 3

```

script opcode_check(cmd_mnemonic, print_pass)
begin
    goto skip_hdr

# *****
# -----
# Name          : $RCSfile$
# Project       : IBEX
# Original Author: Patrick Noonan (pnoonan@swri.edu)
#
# Last Modified $Author: gdunn $
# $Date: 2007-08-30 11:46:53 -0500 (Thu, 30 Aug 2007) $
#
# -----
# Copyright 2006, Southwest Research Institute
# -----
# File Description:
#
# opcode_check(command, ...)
#   Check opcode of last command executed
#
# Parameters:
#   cmd_mnemonic - the expected value of the 'LAST_OPCODE' field
#   print_pass   - 0: Don't print messages for successful commands
#                  1: print messages for successful commands
# -----
# *****/

```

```
skip_hdr:

# >>> import re
# >>> from GseosConversion import EU
# >>> from Instruments.ECT.HOPE.Util import testlog as testlog

# The value reported in hk for last_opcode omits the HOPE_ prefix, so strip it
# off of the passed in cmd_mnemonic

# >>> cmd_mnemonic = re.sub("^HOPE_", "", cmd_mnemonic)

test_fail = 1

wait p@HOPE_HK.LAST_OPCODE == cmd_mnemonic timeout 10 then
    test_fail = 0
else
    test_fail = 1
endwait

last_opcode = p@HOPE_HK.LAST_OPCODE
if (test_fail == 0) then
    if (print_pass == 1) then
        msg "HOPE_FPT:: TPASS:: HOPE_HK.LAST_OPCODE = $cmd_mnemonic"
        # >>> testlog.TestLogOk(0, "HOPE_HK.LAST_OPCODE = %s" % cmd_mnemonic)
    endif
else
    msg "HOPE_FPT:: TFAIL:: HOPE_HK.LAST_OPCODE != $cmd_mnemonic (act: $last_opcode)"
    # >>> testlog.TestLogFail(0, "HOPE_HK.LAST_OPCODE != %s (act: %s)" % (cmd_mnemonic,
last_opcode))
    wait
endif

eop:
    return(test_fail)
end
```

Index

- D -

Dialog 5

- E -

Embedded Python 22

Error 23, 24

Error Handler 23, 24

- G -

GUI 5

- I -

Introduction 3

- L -

Language 8

Language Reference 8

- M -

Main 2

- P -

Python 22

- S -

Stol 2

Stol Dialog 5

STOL Language 8

- U -

User Interface 5

Endnotes 2... (after index)

Back Cover