Ball Aerospace COSMOS

Navigate the docs...

Welcome

Improve this page

This site aims to be a comprehensive guide to COSMOS. We'll cover topics such as getting your configuration up and running, developing test and operations scripts, building custom telemetry screens, and give you some advice on participating in the future development of COSMOS itself.

So what is Ball Aerospace COSMOS, exactly?

COSMOS is a set of 15 applications that can be used to control a set of embedded systems. These systems can be anything from test equipment (power supplies, oscilloscopes, switched power strips, UPS devices, etc), to development boards (Arduinos, Raspberry Pi, Beaglebone, etc), to satellites.

Helpful Hints

Throughout this guide there are a number of small-but-handy pieces of information that can make using COSMOS easier, more interesting, and less hazardous. Here's what to look out for.

ProTips™ help you get more from COSMOS
These are tips and tricks that will help you be a COSMOS wizard!

Notes are handy pieces of information
These are for the extra tidbits sometimes necessary to understand COSMOS.

Warnings help you not blow things up
Be aware of these messages if you wish to avoid certain death.

You'll see this by a feature that hasn't been released
Some pieces of this website are for future versions of COSMOS that are not yet released.

If you come across anything along the way that we haven't covered, or if you know of a tip you think others would find handy, please file an issue and we'll see about including it in this guide.

Installation

Improve this page

Windows 7+

Run the COSMOS Installation bat file:

1. Goto this link: INSTALL_COSMOS.bat
2. Choose Save As... in your browser to save the file to your harddrive
3. Run the bat from from Windows explorer or a cmd window

NOTE: The COSMOS installation batch file downloads all the components of the COSMOS system from the Internet. If you want to create an offline installer simply zip up the resulting installation directory. Then manually create the COSMOS_DIR environment variable to point to the root directory where you unzip all the installation files. You might also want to add <COSMOS>\Vendor\Ruby\bin to your path to allow access to Ruby from your terminal.

CentOS Linux 6.5/6.6/7 and Mac OSX Mavericks+

The following instructions work for an installation on CentOS Linux 6.5, 6.6, or 7 from a clean install or any version of Mac OSX after and include Mavericks. Similar steps should work on other distributions/versions, particularly Redhat. Support for Ubuntu is coming soon.

Run the following command in a terminal:

```
bash <(\curl -sSL https://raw.githubusercontent.com/BallAerospace/COSMOS/master/vendor/installers/linux_mac/INSTALL_COSMOS.sh)
```

Linux Notes

The install script will install all needed dependencies using the system package manager and install ruby using rbenv. If another path to installing COSMOS is desired please feel free to just use the INSTALL_COSMOS.sh file as a basis. As always, it is a good idea to review any remote shell script before executing it on your system.

Mac Notes

The install script will install all needed dependencies using homebrew and install ruby using rbenv. If another path to installing COSMOS is desired please feel free to just use the INSTALL_COSMOS.sh file as a basis. As always, it is a good idea to review any remote shell script before executing it on your system.

In the tools/mac folder is a Mac application version of each tool. Launcher.app can be copied into the overall Mac applications folder or the Desktop for easy launching. For this to work you need to set an environment variable for each user so that COSMOS can find it configuration files:

In your .bash_profile add this line (point to your actual COSMOS configuration folder):

```
export COSMOS_USERPATH=/Users/username/demo
```

General

Notes:

1. ruby-termios is a dependency of COSMOS on non-windows platforms but is not listed in the gem dependencies because it is not a dependency on Windows. An extension attempts to install it when gem install cosmos is run. This should work as long as you are online. If attempting an offline installation of cosmos you will need to first manually install ruby-termios: `gem install ruby-termios`
2. Installing the COSMOS gem (and many other binary gems) with rdoc 4.0.0 spits outs warnings like this:

```
unable to convert "\x90" from ASCII-8BIT to UTF-8 for lib/cosmos/ext/array.so, skipping
```

These are just warnings and can be safely ignored. Updating rdoc before installing cosmos will remove the warnings: gem install rdoc.

BACK NEXT

Directory structure

Improve this page

Configuring COSMOS for your hardware unlocks all of its functionality for your system.

COSMOS Configuration is organized into the following directory structure with all files having a well defined location:

```
.
├── Gemfile
├── Launcher
├── Launcher.bat
├── Rakefile
├── userpath.txt
├── config
|   ├── data
|   ├── system
|   ├── targets
|   |   ├── TARGETNAME
|   |   |   ├── cmd_tlm_server.txt
|   |   |   ├── target.txt
|   |   |   ├── cmdtlm
|   |   |   ├── lib
|   |   |   └── screens
|   |   └── ...
|   └── tools
|   |   ├── cmd_tlm_server
|   |   └── ...
├── lib
├── outputs
|   ├── handbooks
|   ├── logs
|   ├── saved_config
|   ├── tables
|   └── tmp
├── procedures
├── tools
|   ├── mac
|   └── ...
```

An overview of what each of these does:

FILE / DIRECTORY	DESCRIPTION
	Defines the gems and their versions used by your COSMOS configuration. If you want to use other

Gemfile	gems in your cosmos project you should add them here and then run "bundle install" from the command line. See the Bundler documents for more information.
Launcher	A small script used to launch the COSMOS Launcher application. ruby Launcher
Launcher.bat	Windows batch file used to launch the COSMOS Launcher application from Windows explorer.
Rakefile	The Rakefile contains user modifiable scripts to perform common tasks using the ruby rake application. By default it comes with a script that is used to calculate CRCs over the project's files. rake crc
userpath.txt	This is a special file used by COSMOS to determine the root folder of a COSMOS configuration. DO NOT DELETE.
config	The config folder contains all of the configuration necessary for a COSMOS project.
config/ data	The config/data folder contains data shared between applications such as images. It also contains the crc.txt file that holds the expected CRCs for each of the configurations files.
config/ system	The config/system folder contains system.txt one of the first files you may need to edit for you COSMOS configuration. system.txt contains settings common to all of the COSMOS applications. It is also defines the targets that make up your COSMOS configuration. See System Configuration for all the details.
config/ targets	The config/targets folder contains the configuration for each target that is to be commanded or receive telemetry (data) from in a COSMOS configuration. Target folders should be named after the name of the target and be ALL CAPS.
config/ targets/ TARGETNAME	config/targets/TARGETNAME folders contains the configuration for a target that is to be commanded or receive telemetry (data) from in a COSMOS configuration. Target folders should be named after the name of the target and be ALL CAPS.
config/ targets/ TARGETNAME/ cmd_tlm_server.txt	config/targets/TARGETNAME/cmd_tlm_server.txt contains a snippet of the configuration for the COSMOS Command and Telemetry Server that defines how to interface with the specific target. See Interface Configuration for more information.
config/ targets/ TARGETNAME/ target.txt	config/targets/TARGETNAME/target.txt contains target specific configuration such as which command parameters should be ignored by Command Sender. See System Configuration for more information.
config/ targets/ TARGETNAME/ cmdtlm	config/targets/TARGETNAME/cmdtlm contains command and telemetry definition files for the target. See Command and Telemetry Configuration for more information.
config/ targets/ TARGETNAME/ lib	config/targets/TARGETNAME/lib contains any custom code required by the target. Often this includes a custom Interface class. See Interfaces for more information.
config/ targets/ TARGETNAME/ screens	config/targets/TARGETNAME/screens contains telemetry screens for the target. See Screen Configuration for more information.
config/ tools	config/tools contains configuration files for the COSMOS applications. Most tools support configuration but do not require it. See Tool Configuration for more information.
config/ tools/ cmd_tlm_server	config/tools/cmd_tlm_server contains the configuration file for the COSMOS Command and Telemetry Server (by default cmd_tlm_server.txt). This file defines how to connect to each target in the COSMOS configuration. See System Configuration for more information.
lib	The lib folder contains shared custom code written for the COSMOS configuration.

outputs	The outputs folder contains all files generated by COSMOS applications.
outputs/ handbooks	The outputs/handbooks folder contains command and telemetry handbooks generated by Handbook Creator.
outputs/ logs	The outputs/logs folder contains packet and message logs.
outputs/ saved_config	The outputs/saved_config folder contains configuration saved by COSMOS. Every time COSMOS runs it saves the current configuration into this folder. Saved configurations are used to enable reading back old packet log files that may have been generated with a different packet configuration than the current configuration.
outputs/ tables	The outputs/tables folder contains table files generated by Table Manager.
outputs/ tmp	The outputs/tmp folder contains temporary files generated by the COSMOS tools. These are typically cache files to improve performance. They may be safely deleted at any time.
procedures	The procedures folder is the default location for storing COSMOS test and operations procedures.
tools	The tools folder contains the scripts used to launch each of the COSMOS tools.
tools/ mac	The tools/mac folder contains Mac application bundles used to launch the COSMOS tools on Mac computers.

BACK NEXT

Getting Started

Improve this page

Welcome to the COSMOS system... Let's get started! This guide is a high level overview that will help with setting up your first COSMOS project.

1. Get COSMOS Installed onto your computer by following the Installation Guide.
   - You should now have COSMOS installed and a Demo project available that we can make changes to.
2. Start the COSMOS Launcher in the Demo project
   - Using a terminal or cmd shell change directories to the demo/tools folder and run: `ruby Launcher`
3. Accept the legal dialog, and then click the Command and Telemetry Server button in the launcher.
   - The COSMOS Command and Telemetry Server will start up. This tool provides all the real-time functionality for the COSMOS System by connecting to each "target" in the system. Targets are external systems that receive commands and generate telemetry, often over ethernet or serial connections. The Command and Telemetry Server is the hub through which commands are sent and telemetry is received. It also logs all commands and telemetry and performs limits monitoring.
4. Experiment with launching other COSMOS tools.
   - Use Command Sender to send individual commands.
   - Use Limits Monitor to watch for telemetry limits violations
   - Run some of the example scripts in Script Runner and Test Runner
   - View individual Telemetry packets in Packet Viewer
   - View detailed telemetry displays in Telemetry Viewer
   - Graph some data in Telemetry Grapher
   - View log type data in Data Viewer
   - Process Log files with Telemetry Extractor and Command Extractor
   - Create command and telemetry handbooks with Handbook Creator
   - Edit binary files with Table Manager
   - Replay logged telemetry with Replay (requires shutting down the Command and Telemetry Server first)

Interfacing with Your Hardware

Playing with the COSMOS Demo is fun and all, but now you want to talk to your own real hardware? Let's do it!

1.. The first step is to create a "target folder" for your new target. At a minimum this folder will contain all the information defining the packets (command and telemetry) that are needed to communicate with your hardware.

- Inside your demo area, create a folder for your target in config/targets/. The folder name should be ALL CAPS and concise. Let's pretend we're going to interface with custom piece of software you wrote called BOB, so we'll call the folder config/targets/BOB.

2.. Next we need to define the commands and telemetry packets for our target. The details on the command and telemetry definition file formats can be found here: Command and Telemetry Configuration

- Create the folder config/targets/BOB/cmd_tlm
- Create a new text file called config/targets/BOB/cmd_tlm/bob_cmds.txt with the following contents:

```
COMMAND BOB COLLECT BIG_ENDIAN "Collect temperatures"
  APPEND_PARAMETER LENGTH 32 UINT 0 1024 5 "Packet Length"
  APPEND_ID_PARAMETER CMD_ID 8 UINT 1 1 1 "Command Id"
  APPEND_PARAMETER MODE 32 INT 0 1 0 "Temperature Collection Mode"
    STATE NORMAL 0
    STATE FAST 1
```

- Woah, what did we just do!
  - We created a COMMAND for target BOB named COLLECT.
  - The command is made up of BIG_ENDIAN parameters and is described by "Collect temperatures". Here we are using the append flavor of defining parameters which stacks them back to back as it builds up the packet and you don't have to worry about defining the bit offset into the packet.
  - First we APPEND_PARAMETER a parameter called LENGTH that is a 32-bit unsigned integer (UINT) that has a minimum value of 0, a maximum value of 1024, and a default value of 5.
  - Then we APPEND_ID_PARAMETER a parameter that is used to identify the packet called CMD_ID that is an 8-bit unsigned integer (UINT) with a minimum value of 1, a maximum value of 1, and a default value of 1, that is described as the "Command Id".
  - Then we APPEND_PARAMETER a third parameter called MODE which is a 32-bit integer (INT) with a minimum value of 0, a maximum value of 1, and a default value of 0, that is described as the "Temperature Collection Mode". MODE has two states which are just a fancy way of giving meaning to the integer values 0 and 1. The STATE NORMAL has a value of 0 and the STATE FAST has a value of 1.
- In summary we defined a 72-bit command packet made up of three parameters, LENGTH which tells us the length of the packet in bytes not included itself, CMD_ID which is used to identify the command, and MODE which has two values NORMAL and FAST.
- Onto telemetry, Create a new text file called config/targets/BOB/cmd_tlm/bob_tlm.txt with the following contents:

```
TELEMETRY BOB TEMPS BIG_ENDIAN "Temperature Telemetry"
  ITEM LENGTH 0 32 UINT "Packet Length"
  ID_ITEM TLM_ID 32 32 INT 3 "Message Identifier"
  ITEM TEMP1 64 32 FLOAT "Temperature 1"
  ITEM TEMP2 96 32 FLOAT "Temperature 2"
```

- This time we created a TELEMETRY packet for target BOB called TEMPS that contains BIG_ENDIAN items and is described as "Temperature Telemetry". Unlike above, in this example I am not using the APPEND flavor of defining items so each item contains both a bit offset and a bit size. In general, if creating configuration files by hand I recommend using the APPEND versions as they are much easier to maintain.
  - So we start by defining an item called LENGTH at bit offset 0 with a bit size of 32 bits of type UINT (unsigned integer) described as "Packet Length".
  - Next an ID_ITEM called TLM_ID at bit offset 32 with a bit size of 32 bits of type INT (integer) with an id value of 3 and described as "Message Identifier". Id items are used to take unidentified blobs of bytes and determine which packet they are. In this case if a blob comes in with a value of 3 at bit offset 32 interpreted as a 32-bit integer then this packet will be "identified".
  - Next we define two items that are temperatures. The first at bit offset 64 that is a 32-bit FLOAT and the second at bit offset 96 which is also a 32-bit float.

3.. We have successfully defined the commands and telemetry packets for our target. Most targets will obviously have more than one command and one telemetry packet. Before we move on, now is a great time to look at the contents of some of the other target folders in config/target that come with COSMOS. They provide good examples of what the configuration for other types of targets might look like and use a lot of the available keywords for the configuration files.

4.. Next we need to tell COSMOS that our new target BOB exists. We do that in the config/system/system.txt file. Edit this file and add the following line. See System Configuration Guide:

```
DECLARE_TARGET BOB
```

- This tells COSMOS to look for a folder called BOB in config/targets.

5.. Now we need to configure how to communicate with BOB. BOB is acting as a TCP/IP server at 192.168.1.5 and is listening on port 8888. We tell COSMOS how to talk to it by adding the following snippet to config/tools/cmd_tlm_server/cmd_tlm_server.txt. See System Configuration Guide:

```
INTERFACE BOB_INT tcpip_client_interface.rb 192.168.1.5 8888 8888 5.0 nil LENGTH 0 32 4
  TARGET BOB
```

- This tells COSMOS there is a new INTERFACE called BOB_INT that will connect as a TCP/IP client using the code in tcpip_client_interface.rb to address 192.168.1.5 using port 8888 for both reading and writing. It also has a write timeout of 5 seconds, reads will never timeout (nil). The TCP/IP stream will be interpreted using the COSMOS LENGTH protocol with the length field found at bit offset 0 with bit size of 32-bits and a value offset of 4 bytes (because the value in the length field does not include itself). For all the details on how to configure COSMOS interfaces please see the Interface Guide. The TARGET BOB line tells COSMOS that it will receive telemetry from and send commands to BOB using the BOB_INT interface.

6.. COSMOS is now fully configured with everything needed to talk to our new target. Other things you might like to do at this point is define telemetry screens in config/targets/BOB/screens. See Telemetry Screen Configuration. Configure LENGTH and CMD_ID as IGNORED_PARAMETER in config/targets/BOB/target.txt.

7.. That's all there is to it! In 14 lines of configuration we now have a fully configured system that is capable of connecting to, receiving telemetry from, sending commands to, displaying/graphing/logging data from our new target!

BACK NEXT

Scripting Guide

Improve this page

Table of Contents

This document provides the information necessary to write test procedures using the COSMOS scripting API. Scripting in COSMOS is designed to be simple and intuitive. The code completion ability for command and telemetry mnemonics makes Script Runner the ideal place to write your procedures, however any text editor will do. If there is functionality that you don't see here or perhaps an easier syntax for doing something, please submit a ticket.

Concepts

Ruby Programming Language

COSMOS scripting is implemented using the Ruby Programming language. This should be largely transparent to the user, but if advanced processing needs to be done such as writing files, then knowledge of Ruby is necessary. Please see the Ruby Guide for more information about Ruby.

A basic summary of Ruby:

1. There is no 80 character limit on line length. Lines can be as long as you like, but be careful to not make them too long as it makes printed reviews of scripts more difficult.
2. Variables do not have to be declared ahead of time and can be reassigned later, i.e. Ruby is dynamically typed.
3. Variable values can be placed into strings using the "#{variable}" syntax. This is called variable interpolation.
4. A variable declared inside of a block or loop will not exist outside of that block unless it was already declared (see Ruby's variable scoping for more information).

The Ruby programming language provides a script writer a lot of power. But with great power comes great responsibility. Remember when writing your scripts that you or someone else will come along later and need to understand them. Therefore use the following style guidelines:

- Use two spaces for indentation and do NOT use tabs
- Constants should be all caps with underscores
  - `SPEED_OF_LIGHT = 299792458 # meters per s`
- Variable names and method names should be in lowercase with underscores
  - `last_name = "Smith"`
  - `perform_setup_operation()`
- Class names (when used) should be camel case and the files which contain them should match but be lowercase with underscores
  - `class DataUploader # in 'data_uploader.rb'`
  - `class CcsdsUtility # in 'ccsds_utility.rb'`
- Don't add useless comments but instead describe intent

The following is an example of good style:

Example Code

```
#####################################
# Title block which describes the test
# Author: John Doe
# Date: 7/27/2007
#####################################

load 'upload_utility.rb' # library we don't want to show executing
require_utility 'helper_utility' # library we do want to show executing

# Declare constants
OUR_TARGETS = ['INST','INST2']

# Clear the collect counter of the passed in target name
def clear_collects(target)
  cmd("#{target} CLEAR")
  wait_check("#{target} HEALTH_STATUS COLLECTS == 0", 5)
end

#####################################
# START
#####################################
helper = HelperUtility.new
helper.setup

# Perform collects on all the targets
OUR_TARGETS.each do |target|
  collects = tlm("#{target} HEALTH_STATUS COLLECTS")
  cmd("#{target} COLLECT with TYPE SPECIAL")
  wait_check("#{target} HEALTH_STATUS COLLECTS == #{collects + 1}", 5)
end

clear_collects('INST')
clear_collects('INST2')
```

This example shows several features of COSMOS scripting in action. Notice the difference between 'load' and 'require_utility'. The first is to load additional scripts which will NOT be shown in Script Runner when executing. This is a good place to put code which takes a long time to run such as image analysis or other looping code where you just want the output. 'require_utility' will visually execute the code line by line to show the user what is happening.

Next we declare our constants and create an array of strings which we store in OUR_TARGETS. Notice the constant is all uppercase with underscores.

Then we declare our local methods of which we have one called clear_collects. Please provide a comment at the beginning of each method describing what it does and the parameters that it takes.

The 'helper_utility' is then created by HelperUtility.new. Note the similarity in the class name and the file name we required.

The collect example shows how you can iterate over the array of strings we previously created and use variables when commanding and checking telemetry. The pound bracket #{} notation puts whatever the variable holds inside the #{} into the string. You can even execute additional code inside the #{} like we do when checking for the collect count to increment.

Finally we call our clear_collects method on each target by passing the target name. You'll notice there we used single quotes instead of double quotes. The only difference is that double quotes allow for the #{} syntax and support escape

characters like newlines (\n) while single quotes do not. Otherwise it's just a personal style preference.

Telemetry Types

There are four different ways that telemetry values can be retrieved in COSMOS. The following chart explains their differences.

TELEMETRY TYPE	DESCRIPTION
Raw	Raw telemetry is exactly as it is in the telemetry packet before any conversions. All telemetry items will have a raw value except for Derived telemetry points which have no real location in a packet. Requesting raw telemetry on a derived item will return nil.
Converted	Converted telemetry is raw telemetry that has gone through a conversion factor such as a state conversion or a polynomial conversion. If a telemetry item does not have a conversion defined, then converted telemetry will be the same as raw telemetry. This is the most common type of telemety used in scripts.
Formatted	Formatted telemetry is converted telemetry that has gone through a printf style conversion into a string. Formatted telemetry will always have a string representation. If no format string is defined for a telemetry point, then formatted telemetry will be the same as converted telemetry except represented as string.
Formatted with Units	Formatted with Units telemetry is the same as Formatted telemetry except that a space and the units of the telemetry item are appended to the end of the string. If no units are defined for a telemetry item then this type is the same as Formatted telemetry.

Writing Test Procedures

Using Subroutines

Subroutines in COSMOS scripting are first class citizens. They can allow you to perform repetitive tasks without copying the same code multiple times and in multiple different test procedures. This reduces errors and makes your test procedures more maintainable. For example, if multiple test procedures need to turn on a power supply and check telemetry, they can both use a common subroutine. If a change needs to be made to how the power supply is turned on, then it only has to be done in one location and all test procedures reap the benefits. No need to worry about forgetting to update one. Additionally using subroutines allows your high level procedure to read very cleanly and makes it easier for others to review. See the Subroutine Example example.

Example Test Procedures

Subroutines

```ruby
# My Utility Procedure: program_utilities.rb
# Author: Bob

#####################################################################
# Define helpful subroutines useful by multiple test procedures
#####################################################################

# This subroutine will put the instrument into safe mode
def goto_safe_mode
  cmd("INST SAFE")
  wait_check("INST SOH MODE == 'SAFE'", 30)
  check("INST SOH VOLTS1 < 1.0")
  check("INST SOH TEMP1 > 20.0")
  puts("The instrument is in SAFE mode")
end

# This subroutine will put the instrument into run mode
def goto_run_mode
  cmd("INST RUN")
  wait_check("INST SOH MODE == 'RUN'", 30)
  check("INST SOH VOLTS1 > 27.0")
  check("INST SOH TEMP1 > 20.0")
  puts("The instrument is in RUN mode")
end

# This subroutine will turn on the power supply
def turn_on_power
  cmd("GSE POWERON")
  wait_check("GSE SOH VOLTAGE > 27.0")
  check("GSE SOH CURRENT < 2.0")
  puts("WARNING: Power supply is ON!")
end

# This subroutine will turn off the power supply
def turn_off_power
  cmd("GSE POWEROFF")
  wait_check("GSE SOH VOLTAGE < 1.0")
  check("GSE SOH CURRENT < 0.1")
  puts("Power supply is OFF")
end

# My Test Procedure: run_instrument.rb
# Author: Larry

require_utility("program_utilities.rb")

turn_on_power()
goto_run_mode()

# Perform unique tests here

goto_safe_mode()
turn_off_power()
```

Ruby Control Structures

```
#if, elsif, else structure

x = 3

if tlm("INST HEALTH_STATUS COLLECTS") > 5
  puts "More than 5 collects!"
elsif (x == 4)
  puts "variable equals 4!"
else
  puts "Nothing interesting going on"
end

#Endless loop and single-line if

loop do
  break if tlm("INST HEALTH_STATUS TEMP1") > 25.0
  wait(1)
end

#Do something a given number of times

5.times do
  cmd("INST COLLECT")
end
```

Iterating over similarly named telemetry points

```
#This block of code goes through the range of numbers 1 through 4 (1..4)
#and checks telemetry items TEMP1, TEMP2, TEMP3, and TEMP4

(1..4).each do |num|
  check("INST HEALTH_STATUS TEMP#{num} > 25.0")
end

#You could also do

num = 1
4.times do
  check("INST HEALTH_STATUS TEMP#{num} > 25.0")
  num = num + 1
end
```

Prompting for User Input

```
numloops = ask("Please enter the number of times to loop")

numloops.times do
  puts "Looping"
end
```

Running Test Procedures

Execution Environment

Using Script Runner

Script Runner is a graphical application that provides the ideal environment for running and implementing your test procedures. The Script Runner tool is broken into 4 main sections. At the top of the tool is a menu bar that allows you to do such things as open and save files, comment out blocks of code, perform a syntax check, and execute your script.

Next is a tool bar that displays the currently executing line number of the script and three buttons, "Go", "Pause/Resume?", and "Stop". The Go button is used to skip wait statements within the script. This is sometimes useful if an excessive wait statement is added to a script. The Pause/Resume? button will pause the executing script and display the next line that will be executed. Resume will resume execution of the script. The Resume button is also used to continue script execution after an exception occurs such as trying to send a command with a parameter that is out of range. Finally, the Stop button will stop the executing script at any time.

Third is the display of the actual script. While the script is not running, you may edit and compose scripts in this area. A handy code completion feature is provided that will list out the available commands or telemetry points as you are writing your script. Simply begin writing a cmd(or tlm(line to bring up code completion. This feature greatly reduces typos in command and telemetry mnemonics.

Finally, displayed is the script output. All commands that are sent, errors that occur, and user puts statements appear in this output section. Additionally anything printed into this section is logged by Script Runner into your projects COSMOS user area.

From the Command Line

Note that any COSMOS script can also be run from the command line if the script begins with the following two lines:

```
require 'cosmos'
require 'cosmos/script'
```

The Script Runner Tool automatically executes these lines for you so they aren't required for scripts that will only be run from Script Runner. Nice features such as display of the current line or the ability to pause a script are not available from the command line.

Test Procedure API

The following methods are designed to be used in test procedures. However, they can also be used in custom built COSMOS tools. Please see the COSMOS Tool API section for methods that are more efficient to use in custom tools.

Retrieving User Input

These methods allow the user to enter values that are needed by the script.

ask

The ask method prompts the user for input with a question. User input is automatically converted from a string to the appropriate data type. For example if the user enters "1", the number 1 as an integer will be returned.

Syntax: `ask("<question>")`

PARAMETER	DESCRIPTION
question	Question to prompt the user with.
allow_blank	Whether or not to allow empty responses (optional - defaults to false)

Example:

```
value = ask("Enter an integer")
value = ask("Enter a value or nothing", true)
```

ask_string

The ask_string method prompts the user for input with a question. User input is always returned as a string. For exampe if the user enters "1", the string "1" will be returned.

Syntax: `ask_string("<question>")`

PARAMETER	DESCRIPTION
question	Question to prompt the user with.
allow_blank	Whether or not to allow empty responses (optional - defaults to false)

Example:

```
string = ask_string("Enter a String")
string = ask_string("Enter a value or nothing", true)
```

message_box, vertical_message_box (COSMOS 3.5.0+), combo_box (COSMOS 3.5.0+)

The message_box, vertical_message_box, and combo_box methods create a message box with arbitrary buttons or selections that the user can click. The text of the button clicked is returned.

Syntax:

```
message_box("<message>", "<button text 1>", ...)
vertical_message_box("<message>", "<button text 1>", ...)
combo_box("<message>", "<selection text 1>", ...)
```

PARAMETER	DESCRIPTION
message	Message to prompt the user with.
button/selection text #x	Text for a button or selection

Example:

```
value = message_box("Press OK to continue or CANCEL to cancel", 'OK', 'CANCEL')
value = vertical_message_box("Press OK to continue or CANCEL to cancel", 'OK', 'CANCEL')
value = combo_box("Select OK to continue or CANCEL to cancel", 'OK', 'CANCEL')
if value == 'OK'
  puts 'OK Pressed'
else
  puts 'CANCEL Pressed'
end
```

Providing information to the user

These methods notify the user that something has occurred.

prompt

The prompt method displays a message to the user and waits for them to press an ok button.

Syntax: `prompt("<message>")`

PARAMETER	DESCRIPTION
message	Message to prompt the user with.

Example:

```
prompt("Press OK to continue")
```

status_bar

The status_bar method displays a message to the user in the status bar (at the bottom of the tool).

Syntax: `status_bar("<message>")`

PARAMETER	DESCRIPTION
message	Message to display in the status bar

Example:

```
status_bar("Connection Successful")
```

play_wav_file

The play_wav_file method plays the provided wav file once. Note that the script will proceed while the wav file plays.

Syntax: `play_wav_file(wav_filename)`

PARAMETER	DESCRIPTION
wav_filename	Path and filename of the wav file to play.

Example:

```
play_wav_file("config/data/alarm.wav")
```

Commands

These methods provide capability to send commands to a target and receive information about commands in the system.

cmd

The cmd method sends a specified command.

Syntax:

```
cmd("<Target Name> <Command Name> with <Param #1 Name> <Param #1 Value>, <Param #2 Name> <Param #2 Value>, ...")
cmd("<Target Name>", "<Command Name>", "Param #1 Name" => <Param #1 Value>, "Param #2 Name" => <Param #2 Value>, ...)
```

PARAMETER	DESCRIPTION
Target Name	Name of the target this command is associated with.
Command Name	Name of this command. Also referred to as its mnemonic.
Param #x Name	Name of a command parameter. If there are no parameters then the 'with' keyword should not be given.

Param #x Value	Value of the command parameter. Values are automatically converted to the appropriate type.

Example:

```
cmd("INST COLLECT with DURATION 10, TYPE NORMAL")
cmd("INST", "COLLECT", "DURATION" => 10, "TYPE" => "NORMAL")
```

cmd_no_range_check

The cmd_no_range_check method sends a specified command without performing range checking on its parameters. This should only be used when it is necessary to intentionally send a bad command parameter to test a target.

Syntax:

```
cmd_no_range_check("<Target Name> <Command Name> with <Param #1 Name> <Param #1 Value>, <Param #2 Name> <Param #
cmd_no_range_check("<Target Name>", "<Command Name>", "Param #1 Name" => <Param #1 Value>, "Param #2 Name" => <Par
```

PARAMETER	DESCRIPTION
Target Name	Name of the target this command is associated with.
Command Name	Name of this command. Also referred to as its mnemonic.
Param #x Name	Name of a command parameter. If there are no parameters then the 'with' keyword should not be given.
Param #x Value	Value of the command parameter. Values are automatically converted to the appropriate type.

Example:

```
cmd_no_range_check("INST COLLECT with DURATION 11, TYPE NORMAL")
cmd_no_range_check("INST", "COLLECT", "DURATION" => 11, "TYPE" => "NORMAL")
```

cmd_no_hazardous_check

The cmd_no_hazardous_check method sends a specified command without performing the notification if it is a hazardous command. This should only be used when it is necessary to fully automate testing involving hazardous commands.

Syntax:

```
cmd_no_hazardous_check("<Target Name> <Command Name> with <Param #1 Name> <Param #1 Value>, <Param #2 Name> <Par
cmd_no_hazardous_check("<Target Name>", "<Command Name>", "Param #1 Name" => <Param #1 Value>, "Param #2 Name" =>
```

PARAMETER	DESCRIPTION
Target Name	Name of the target this command is associated with.
Command Name	Name of this command. Also referred to as its mnemonic.
Param #x Name	Name of a command parameter. If there are no parameters then the 'with' keyword should not be given.
Param #x Value	Value of the command parameter. Values are automatically converted to the appropriate type.

Example:

```
cmd_no_hazardous_check("INST CLEAR")
cmd_no_hazardous_check("INST", "CLEAR")
```

cmd_no_checks

The cmd_no_checks method sends a specified command without performing the parameter range checks or notification if it is a hazardous command. This should only be used when it is necessary to fully automate testing involving hazardous commands that intentially have invalid parameters.

Syntax:

```
cmd_no_checks("<Target Name> <Command Name> with <Param #1 Name> <Param #1 Value>, <Param #2 Name> <Param #2 Valu
cmd_no_checks("<Target Name>", "<Command Name>", "Param #1 Name" => <Param #1 Value>, "Param #2 Name" => <Param #2
```

PARAMETER	DESCRIPTION
Target Name	Name of the target this command is associated with.
Command Name	Name of this command. Also referred to as its mnemonic.
Param #x Name	Name of a command parameter. If there are no parameters then the 'with' keyword should not be given.
Param #x Value	Value of the command parameter. Values are automatically converted to the appropriate type.

Example:

```
cmd_no_checks("INST COLLECT with DURATION 11, TYPE SPECIAL")
cmd_no_checks("INST", "COLLECT", "DURATION" => 11, "TYPE" => "SPECIAL")
```

cmd_raw

The cmd_raw method sends a specified command without running conversions.

Syntax:

```
cmd_raw("<Target Name> <Command Name> with <Param #1 Name> <Param #1 Value>, <Param #2 Name> <Param #2 Value>, ..."
cmd_raw("<Target Name>", "<Command Name>", "<Param #1 Name>" => <Param #1 Value>, "<Param #2 Name>" => <Param #2
```

PARAMETER	DESCRIPTION
Target Name	Name of the target this command is associated with.
Command Name	Name of this command. Also referred to as its mnemonic.
Param #x Name	Name of a command parameter. If there are no parameters then the 'with' keyword should not be given.
Param #x Value	Value of the command parameter. Values are automatically converted to the appropriate type.

Example:

```
cmd_raw("INST COLLECT with DURATION 10, TYPE 0")
cmd_raw("INST", "COLLECT", "DURATION" => 10, TYPE => 0)
```

cmd_raw_no_range_check

The cmd_raw_no_range_check method sends a specified command without running conversions or performing range checking on its parameters. This should only be used when it is necessary to intentionally send a bad command parameter to test a target.

Syntax:

```
cmd_raw_no_range_check("<Target Name> <Command Name> with <Param #1 Name> <Param #1 Value>, <Param #2 Name> <Par
cmd_raw_no_range_check("<Target Name>", "<Command Name>", "<Param #1 Name>" => <Param #1 Value>, "<Param #2 Name>
```

PARAMETER	DESCRIPTION
Target Name	Name of the target this command is associated with.
Command Name	Name of this command. Also referred to as its mnemonic.
Param #x Name	Name of a command parameter. If there are no parameters then the 'with' keyword should not be given.
Param #x Value	Value of the command parameter. Values are automatically converted to the appropriate type.

Example:

```
cmd_raw_no_range_check("INST COLLECT with DURATION 11, TYPE 0")
cmd_raw_no_range_check("INST", "COLLECT", "DURATION" => 11, "TYPE" => 0)
```

cmd_raw_no_hazardous_check

The cmd_raw_no_hazardous_check method sends a specified command without running conversions or performing the notification if it is a hazardous command. This should only be used when it is necessary to fully automate testing involving hazardous commands.

Syntax:

```
cmd_raw_no_hazardous_check("<Target Name> <Command Name> with <Param #1 Name> <Param #1 Value>, <Param #2 Name>
cmd_raw_no_hazardous_check("<Target Name>", "<Command Name>", "<Param #1 Name>" => <Param #1 Value>, "<Param #2 Na
```

PARAMETER	DESCRIPTION
Target Name	Name of the target this command is associated with.
Command Name	Name of this command. Also referred to as its mnemonic.
Param #x Name	Name of a command parameter. If there are no parameters then the 'with' keyword should not be given.
Param #x Value	Value of the command parameter. Values are automatically converted to the appropriate type.

Example:

```
cmd_raw_no_hazardous_check("INST CLEAR")
cmd_raw_no_hazardous_check("INST", "CLEAR")
```

cmd_raw_no_checks

The cmd_raw_no_checks method sends a specified command without running conversions or performing the parameter range checks or notification if it is a hazardous command. This should only be used when it is necessary to fully automate testing involving hazardous commands that intentially have invalid parameters.

Syntax:

```
cmd_raw_no_checks("<Target Name> <Command Name> with <Param #1 Name> <Param #1 Value>, <Param #2 Name> <Param #2
cmd_raw_no_checks("<Target Name>", "<Command Name>", "<Param #1 Name>" => <Param #1 Value>, "<Param #2 Name>" =>
```

PARAMETER	DESCRIPTION
Target Name	Name of the target this command is associated with.
Command Name	Name of this command. Also referred to as its mnemonic.
Param #x Name	Name of a command parameter. If there are no parameters then the 'with' keyword should not be given.
Param #x Value	Value of the command parameter. Values are automatically converted to the appropriate type.

Example:

```
cmd_raw_no_checks("INST COLLECT with DURATION 11, TYPE 1")
cmd_raw_no_checks("INST", "COLLECT", "DURATION" => 11, "TYPE" => 1)
```

send_raw

The send_raw method sends raw data on an interface.

Syntax: `send_raw(<Interface Name>, <data>)`

PARAMETER	DESCRIPTION
Interface Name	Name of the interface to send the raw data on.
Data	Raw ruby string of data to send.

Example:

```
send_raw("INST1INT", data)
```

send_raw_file

The send_raw_file method sends raw data on an interface from a file.

Syntax: `send_raw_file(<Interface Name>, <filename>)`

PARAMETER	DESCRIPTION
Interface Name	Name of the interface to send the raw data on.
filename	Full path to the file with the data to send.

Example:

```
send_raw_file("INST1INT", "/home/user/data_to_send.bin")
```

get_cmd_list

The get_cmd_list method returns an array of the commands that are available for a particular target. The returned array is an array of array swhere each subarray contains the command name and description.

Syntax: `get_cmd_list("<Target Name>")`

PARAMETER	DESCRIPTION
Target Name	Name of the target.

Example:

```
cmd_list = get_cmd_list("INST")
puts cmd_list.inspect # [['TARGET_NAME', 'DESCRIPTION'], ...]
```

get_cmd_param_list

The get_cmd_param_list method returns an array of the command parameters that are available for a particular command. The returned array is an array of arrays where each subarray contains [parameter_name, default_value, states_hash, description, units_full, units, required_flag]

Syntax: `get_cmd_param_list("<Target Name>", "<Command Name>")`

PARAMETER	DESCRIPTION
Target Name	Name of the target.
Command Name	Name of the command.

Example:

```
cmd_param_list = get_cmd_param_list("INST", "COLLECT")
puts cmd_param_list.inspect # [["CCSDSVER", 0, nil, "CCSDS primary header version number", nil, nil, false], ...]
```

get_cmd_hazardous

The get_cmd_hazardous method returns true/false indicating whether a particular command is flagged as hazardous.

Syntax:

```
get_cmd_hazardous("<Target Name>", "<Command Name>", <Command Params - optional>)
```

PARAMETER	DESCRIPTION
Target Name	Name of the target.
Command Name	Name of the command.
Command Params	Hash of the parameters given to the command (optional). Note that some commands are only hazardous based on parameter states.

Example:

```
hazardous = get_cmd_hazardous("INST", "COLLECT", {'TYPE' => 'SPECIAL'})
```

get_cmd_value (COSMOS 3.5.0+)

The get_cmd_value method returns reads a value from the most recently sent command packet. The pseudo-parameters 'RECEIVED_COUNT', 'RECEIVED_TIMEFORMATTED', and 'RECEIVED_TIMESECONDS' are also supported.

Syntax:

```
get_cmd_value("<Target Name>", "<Command Name>", "<Parameter Name>", <Value Type - optional>)
```

PARAMETER	DESCRIPTION
Target Name	Name of the target.
Command Name	Name of the command.
Parameter Name	Name of the command parameter.
Value Type	Value Type to read. :RAW, :CONVERTED, :FORMATTED, or :WITH_UNITS

Example:

```
value = get_cmd_value("INST", "COLLECT", "TEMP")
```

get_cmd_time (COSMOS 3.5.0+)

The get_cmd_time method returns the time of the most recent command sent.

Syntax:

```
get_cmd_time("<Target Name - optional>", "<Command Name - optional>")
```

PARAMETER	DESCRIPTION
Target Name	Name of the target. If not given, then the most recent command time to any target will be returned
Command Name	Name of the command. If not given, then the most recent command time to the given target will be returned

Example:

```
target_name, command_name, time = get_cmd_time() # Name of the most recent command sent to any target and time
target_name, command_name, time = get_cmd_time("INST") # Name of the most recent command sent to the INST target and time
target_name, command_name, time = get_cmd_time("INST", "COLLECT") # Name of the most recent INST COLLECT command and tim
```

Handling Telemetry

These methods allow the user to interact with telemetry items.

check

The check method performs a verification of a telemetry item using its converted telemetry type. If the verification fails then the script will be paused with an error. If no comparision is given to check then the telemetry item is simply printed to the script output. Note: In most cases using wait_check is a better choice than using check.

Syntax:

```
check("<Target Name> <Packet Name> <Item Name> <Comparison - optional>")
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.

Comparison	A comparison to perform against the telemetry item. If a comparison is not given then the telemetry item will just be printed into the script log.

Example:

```
check("INST HEALTH_STATUS COLLECTS > 1")
```

check_raw

The check_raw method performs a verification of a telemetry item using its raw telemetry type. If the verification fails then the script will be paused with an error. If no comparision is given to check then the telemetry item is simply printed to the script output. Note: In most cases using wait_check_raw is a better choice than using check_raw.

Syntax:

```
check_raw("<Target Name> <Packet Name> <Item Name> <Comparison>")
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Comparison	A comparison to perform against the telemetry item. If a comparison is not given then the telemetry item will just be printed into the script log. If a comparison is not given then the telemetry item will just be printed into the script log.

Example:

```
check_raw("INST HEALTH_STATUS COLLECTS > 1")
```

check_formatted

The check_formatted method performs a verification of a telemetry item using its formatted telemetry type. If the verification fails then the script will be paused with an error. If no comparision is given to check then the telemetry item is simply printed to the script output.

Syntax:

```
check_formatted("<Target Name> <Packet Name> <Item Name> <Comparison>")
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Comparison	A comparison to perform against the telemetry item. If a comparison is not given then the telemetry item will just be printed into the script log. If a comparison is not given then the telemetry item will just be printed into the script log.

Example:

```
check_formatted("INST HEALTH_STATUS COLLECTS == '1'")
```

check_with_units

The check_with_units method performs a verification of a telemetry item using its formatted with units telemetry type. If the verification fails then the script will be paused with an error. If no comparision is given to check then the telemetry item is simply printed to the script output.

Syntax:

```
check_with_units("<Target Name> <Packet Name> <Item Name> <Comparison>")
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Comparison	A comparison to perform against the telemetry item. If a comparison is not given then the telemetry item will just be printed into the script log. If a comparison is not given then the telemetry item will just be printed into the script log.

Example:

```
check_with_units("INST HEALTH_STATUS COLLECTS == '1'")
```

check_tolerance

The check_tolerance method checks a converted telemetry item against an expected value with a tolerance. If the verification fails then the script will be paused with an error. Note: In most cases using wait_check_tolerance is a better choice than using check_tolerance.

Syntax:

```
check_tolerance("<Target Name> <Packet Name> <Item Name>", <Expected Value>, <Tolerance>)
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Expected Value	Expected value of the telemetry item.
Tolerance	± Tolerance on the expected value.

Example:

```
check_tolerance("INST HEALTH_STATUS COLLECTS", 10.0, 5.0)
```

check_tolerance_raw

The check_tolerance_raw method checks a raw telemetry item against an expected value with a tolerance. If the verification fails then the script will be paused with an error. Note: In most cases using wait_check_tolerance_raw is a better choice than using check_tolerance_raw.

Syntax:

```
check_tolerance_raw("<Target Name> <Packet Name> <Item Name>", <Expected Value>, <Tolerance>)
```

PARAMETER	DESCRIPTION

Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Expected Value	Expected value of the telemetry item.
Tolerance	± Tolerance on the expected value.

Example:

```
check_tolerance_raw("INST HEALTH_STATUS COLLECTS", 10.0, 5.0)
```

check_expression

The check_expression method evaluates an expression. If the expression evaluates to false the script will be paused with an error. This method can be used to perform more complicated comparisons than using check as shown in the example. Note: In most cases using wait_check_expression is a better choice than using check_expression.

Remember that everything inside the check_expression string will be evaluated directly by the Ruby interpreter and thus must be valid syntax. A common mistake is to check a variable like so:

```
check_expression("#{answer} == 'yes'") # where answer contains 'yes'
```

This evaluates to `yes == 'yes'` which is not valid syntax because the variable yes is not defined (usually). The correct way to write this expression is as follows:

```
check_expression("'#{answer}' == 'yes'") # where answer contains 'yes'
```

Now this evaluates to `'yes' == 'yes'` which is true so the check passes.

Syntax: `check_expression("<Expression>")`

PARAMETER	DESCRIPTION
Expression	A ruby expression to evaluate.

Example:

```
check_expression("tlm('INST HEALTH_STATUS COLLECTS') > 5 and tlm('INST HEALTH_STATUS TEMP1') > 25.0")
```

tlm

The tlm method reads the converted form of a specified telemetry item.

Syntax: `tlm("<Target Name> <Packet Name> <Item Name>")`

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.

Example:

```
value = tlm("INST HEALTH_STATUS COLLECTS")
```

tlm_raw

The tlm_raw method reads the raw form of a specified telemetry item.

Syntax: `tlm_raw("<Target Name> <Packet Name> <Item Name>")`

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.

Example:

```
value = tlm_raw("INST HEALTH_STATUS COLLECTS")
```

tlm_formatted

The tlm_formatted method reads the formatted form of a specified telemetry item.

Syntax: `tlm_formatted("<Target Name> <Packet Name> <Item Name>")`

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.

Example:

```
value = tlm_formatted("INST HEALTH_STATUS COLLECTS")
```

tlm_with_units

The tlm_with_units method reads the formatted with units form of a specified telemetry item.

Syntax: `tlm_with_units("<Target Name> <Packet Name> <Item Name>")`

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.

Example:

```
value = tlm_with_units("INST HEALTH_STATUS COLLECTS")
```

tlm_variable

The tlm_variable method reads a specified telemetry item with a variable value type.

Syntax: `tlm_variable("<Target Name> <Packet Name> <Item Name>", <Value Type>)`

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.

Value Type	Value Type to read. :RAW, :CONVERTED, :FORMATTED, or :WITH_UNITS

Example:

```
value = tlm_variable("INST HEALTH_STATUS COLLECTS", :RAW)
```

get_tlm_packet

The get_tlm_packet method returns the names, values, and limits states of all telemetry items in a specified packet. The value is returned as an array of arrays with each entry containing [item_name, item_value, limits_state].

Syntax: `get_tlm_packet("<Target Name>", "<Packet Name>", value_type)`

PARAMETER	DESCRIPTION
Target Name	Name of the target.
Packet Name	Name of the packet.
value_type	Telemetry Type to read the values in. :RAW, :CONVERTED, :FORMATTED, or :WITH_UNITS. Defaults to :CONVERTED

Example:

```
names_values_and_limits_states = get_tlm_packet("INST", "HEALTH_STATUS", :FORMATTED)
```

get_tlm_values

The get_tlm_values method returns the values, limits_states, limits_settings, and current limits_set for a specified set of telemetry items. Items can be in any telemetry packet in the system. They can all be retrieved using the same value type or a specific value type can be specified for each item.

Syntax:

```
values, limits_states, limits_settings, limits_set = get_tlm_values(<items>, <value_types>)
```

PARAMETER	DESCRIPTION
items	Array of item arrays of the form [[Target Name #1, Packet Name #1, Item Name #1], …]
value_types	Telemetry Type to read the values in. :RAW, :CONVERTED, :FORMATTED, or :WITH_UNITS. Defaults to :CONVERTED . May be specified as a single symbol that applies to all items or an array of symbols, one for each item.

Example:

```
values, limits_states, limits_settings, limits_set = get_tlm_values([["INST", "ADCS", "Q1"], ["INST", "ADCS", "Q2"]], [:FORMATTED, :WITH_
```

get_tlm_list

The get_tlm_list method returns an array of the telemetry packets and their descriptions that are available for a particular target.

Syntax:

```
packet_names_and_descriptions = get_tlm_pkt_list("<Target Name>")
```

PARAMETER	DESCRIPTION
Target Name	Name of the target.

Example:

```
packet_names_and_descriptions = = get_tlm_list("INST")
```

get_tlm_item_list

The get_tlm_item_list method returns an array of the telemetry items that are available for a particular telemetry packet. The returned value is an array of arrays where each subarray contains [item_name, item_states_hash, description]

Syntax: `get_tlm_item_list("<Target Name>", "<Packet Name>")`

PARAMETER	DESCRIPTION
Target Name	Name of the target.
Packet Name	Name of the telemetry packet.

Example:

```
item_names_states_and_descriptions = get_tlm_item_list("INST", "HEALTH_STATUS")
```

get_tlm_details

The get_tlm_details method returns an array with details about the specified telemetry items such as their limits and states.

Syntax: `get_tlm_item_details(<items>)`

PARAMETER	DESCRIPTION
items	Array of item arrays of the form [[Target Name #1, Packet Name #1, Item Name #1], ...]

Example:

```
details = get_tlm_item_details("INST", "HEALTH_STATUS", "COLLECTS")
```

set_tlm

The set_tlm method sets a telemetry item value in the Command and Telemetry Server. This value will be overwritten if a new packet is received from an interface. For that reason this method is most useful if interfaces are disconnected or for testing via the Script Runner disconnect mode. (Note that in disconnect mode it will only set telemetry within ScriptRunner. Other tools like TlmViewer will not reflect any changes) Manually setting telemetry values allows for the execution of many logical paths in scripts.

Syntax: `set_tlm("<Target> <Packet> <Item> = <Value>")`

PARAMETER	DESCRIPTION
Target	Target name
Packet	Packet name
Item	Item name
Value	Value to set

Example:

```
set_tlm("INST HEALTH_STATUS COLLECTS = 5")
check("INST HEALTH_STATUS COLLECTS == 5") # This will pass since we just set it to 5
```

set_tlm_raw

The set_tlm_raw method sets a raw telemetry item value in the Command and Telemetry Server. This value will be overwritten if a new packet is received from an interface. For that reason this method is most useful if interfaces are disconnected or for testing via the Script Runner disconnect mode. (Note that in disconnect mode it will only set telemetry within ScriptRunner. Other tools like TlmViewer will not reflect any changes) Manually setting telemetry values allows for the execution of many logical paths in scripts.

Syntax: `set_tlm_raw("<Target> <Packet> <Item> = <Value>")`

PARAMETER	DESCRIPTION
Target	Target name
Packet	Packet name
Item	Item name
Value	Value to set

Example:

```
# Assuming TEMP1 is defined with a conversion (as it is in the COSMOS demo)
set_tlm("INST HEALTH_STATUS TEMP1 = 5")
check_tolerance("INST HEALTH_STATUS TEMP1", 5, 0.5) # Pass
set_tlm_raw("INST HEALTH_STATUS TEMP1 = 5")
check_tolerance("INST HEALTH_STATUS TEMP1", 5, 0.5) # Fail because we set the raw value not the converted value
```

Packet Data Subscriptions

Methods for subscribing to specific packets of data. This provides an interface to ensure that each telemetry packet is received and handled rather than relying on polling where some data may be missed.

subscribe_packet_data

The subscribe_packet_data method allows the user to listen for one or more telemetry packets of data to arrive. A unique id is returned to the tool which is used to retrieve the data. The subscribed packets are placed into a queue where they can then be processed one at a time.

Syntax: `subscribe_packet_data(packets, queue_size)`

PARAMETER	DESCRIPTION
packets	Nested array of target name/packet name pairs that the user wishes to subscribe to.
queue_size	Number of packets to let queue up before dropping the connection. Defaults to 1000.

Example:

```
id = subscribe_packet_data([['INST', 'HEALTH_STATUS'], ['INST', 'ADCS']], 2000)
```

unsubscribe_packet_data

The unsubscribe_packet_data method allows the user to stop listening for packet_data. This should be called to reduce the server's load if the subscription is no longer needed.

Syntax: `unsubscribe_packet_data(id)`

PARAMETER	DESCRIPTION
id	Unique id given to the tool by subscribe_packet_data.

Example:

```
unsubscribe_packet_data(id)
```

get_packet

Receives a subscribed telemetry packet. If get_packet is called non-blocking = true, get_packet will raise an error if the queue is empty.

Syntax: `get_packet(id, non_block (optional))`

PARAMETER	DESCRIPTION
id	Unique id given to the tool by subscribe_packet_data.
non_block	Boolean to indicate if the method should block until an packet of data is received or not. Defaults to false, blocks reading data from queue.

Example:

```
packet = get_packet(id)
value = packet.read('ITEM_NAME')
```

get_packet_data

NOTE: Most users will want to use get_packet() instead of this lower level method. The get_packet_data method returns a ruby string containing the packet data from a specified telemetry packet. It also returns which telemetry packet the data is from. Can be run in a non-blocking or blocking manner. Packets are queued after calling subscribe_packet_data and none will be lost. If 1000 (or whatever queue_size was specified in subscribe_packet_data) packets are queued and get_packet_data has not been called or has not been keeping up, then the subscription will be dropped.

The returned packet data can be used to populate a packet object. A packet object can be obtained from the System object.

If get_packet_data is called non-blocking = true, get_packet_data will raise an error if the queue is empty.

Syntax: `get_packet_data(id, non_block)`

PARAMETER	DESCRIPTION
id	Unique id given to the tool by subscribe_packet_data.
non_block	Boolean to indicate if the method should block until an packet of data is received or not. Defaults to false, blocks reading data from queue.

Example:

```
id = subscribe_packet_data([["TGT", "PKT1"], ["TGT", "PKT2"]]) # note double nested array

buffer, target_name, packet_name, received_time, received_count = get_packet_data(id)
packet = System.telemetry.packet(target_name, packet_name).clone
packet.buffer = buffer
packet.received_time = received_time
packet.received_count = received_count
```

Delays

These methods allow the user to pause the script to wait for telemetry to change or for an amount of time to pass.

wait

The wait method pauses the script for a configurable amount of time or until a converted telemetry item meets given criteria. It supports three different syntaxes as shown. If no parameters are given then an infinite wait occurs until the user presses Go. Note that on a timeout, wait does not stop the script, usually wait_check is a better choice.

Syntax:

```
wait()
wait(<Time>)
```

PARAMETER	DESCRIPTION
Time	Time in Seconds to delay for.

```
wait("<Target Name> <Packet Name> <Item Name> <Comparison>", <Timeout>, <Polling Rate (optional)>)
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Comparison	A comparison to perform against the telemetry item.
Timeout	Timeout in seconds. Script will proceed if the wait statement times out waiting for the comparison to be true.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Examples:

```
wait()
wait(5)
wait("INST HEALTH_STATUS COLLECTS == 3", 10)
```

wait_raw

The wait_raw method pauses the script for a configurable amount of time or until a raw telemetry item meets given criteria. It supports two different syntaxes as shown. If no parameters are given then an infinite wait occurs until the user presses Go. Note that on a timeout, wait_raw does not stop the script, usually wait_check_raw is a better choice.

Syntax: `wait_raw(<Time>)`

PARAMETER	DESCRIPTION
Time	Time in Seconds to delay for.

```
wait_raw("<Target Name> <Packet Name> <Item Name> <Comparison>", <Timeout>, <Polling Rate (optional)>)
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Comparison	A comparison to perform against the telemetry item.

Timeout	Timeout in seconds. Script will proceed if the wait statement times out waiting for the comparison to be true.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Examples:

```
wait_raw(5)
wait_raw("INST HEALTH_STATUS COLLECTS == 3", 10)
```

wait_tolerance

The wait_tolerance method pauses the script for a configurable amount of time or until a converted telemetry item meets equals an expected value within a tolerance. Note that on a timeout, wait_tolerance does not stop the script, usually wait_check_tolerance is a better choice.

Syntax:

```
wait_tolerance("<Target Name> <Packet Name> <Item Name>", <Expected Value>, <Tolerance>, <Timeout>, <Polling Rate (optional):
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Expected Value	Expected value of the telemetry item.
Tolerance	± Tolerance on the expected value.
Timeout	Timeout in seconds. Script will proceed if the wait statement times out waiting for the comparison to be true.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Examples:

```
wait_tolerance("INST HEALTH_STATUS COLLECTS", 10.0, 5.0, 10)
```

wait_tolerance_raw

The wait_tolerance_raw method pauses the script for a configurable amount of time or until a raw telemetry item meets equals an expected value within a tolerance. Note that on a timeout, wait_tolerance_raw does not stop the script, usually wait_check_tolerance_raw is a better choice.

Syntax:

```
wait_tolerance_raw("<Target Name> <Packet Name> <Item Name>", <Expected Value>, <Tolerance>, <Timeout>, <Polling Rate (optic
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Expected Value	Expected value of the telemetry item.

Tolerance	± Tolerance on the expected value.
Timeout	Timeout in seconds. Script will proceed if the wait statement times out waiting for the comparison to be true.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Examples:

```
wait_tolerance_raw("INST HEALTH_STATUS COLLECTS", 10.0, 5.0, 10)
```

wait_expression

The wait_expression method pauses the script until an expression is evaluated to be true or a timeout occurs. If a timeout occurs the script will continue. This method can be used to perform more complicated comparisons than using wait as shown in the example. Note that on a timeout, wait_expression does not stop the script, usually wait_check_expression is a better choice.

Syntax:

```
wait_expression("<Expression>", <Timeout>, <Polling Rate (optional)>)
```

PARAMETER	DESCRIPTION
Expression	A ruby expression to evaluate.
Timeout	Timeout in seconds. Script will proceed if the wait statement times out waiting for the comparison to be true.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Example:

```
wait_expression("tlm('INST HEALTH_STATUS COLLECTS') > 5 and tlm('INST HEALTH_STATUS TEMP1') > 25.0", 10)
```

wait_packet

The wait_packet method pauses the script until a certain number of packets have been received. If a timeout occurs the script will continue. Note that on a timeout, wait_packet does not stop the script, usually wait_check_packet is a better choice.

Syntax:

```
wait_packet("<Target>", "<Packet>", <Num Packets>, <Timeout>, <Polling Rate (optional)>)
```

PARAMETER	DESCRIPTION
Target	The target name
Packet	The packet name
Num Packets	The number of packets to receive
Timeout	Timeout in seconds.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Example:

```
wait_packet('INST', 'HEALTH_STATUS', 5, 10) # Wait for 5 INST HEALTH_STATUS packets over 10s
```

wait_check

The wait_check method combines the wait and check keywords into one. This pauses the script until the converted value of a telemetry item meets given criteria or times out. On a timeout the script stops.

Syntax:

```
wait_check("<Target Name> <Packet Name> <Item Name> <Comparison>", <Timeout>, <Polling Rate (optional)>)
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Comparison	A comparison to perform against the telemetry item.
Timeout	Timeout in seconds. Script will stop if the wait statement times out waiting for the comparison to be true.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Example:

```
wait_check("INST HEALTH_STATUS COLLECTS > 5", 10)
```

wait_check_raw

The wait_check_raw method combines the wait_raw and check_raw keywords into one. This pauses the script until the raw value of a telemetry item meets given criteria or times out. On a timeout the script stops.

Syntax:

```
wait_check_raw("<Target Name> <Packet Name> <Item Name> <Comparison>", <Timeout>, <Polling Rate (optional)>)
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Comparison	A comparison to perform against the telemetry item.
Timeout	Timeout in seconds. Script will stop if the wait statement times out waiting for the comparison to be true.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Example:

```
wait_check_raw("INST HEALTH_STATUS COLLECTS > 5", 10)
```

wait_check_tolerance

The wait_check_tolerance method pauses the script for a configurable amount of time or until a converted telemetry item equals an expected value within a tolerance. On a timeout the script stops.

Syntax:

```
wait_check_tolerance("<Target Name> <Packet Name> <Item Name>", <Expected Value>, <Tolerance>, <Timeout>, <Polling Rate (op
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Expected Value	Expected value of the telemetry item.
Tolerance	± Tolerance on the expected value.
Timeout	Timeout in seconds. Script will proceed if the wait statement times out waiting for the comparison to be true.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Examples:

```
wait_check_tolerance("INST HEALTH_STATUS COLLECTS", 10.0, 5.0, 10)
```

wait_check_tolerance_raw

The wait_check_tolerance_raw method pauses the script for a configurable amount of time or until a raw telemetry item meets equals an expected value within a tolerance. On a timeout the script stops.

Syntax:

```
wait_check_tolerance_raw("<Target Name> <Packet Name> <Item Name>", <Expected Value>, <Tolerance>, <Timeout>, <Polling Rat
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Expected Value	Expected value of the telemetry item.
Tolerance	± Tolerance on the expected value.
Timeout	Timeout in seconds. Script will proceed if the wait statement times out waiting for the comparison to be true.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Examples:

```
wait_check_tolerance_raw("INST HEALTH_STATUS COLLECTS", 10.0, 5.0, 10)
```

wait_check_expression

The wait_check_expression method pauses the script until an expression is evaluated to be true or a timeout occurs. If a timeout occurs the script will stop. This method can be used to perform more complicated comparisons than using wait as shown in the example. Also see the syntax notes for check_expression.

Syntax:

```
wait_check_expression("<Expression>", <Timeout>, <Polling Rate (optional)>)
```

PARAMETER	DESCRIPTION
Expression	A ruby expression to evaluate.
Timeout	Timeout in seconds. Script will stop if the wait statement times out waiting for the comparison to be true.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Example:

```
wait_check_expression("tlm('INST HEALTH_STATUS COLLECTS') > 5 and tlm('INST HEALTH_STATUS TEMP1') > 25.0", 10)
```

wait_check_packet

The wait_check_packet method pauses the script until a certain number of packets have been received. If a timeout occurs the script will stop.

Syntax:

```
wait_check_packet("<Target>", "<Packet>", <Num Packets>, <Timeout>, <Polling Rate (optional)>)
```

PARAMETER	DESCRIPTION
Target	The target name
Packet	The packet name
Num Packets	The number of packets to receive
Timeout	Timeout in seconds. Script will stop if the wait statement times out waiting specified number of packets.
Polling Rate	How often the comparison is evaluated in seconds. Defaults to 0.25 if not specified.

Example:

```
wait_check_packet('INST', 'HEALTH_STATUS', 5, 10) # Wait for 5 INST HEALTH_STATUS packets over 10s
```

Limits

These methods deal with handling telemetry limits.

limits_enabled?

The limits_enabled? method returns true/false depending on whether limits are enabled for a telemetry item.

Syntax: `limits_enabled?("<Target Name> <Packet Name> <Item Name>")`

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.

Example:

```
enabled = limits_enabled?("INST HEALTH_STATUS TEMP1")
```

enable_limits

The enable_limits method enables limits monitoring for the specified telemetry item.

Syntax: `enable_limits("<Target Name> <Packet Name> <Item Name>")`

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.

Example:

```
enable_limits("INST HEALTH_STATUS TEMP1")
```

disable_limits

The disable_limits method disables limits monitoring for the specified telemetry item.

Syntax: `disable_limits("<Target Name> <Packet Name> <Item Name>")`

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.

Example:

```
disable_limits("INST HEALTH_STATUS TEMP1")
```

enable_limits_group

The enable_limits_group method enables limits monitoring on a set of telemetry items specified in a limits group.

Syntax: `enable_limits_group("<Limits Group Name>")`

PARAMETER	DESCRIPTION
Limits Group Name	Name of the limits group.

Example:

```
enable_limits_group("SAFE_MODE")
```

disable_limits_group

The disable_limits_group method disables limits monitoring on a set of telemetry items specified in a limits group.

Syntax: `disable_limits_group("<Limits Group Name>")`

PARAMETER	DESCRIPTION
Limits Group Name	Name of the limits group.

Example:

```
disable_limits_group("SAFE_MODE")
```

get_limits_groups

The get_limits_groups method returns the list of limits groups in the system.

Syntax: `get_limits_groups()`

Example:

```
limits_groups = get_limits_groups()
```

set_limits_set

The set_limits_set method sets the current limits set. The default limits set is :DEFAULT.

Syntax: `set_limits_set("<Limits Set Name>")`

PARAMETER	DESCRIPTION
Limits Set Name	Name of the limits set.

Example:

```
set_limits_set("DEFAULT")
```

get_limits_set

The get_limits_set method returns the name of the current limits set. The default limits set is :DEFAULT.

Syntax: `get_limits_set()`

Example:

```
limits_set = get_limits_set()
```

get_limits_sets

The get_limits_sets method returns the list of limits sets in the system.

Syntax: `get_limits_sets()`

Example:

```
limits_sets = get_limits_sets()
```

get_limits

The get_limits method returns limits settings for a telemetry point.

Syntax:

```
get_limits(<Target Name>, <Packet Name>, <Item Name>, <Limits Set (optional)>)
```

e

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.

Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Limits Set	Get the limits for a specific limits set. If not given then it defaults to returning the settings for the current limits set.

Example:

```
limits_set, persistence_setting, enabled, red_low, yellow_low, yellow_high, red_high, green_low, green_high = get_limits('INST', 'HEALTH_S
```

set_limits

The set_limits_method sets limits settings for a telemetry point. Note: In most cases it would be better to update your config files or use different limits sets rather than changing limits settings in realtime.

Syntax:

```
set_limits(<Target Name>, <Packet Name>, <Item Name>, <Red Low>, <Yellow Low>, <Yellow High>, <Red High>, <Green Low (optio
```

PARAMETER	DESCRIPTION
Target Name	Name of the target of the telemetry item.
Packet Name	Name of the telemetry packet of the telemetry item.
Item Name	Name of the telemetry item.
Red Low	Red Low setting for this limits set. Any value below this value will be make the item red.
Yellow Low	Yellow Low setting for this limits set. Any value below this value but greater than Red Low will be make the item yellow.
Yellow High	Yellow High setting for this limits set. Any value above this value but less than Red High will be make the item yellow.
Red High	Red High setting for this limits set. Any value above this value will be make the item red.
Green Low	Optional. If given, any value greater than Green Low and less than Green_High will make the item blue indicating a good operational value.
Green High	Optional. If given, any value greater than Green Low and less than Green_High will make the item blue indicating a good operational value.
Limits Set	Optional. Set the limits for a specific limits set. If not given then it defaults to setting limts for the :CUSTOM limits set.
Persistence	Optional. Set the number of samples this item must be out of limits before changing limits state. Defaults to no change. Note: This affects all limits settings across limits sets.
Enabled	Optional. Whether or not limits are enabled for this item. Defaults to true. Note: This affects all limits settings across limits sets.

Example:

```
set_limits('INST', 'HEALTH_STATUS', 'TEMP1', -10.0, 0.0, 50.0, 60.0, 30.0, 40.0, :TVAC, 1, true)
```

get_out_of_limits

The get_out_of_limits method returns an array with the target_name, packet_name, item_name, and limits_state of all items that are out of their limits ranges.

Syntax: `get_out_of_limits()`

Example:

```
out_of_limits_items = get_out_of_limits()
```

get_overall_limits_state

The get_overall_limits_state method returns the overall limits state for the COSMOS system. Returns :GREEN, :YELLOW, :RED, or :STALE.

Syntax: `get_overall_limits_state(<ignored_items> (optional))`

PARAMETER	DESCRIPTION
Ignored Items	Array of arrays with items to ignore when determining the overall limits state. [[‘TARGET_NAME’, ‘PACKE_NAME’, ‘ITEM_NAME’], ...]

Example:

```
overall_limits_state = get_overall_limits_state()
overall_limits_state = get_overall_limits_state([['INST', 'HEALTH_STATUS', 'TEMP1']])
```

Limits Events

Methods for handling limits events.

subscribe_limits_events

The subscribe_limits_events method allows the user to listen for events regarding telemetry items going out of limits or changes in limits set. A unique id is returned to the tool which is used to retrieve the events.

Syntax: `subscribe_limits_events(<Queue Size (optional)>)`

PARAMETER	DESCRIPTION
Queue Size	How many limits events to queue up before dropping the client. Defaults to 1000 if not given.

Example:

```
id = subscribe_limits_events()
```

unsubscribe_limits_events

The unsubscribe_limits_events method allows the user to stop listening for events regarding telemetry items going out of limits or changes in limits set.

Syntax: `unsubscribe_limits_events(<id>)`

PARAMETER	DESCRIPTION
id	Unique id given to the user by subscribe_limits_events.

Example:

```
unsubscribe_limits_events(id)
```

get_limits_event

The get_limits_event method returns a limits event to the user who has already subscribed to limits event. Can be run in a non-blocking or blocking manner.

Syntax: `get_limits_event(<id>, <non_block (optional)>)`

PARAMETER	DESCRIPTION
id	Unique id given to the tool by subscribe_limits_events.
non_block	Boolean to indicate if the method should block until an event is received or not. Defaults to false.

Example:

```
event = get_limits_event(id, true)
puts event.inspect # [:LIMITS_CHANGE, "TARGET_NAME", "PACKET_NAME", "ITEM_NAME", :OLD_STATE, :NEW_STATE)
puts event.inspect # [:LIMITS_SET, :NEW_LIMITS_SET)
puts event.inspect # [:LIMITS_SETTINGS, "TARGET_NAME", "PACKET_NAME", "ITEM_NAME", :LIMITS_SET, persistence_setting, enabled_
```

Targets

Methods for getting knowledge about targets.

get_target_list

The get_target_list method returns a list of the targets in the system in an array.

Syntax: `get_target_list()`

Example:

```
targets = get_target_list()
```

Interfaces

These methods allow the user to manipulate COSMOS interfaces.

connect_interface

The connect_interface method connects to targets associated with a COSMOS interface.

Syntax: `connect_interface("<Interface Name>", <Interface Parameters (optional)>)`

PARAMETER	DESCRIPTION
Interface Name	Name of the interface.
Interface Parameters	Parameters used to initialize the interface. If none are given then the interface will use the parameters that were given in the server configuration file.

Example:

```
connect_interface("INT1")
```

disconnect_interface

The disconnect_interface method disconnects from targets associated with a COSMOS interface.

Syntax: `disconnect_interface("<Interface Name>")`

PARAMETER	DESCRIPTION
Interface Name	Name of the interface.

Example:

```
disconnect_interface("INT1")
```

interface_state

The interface_state method retrieves the current state of a COSMOS interface. Returns either 'CONNECTED', 'DISCONNECTED', or 'ATTEMPTING'.

Syntax: `interface_state("<Interface Name>")`

PARAMETER	DESCRIPTION
Interface Name	Name of the interface.

Example:

```
interface_state("INT1")
```

map_target_to_interface

The map_target_to_interface method allows a target to be mapped to an interface in realtime. If the target is already mapped to an interface it will be unmapped from the existing interface before being mapped to the new interface.

Syntax: `map_target_to_interface("<Target Name>", "<Interface Name>")`

PARAMETER	DESCRIPTION
Target Name	Name of the target.
Interface Name	Name of the interface.

Example:

```
map_target_to_interface("INST", "INT2")
```

get_interface_names

The get_interface_names method returns a list of the interfaces in the system in an array.

Syntax: `get_interface_names()`

Example:

```
interface_names = get_interface_names()
```

Routers

These methods allow the user to manipulate COSMOS routers.

connect_router

The connect_router method connects a COSMOS router.

Syntax: `connect_router("<Router Name>", <Router Parameters (optional)>)`

PARAMETER	DESCRIPTION
Router Name	Name of the router.

Router Parameters	Parameters used to initialize the router. If none are given then the routerwill use the parameters that were given in the server configuration file.

Example:

```
connect_ROUTER("INT1_ROUTER")
```

disconnect_router

The disconnect_router method disconnects a COSMOS router.

Syntax: `disconnect_router("<Router Name>")`

PARAMETER	DESCRIPTION
Router Name	Name of the router.

Example:

```
disconnect_router("INT1_ROUTER")
```

router_state

The router_state method retrieves the current state of a COSMOS router. Returns either 'CONNECTED', 'DISCONNECTED', or 'ATTEMPTING'.

Syntax: `router_state("<Router Name>")`

PARAMETER	DESCRIPTION
Router Name	Name of the router.

Example:

```
router_state("INT1_ROUTER")
```

get_router_names

The get_router_names method returns a list of the routers in the system in an array.

Syntax: `get_router_names()`

Example:

```
router_names = get_router_names()
```

Logging

These methods control command and telemetry logging.

get_cmd_log_filename

The get_cmd_log_filename method retrieves the current command log file for the specified log writer. Returns nil if not logging.

Syntax: `get_cmd_log_filename("<Packet Log Writer Name (optional)>")`

PARAMETER	DESCRIPTION
Packet Log Writer Name	Name of the packet log writer. Defaults to "DEFAULT"

Example:

```
get_cmd_log_filename("INT1")
```

get_tlm_log_filename

The get_tlm_log_filename method retrieves the current telemetry log file for the specified log writer. Returns nil if not logging.

Syntax: `get_tlm_log_filename("<Packet Log Writer Name (optional)>")`

PARAMETER	DESCRIPTION
Interface Name	Name of the interface.

Example:

```
get_tlm_log_filename("INT1")
```

start_logging

The start_logging method starts logging of commands sent and telemetry received for a packet log writer. If a log writer is already logging, this will start a new log file.

Syntax: `start_logging("<Packet Log Writer Name (optional)>", "<Label (optional)>")`

PARAMETER	DESCRIPTION
Packet Log Writer Name	Name of the packet log writer to command to start logging. Defaults to 'ALL' which causes all packet log writers to start logging commands and telemetry. If a log writer is already logging it will start a new file.
Label	Label to place on log files. Defaults to nil which means no label.

Example:

```
start_logging("int1")
```

start_cmd_log

The start_cmd_log method starts logging of commands sent. If a log writer is already logging, this will start a new log file.

Syntax: `start_cmd_log("<Packet Log Writer Name (optional)>")`

PARAMETER	DESCRIPTION
Packet Log Writer Name	Name of the packet log writer to command to start logging. Defaults to 'ALL' which causes all packet log writers to start logging commands. If a log writer is already logging it will start a new file.
Label	Label to place on log files. Defaults to nil which means no label.

Example:

```
start_cmd_log("int1")
```

start_tlm_log

The start_tlm_log method starts logging of telemetry received. If a log writer is already logging, this will start a new log

file.

Syntax: `start_tlm_log("<Packet Log Writer Name (optional)>")`

PARAMETER	DESCRIPTION
Packet Log Writer Name	Name of the packet log writer to command to start logging. Defaults to 'ALL' which causes all packet log writers to start logging telemetry. If a log writer is already logging it will start a new file.
Label	Label to place on log files. Defaults to nil which means no label.

Example:

```
start_tlm_log("int1")
```

stop_logging

The stop_logging method stops logging of commands sent and telemetry received for a packet log writer.

Syntax: `stop_logging("<Packet Log Writer Name (optional)>")`

PARAMETER	DESCRIPTION
Packet Log Writer Name	Name of the packet log writer to command to stop logging. Defaults to 'ALL' which causes all packet log writers to stop logging commands and telemetry.

Example:

```
stop_logging("int1")
```

stop_cmd_log

The stop_cmd_log method stops logging of commands sent.

Syntax: `stop_cmd_log("<Packet Log Writer Name (optional)>")`

PARAMETER	DESCRIPTION
Packet Log Writer Name	Name of the packet log writer to command to stop logging. Defaults to 'ALL' which causes all packet log writers to stop logging commands.

Example:

```
stop_cmd_log()
```

stop_tlm_log

The stop_tlm_log method stops logging of telemetry received.

Syntax: `stop_tlm_log("<Packet Log Writer Name (optional)>")`

PARAMETER	DESCRIPTION
Packet Log Writer Name	Name of the packet log writer to command to stop logging. Defaults to 'ALL' which causes all packet log writers to stop logging telemetry.

Example:

```
stop_tlm_log()
```

get_server_message_log_filename

Returns the filename of the COSMOS Command and Telemetry Server message log.

Syntax: `get_server_message_log_filename()`

Example:

```
filename = get_server_message_log_filename()
```

start_new_server_message_log

Starts a new COSMOS Command and Telemetry Server message log.

Syntax: `start_new_server_message_log()`

Example:

```
start_new_server_message_log()
```

start_raw_logging_interface

The start_raw_logging_interface method starts logging of raw data on one or all interfaces. This is for debugging purposes only.

Syntax: `start_raw_logging_interface("<Interface Name (optional)>")`

PARAMETER	DESCRIPTION
Interface Name	Name of the Interface to command to start raw data logging. Defaults to 'ALL' which causes all interfaces that support raw data logging to start logging raw data.

Example:

```
start_raw_logging_interface ("int1")
```

stop_raw_logging_interface

The stop_raw_logging_interface method stops logging of raw data on one or all interfaces. This is for debugging purposes only.

Syntax: `stop_raw_logging_interface("<Interface Name (optional)>")`

PARAMETER	DESCRIPTION
Interface Name	Name of the Interface to command to stop raw data logging. Defaults to 'ALL' which causes all interfaces that support raw data logging to stop logging raw data.

Example:

```
stop_raw_logging_interface ("int1")
```

start_raw_logging_router

The start_raw_logging_router method starts logging of raw data on one or all routers. This is for debugging purposes only.

Syntax: `start_raw_logging_router("<Router Name (optional)>")`

PARAMETER	DESCRIPTION

Router Name	Name of the Router to command to start raw data logging. Defaults to 'ALL' which causes all routers that support raw data logging to start logging raw data.

Example:

```
start_raw_logging_router("router1")
```

stop_raw_logging_router

The stop_raw_logging_router method stops logging of raw data on one or all routers. This is for debugging purposes only.

Syntax: `stop_raw_logging_router("<Router Name (optional)>")`

PARAMETER	DESCRIPTION
Router Name	Name of the Router to command to stop raw data logging. Defaults to 'ALL' which causes all routers that support raw data logging to stop logging raw data.

Example:

```
stop_raw_logging_router("router1")
```

Executing Other Procedures

These methods allow the user to bring in files of subroutines and execute other test procedures.

start

The start method starts execution of another high level test procedure. No parameters can be given to high level test procedures. If parameters are necessary, then consider using a subroutine.

Syntax: `start("<Procedure Filename>")`

PARAMETER	DESCRIPTION
Procedure Filename	Name of the test procedure file. These files are normally in the procedures folder but may be anywhere in the Ruby search path. Additionally, absolute paths are supported.

Example:

```
start("test1.rb")
```

load_utility

The load_utility method reads in a script file that contains useful subroutines for use in your test procedure. When these subroutines run in ScriptRunner or TestRunner, their lines will not be highlighted. This is very useful for methods containing loops which can be slow to execute when highlighting lines.

Syntax: `load_utility("<Utility Filename>")`

PARAMETER	DESCRIPTION
Utility Filename	Name of the script file containing subroutines. These files are normally in the procedures folder but may be anywhere in the Ruby search path. Additionally, absolute paths are supported.

Example:

```
load_utility("mode_changes.rb")
```

Opening and Closing Telemetry Screens

These methods allow the user to open or close telemetry screens from within a test procedure.

display

The display method opens a telemetry screen at the specified position.

Syntax: `display("<Display Name>", <X Position (optional)>, <Y Position (optional)>)`

PARAMETER	DESCRIPTION
Display Name	Name of the telemetry screen to display. Screens are normally named by "TARGET_NAME SCREEN_NAME"
X Position	The X coordinate on screen where the top left corner of the telemetry screen will be placed.
Y Position	The Y coordinate on screen where the top left corner of the telemetry screen will be placed.

Example:

```
display("INST ADCS", 100, 200)
```

clear

The clear method closes an open telemetry screen.

Syntax: `clear("<Display Name>")`

PARAMETER	DESCRIPTION
Display Name	Name of the telemetry screen to close. Screens are normally named by "TARGET_NAME SCREEN_NAME"

Example:

```
clear("INST ADCS")
```

Script Runner Specific Functionality

These methods allow the user to interact with ScriptRunner functions.

set_line_delay

This method sets the line delay in script runner.

Syntax: `set_line_delay(<delay>)`

PARAMETER	DESCRIPTION
delay	The amount of time script runner will wait between lines when executing a script, in seconds. Should be ≥ 0.0

Example:

```
set_line_delay(0.0)
```

get_line_delay

The method gets the line delay that script runner is currently using.

Syntax: `get_line_delay()`

Example:

```
curr_line_delay = get_line_delay()
```

get_scriptrunner_message_log_filename

Returns the filename of the ScriptRunner message log.

Syntax: `get_scriptrunner_message_log_filename()`

Example:

```
filename = get_scriptrunner_message_log_filename()
```

start_new_scriptrunner_message_log

Starts a new ScriptRunner message log. Note: ScriptRunner will automatically start a new log whenever a script is started. This method is only needed for starting a new log mid-script execution.

Syntax: `start_new_scriptrunner_message_log()`

Example:

```
filename = start_new_scriptrunner_message_log()
```

disable_instrumentation

*** Added in COSMOS 3.3.3 ***

Disables instrumentation for a block of code (line highlighting and exception catching). This is especially useful for speeding up loops that are very slow if lines are instrumented. Consider breaking code like this into a seperate file and using either require/load to read the file for the same effect while still allowing errors to be caught by your script.

*** WARNING: Use with caution. Disabling instrumentation will cause any error that occurs while disabled to cause your script to completely stop. ***

Syntax: `disable_instrumentation do`

Example:

```
disable_instrumentation do
  1000.times do
    # Don't want this to have to highlight 1000 times
  end
end
```

set_stdout_max_lines

*** Added in COSMOS 3.3.3 ***

This method sets the maximum amount of lines of output that a single line in Scriptrunner can generate without being truncated.

Syntax: `set_stdout_max_lines(max_lines)`

PARAMETER	DESCRIPTION
max_lines	The maximum number of lines that will be written to the ScriptRunner log at once

Example:

```
set_stdout_max_lines(2000)
```

Debugging

These methods allow the user to debug scripts with ScriptRunner.

insert_return

Inserts a ruby return statement into the currently executing context. This can be used to break out of methods early from the ScriptRunner Debug prompt.

Syntax: `insert_return (<return value (optional)>, ...)`

PARAMETER	DESCRIPTION
return value	One or more values that are returned from the method

Example(s):

```
insert_return()
insert_return(5, 10)
```

step_mode

Places ScriptRunner into step mode where Go must be hit to proceed to the next line.

Syntax: `step_mode()`

Example:

```
step_mode()
```

run_mode

Places ScriptRunner into run mode where the next line is run automatically.

Syntax: `run_mode()`

Example:

```
run_mode()
```

show_backtrace

Makes ScriptRunner print out a backtrace when an error occurs. Also prints out a backtrace for the most recent error.

Syntax: `show_backtrace(<true or false>)`

Example:

```
show_backtrace(true)
```

shutdown_cmd_tlm

The shutdown_cmd_tlm method disconnects from the Command and Telemetry Server. This is good practice to do before your tool shuts down.

Syntax: `shutdown_cmd_tlm()`

Example:

```
shutdown_cmd_tlm()
```

set_cmd_tlm_disconnect

The set_cmd_tlm_disconnect method puts scripting into or out of disconnect mode. In disconnect mode, messages are not sent to CmdTlmServer. Instead things are reported as nominally succeeding. Disconnect mode is useful for dry-running scripts without having a connected CmdTlmServer.

Syntax: `set_cmd_tlm_disconnect(<Disconnect>, <Config File>)`

PARAMETER	DESCRIPTION
Disconnect	True or Fase. True enters disconnect mode and False leaves it.
Config File	Command and Telemetry Server configuration file to use to simulate the CmdTlmServer. Defaults to cmd_tlm_server.txt.

Example:

```
set_cmd_tlm_disconnect(true)
```

get_cmd_tlm_disconnect

The get_cmd_tlm_disconnect method returns true if currently in disconnect mode.

Syntax: `get_cmd_tlm_disconnect()`

Example:

```
mode = get_cmd_tlm_disconnect()
```

BACK NEXT

System Configuration

Improve this page

Table of Contents

Project CRC Checking

This document provides the information necessary to configure the COSMOS Command and Telemetry Server and other top level configuration options for your unique project.

Configuration file formats for the following are provided:

- system.txt (found in config/system)
- target.txt (found in config/targets/TARGETNAME)
- cmd_tlm_server.txt (found in config/targets/TARGETNAME and config/tools/cmd_tlm_server)
- crc.txt (found in data/crc.txt)

System Configuration

The COSMOS system configuration is performed by system.txt in the config/system directory. This file declares all the targets that will be used by COSMOS as well as top level configuration information which is primarily used by the Command and Telemetry Server.

By default, all COSMOS tools use the config/system/system.txt file. However, all tools can take a custom system configuration file by passing the "–system " option to the tool when it starts. NOTE: Mixing system configuration files between tools can be confusing as some tools could be configured with more or less targets than the Command and Telemetry Server. However, this is the only way to control which targets, ports, paths, and log writers are used by the various tools.

system.txt Keywords

AUTO_DECLARE_TARGETS

If this keyword is present, COSMOS will automatically load all the target folders under config/targets into the system. The target folders must be uppercase and be named according to how COSMOS will access them. For example, if you create a config/targets/INST directory, COSMOS will create a target named 'INST' which is how it will be referenced in command and telemetry. This keyword is REQUIRED unless you individually declare your targets using the DECLARE_TARGET keyword.

Example Usage:

```
AUTO_DECLARE_TARGETS
```

DECLARE_TARGET

Declare target is used in place of AUTO_DECLARE_TARGETS to give more fine grained control over how the target folder is loaded and named within COSMOS. This is required if AUTO_DECLARE_TARGET is not present.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	The directory name which contains the target information. This must match a directory under config/targets.	Yes
Substitute Target Name	The target name in the COSMOS system. This is how the target will be referred to in scripts. If this is not given (or given as nil) the target name will be the directory name given above.	No
Target Filename	The name of the file in the target directory which contains the configuration information for the target. By default this is 'target.txt' but if you want to rename this you need to set this parameter.	No

Example Usage:

```
DECLARE_TARGET INST INST2 inst.txt
```

PORT

Port is used to set the default ports used by the Command and Telemetry Server. It is not necessary to set this option unless you wish to override the defaults (given in the example usage). Overriding ports is necessary if you want to run two Command and Telemetry Servers on the same computer simultaneously.

PARAMETER	DESCRIPTION	REQUIRED
Port Name	Port name to set. Must be one of the following: 'CTS_API', 'TLMVIEWER_API', 'CTS_PREIDENTIFIED'. CTS_API - This port is what tools connect to to communicate with the COSMOS Scripting API. TLMVIEWER_API - This port is used to remotely open and close telemetry screens in Telemetry Viewer. CTS_PREIDENTIFIED - This port provides access to a preidentified stream of all telemetry packets in the system. This is currently used by Telemetry Grapher and can be used to chain Command and Telemetry Servers together.	Yes
Port Value	Port number to use for the specified port name.	Yes

Example Usage:

```
PORT CTS_API 7777 # Default
PORT TLMVIEWER_API 7778 # Default
PORT CTS_PREIDENTIFIED 7779 # Default
```

PATH

Path is used to set the default paths used by the Command and Telemetry Server to access or create files. It is not necessary to set this option unless you wish to override the defaults (given in the example usage).

PARAMETER	DESCRIPTION	REQUIRED
Path Name	Path name to set. Must be one of the following: 'LOGS', 'TMP', 'SAVED_CONFIG', 'TABLES', 'PROCEDURES', 'HANDBOOKS'.	Yes
Path Value	File system path to use for the specified path name.	Yes

Example Usage:

```
PATH LOGS './logs' # Default location of system and tool log files
PATH TMP './tmp' # Default location of temporary marshal files
PATH SAVED_CONFIG './saved_config' # Default location of saved configurations (see note)
PATH TABLES './tables' # Default location of table files
PATH PROCEDURES './procedures' # Default location of Script procedure files
PATH HANDBOOKS './handbooks' # Default location to place handbook files
```

The PROCEDURES path must be set for Script Runner and Test Runner to locate your procedure files. You can add multiple 'PATH PROCEDURES' lines to your configuration file to set multiple locations.

The SAVED_CONFIG option sets the location of saved configuration. A saved configuration is a snapshot in time of when COSMOS parses all the target directories called out by either AUTO_DECLARE_TARGETS or DECLARE_TARGET. If something has changed in a target configuration, COSMOS will create a new directory in the saved_config folder containing this new target configuration. These directories should NOT be deleted as they provide a way for COSMOS to go back in time to a known configuration when parsing old binary log files. Only if you're sure you do not want this configuration history, e.g. you are developing your configuration and it is constantly changing, you can delete these folders.

DEFAULT_PACKET_LOG_WRITER and DEFAULT_PACKET_LOG_READER

Default packet log writer and reader are used to set the class COSMOS uses when creating and reading binary packet log files. It is not necessary to set these options unless you wish to override the defaults (given in the example usage). NOTE: You should NOT override the default without consulting a COSMOS expert as this may break the ability to

successfully read and write log files throughout the COSMOS system.

PARAMETER	DESCRIPTION	REQUIRED
Filename	Ruby file to use when instantiating a new log writer / reader	Yes

Example Usage:

```
DEFAULT_PACKET_LOG_WRITER packet_log_writer.rb # Default
DEFAULT_PACKET_LOG_READER packet_log_reader.rb # Default
```

CMD_TLM_VERSION

Cmd tlm version is used to set an arbitrary command and telemetry version string which can be accessed in telemetry. This is useful in scripts for reporting a program specific version that changes along with another configuration management system.

PARAMETER	DESCRIPTION	REQUIRED
Version	Arbitrary string	Yes

Example Usage:

```
CMD_TLM_VERSION 1.0
CMD_TLM_VERSION A
CMD_TLM_VERSION 'Version A'
```

Note that quotes are only necessary to preserve spaces in your version string.

STALENESS_SECONDS

Staleness seconds represent the number of seconds that must expire without seeing a packet before COSMOS marks it as 'stale'. This is identified in telemetry screens by all the telemetry items in the stale packet being colored purple. It is not necessary to set this option unless you wish to override the default (given in the example usage).

PARAMETER	DESCRIPTION	REQUIRED
Seconds	Integer number of seconds before packets are marked stale	Yes

Example Usage:

```
STALENESS_SECONDS 30 # Default
```

ENABLE_DNS (COSMOS 3.5.0+)

Enable DNS allows you to enable reverse DNS lookups for when tools connect to the Command and Telemetry Server's pre-identified socket or to any target using the TCPIP Server Interface. As of COSMOS 3.5.0 the default is to not use DNS.

```
Example Usage: ENABLE_DNS
```

DISABLE_DNS

Disable DNS allows you to disable reverse DNS lookups for when tools connect to the Command and Telemetry Server's pre-identified socket or to any target using the TCPIP Server Interface. This is useful when you are in an environment where DNS is not available. As of COSMOS 3.5.0 the default is to not use DNS.

```
Example Usage: DISABLE_DNS
```

ENABLE_SOUND (COSMOS 3.5.0+)

Enable sound makes any prompts that occur in ScriptRunner/TestRunner make an audible sound when they popup to alert the operator of needed input.

```
Example Usage: ENABLE_SOUND
```

ALLOW_ACCESS

Allow access provides the ability to individually permit machines to connect to the COSMOS Command and Telemetry Server. It is not necessary to set this option unless you wish to override the default (given in the example usage).

PARAMETER	DESCRIPTION	REQUIRED
Machine Name or IP Address	Machine name to allow access	Yes

Example Usage:

```
ALLOW_ACCESS ALL # Default
```

Target Configuration

Each target is self contained in a target directory named after the target and placed in the config/targets directory. In the target directory there is a configuration file named target.txt which configures the individual target.

target.txt Keywords

REQUIRE

Require is used to load additional Ruby files located in the target's lib directory. These files are typically used to implement additional functionality like custom interfaces, limits responses, complex conversions, etc.

PARAMETER	DESCRIPTION	REQUIRED
Filename	The name of the file to require. The file must be in the target's lib directory to be located.	Yes

Example Usage:

```
REQUIRE limits_response.rb
```

IGNORE_PARAMETER

Ignore Parameter is used to ignore those command parameters which are required by the command protocol but do not change and can be safely ignored by the user. A good example are the CCSDS header parameters which are required by the protocol but are fixed and therefore should not be displayed to the user (by default) as they do not require user input. The tools using the list of ignored parameters are: Command Sender - It will not display (by default) parameters in this list. It will display the list via a GUI option. Command and Telemetry Server - It will not display the ignored parameters in the command log Script Runner (and various others) - It will not display the ignored parameters when doing code completion on a command

PARAMETER	DESCRIPTION	REQUIRED
Parameter Name	The name of the parameter to ignore in various tools. For example, command sender will not display ignored parameters by default.	Yes

Example Usage:

```
IGNORE_PARAMETER CCSDS_VERSION
```

IGNORE_ITEM

Ignore Item is used by various tools to ignore telemetry items which must be present but don't change much or aren't directly useful to the user. A good example are the CCSDS header parameters which are required by the protocol but are fixed. The tools using the list of ignore items are: Command and Telemetry Server - The API routine all_item_strings does not return ignored items Script Runner - Script Audits ignore items when determining which telemetry items were checked

PARAMETER	DESCRIPTION	REQUIRED
Item Name	The name of the item to ignore in various tools. For example, Telemetry Viewer will not include these items in the screen audit.	Yes

Example Usage:

```
IGNORE_ITEM CCSDS_VERSION
```

COMMANDS and TELEMETRY

The commands and telemetry keywords are used to override the default COSMOS behavior of loading all the files located in the target's cmd_tlm directory. If any commands or telemetry keywords are used, COSMOS will no longer load all files in the target's cmd_tlm folder by default but will instead load only the files indicated by these keywords. This can be useful if you have different configurations of a target within a single target folder. It is also necessary if you don't want the files to be processed in alphabetical order.

PARAMETER	DESCRIPTION	REQUIRED
Filename	The name of the file in the cmd_tlm directory to load.	Yes

Example Usage:

```
COMMANDS inst_cmds_v2.txt
TELEMETRY inst_tlm_v2.txt
```

AUTO_SCREEN_SUBSTITUTE

This keyword causes all screens in the target to force substitute the targets name for all target names specified in the targets screen definition files. This is useful in that it allows your target to be renamed simply be changing the folder name and not having to change any internal files. Can not be used if more than one target is mentioned in a screen definition file.

Example Usage:

```
AUTO_SCREEN_SUBSTITUTE
```

Command and Telemetry Server Configuration

The Command and Telemetry Server's configuration file is found in config/tools/cmd_tlm_server. This file is used to configure the server by primarily mapping the interfaces to the targets they service.

cmd_tlm_server.txt Keywords

TITLE

Title is used to set the title of the Command and Telemetry Server's window.

PARAMETER	DESCRIPTION	REQUIRED
Title	Text to put in the title of the Server window	Yes

Example Usage:

```
TITLE "COSMOS Command and Telemetry Server"
```

PACKET_LOG_WRITER

Packet log writer is used to declare a packet log writer class and give it a name which can be referenced by an interface. This is required if you want interfaces to have their own dedicated log writers or want to combine various interfaces into a single log file. By default, COSMOS logs all data on all interfaces into a single command log and a single telemetry log. This keyword can also be used if you want to declare a different log file class to create log files. NOTE: You should NOT override the default (excluding using the meta_packet_log_writer.rb) without consulting a COSMOS expert as this may break the ability to successfully read and write log files.

PARAMETER	DESCRIPTION	REQUIRED
Log Writer Name	The name of the log writer as reference by other cmd_tlm_server keywords. This name also appears in the Logging tab on the Command and Telemetry Server.	Yes
Filename	Ruby file to use when instantiating a new log writer (packet_log_writer.rb unless you have a custom log writer)	Yes
Parameters	Optional parameters to pass to the log writer class when instantiating it. The following parameters are the ones used by the default packet_log_writer.rb class.	Class specific
Log Name	Identifier to put in the log file name. This will be prepended with a date / time stamp and appended by the log type (cmd or tlm). Thus if you specify 'cosmos' the telemetry log will result in a file name of 'YYYY_MM_DD_HH_MM_SS_cosmostlm.bin'. The default is nil which means to just use 'cmd' and 'tlm' alone in the file names.	No
Logging Enabled	Whether to start with logging enabled. The default is true.	No
Cycle Time	The amount of time in seconds before creating a new log file. The default is nil which means files will grow indefinitely.	No
Cycle Size	The size in bytes before creating a new log file. The default is 2GB.	No
Log Directory	The directory to store the log files. The default is to use the system log directory defined in system.txt by the PATH LOGS.	No
Asynchronous	Whether to spawn a new thread to write packets to the log or write the packet to the log in the interface thread. The default is true.	No

Example Usage:

```
PACKET_LOG_WRITER DEFAULT packet_log_writer.rb # Default
# The default logger filename will be <DATE>_cosmostlm.bin and will create a new log every 1MB
PACKET_LOG_WRITER DEFAULT packet_log_writer.rb cosmos true nil 1000000
# Create a logger named COSMOS_LOG which creates a new log every 5 min
PACKET_LOG_WRITER COSMOS_LOG packet_log_writer.rb cosmos true 600
```

AUTO_INTERFACE_TARGETS

Auto interface targets is used to tell COSMOS to automatically look for a cmd_tlm_server.txt file at the top level of each target directory and use this file to configure the interface for that target. This is a good way of keeping the knowledge of how to interface to a target within that target. However, if you use substitute target names (by using DECLARE_TARGET) or use different IP addresses then this will not work and you'll have to use the INTERFACE_TARGET or INTERFACE keyword.

Example Usage:

```
AUTO_INTERFACE_TARGETS
```

INTERFACE_TARGET

Interface target is used similarly to AUTO_INTERFACE_TARGETS except that it loads only the specified target's interface configuration file rather than all target configuration files.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target	Yes
Configuration File	Configuration file name which contains the interface configuration. Defaults to 'cmd_tlm_server.txt'.	No

Example Usage:

```
INTERFACE_TARGET COSMOS # Look in the COSMOS target directory for cmd_tlm_server.txt
INTERFACE_TARGET COSMOS config.txt # Look in the COSMOS target directory for config.txt
```

INTERFACE

Interface is the keyword that should be present in a target directory's cmd_tlm_server.txt file if AUTO_INTERFACE_TARGETS or INTERFACE_TARGET is used. The interface keyword can also be used directly in the config/tools/cmd_tlm_server/cmd_tlm_server.txt file.

PARAMETER	DESCRIPTION	REQUIRED
Interface Name	Name of the interface. This name will appear in the Interfaces tab of the Server and is also referenced by other keywords.	Yes
Filename	Ruby file to use when instantiating the interface. See the Interface Guide to learn more about the interfaces provided by COSMOS.	Yes
Parameters	Parameters to pass to the interface. See the Interface Guide to learn more about the interfaces provided by COSMOS.	Interface Specific

Example Usage:

```
INTERFACE COSMOS_INT cmd_tlm_server_interface.rb
```

More examples provided in the Interface Guide.

ROUTER

Router creates an interface which receives command packets from their remote targets and send them out their interfaces. They receive telemetry packets from their interfaces and send them to their remote targets. This allows routers to be intermediaries between an external client and an actual device.

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the router	Yes
Filename	Ruby file to use when instantiating the interface. See the Interface Guide to learn more about the interfaces provided by COSMOS.	Yes
Parameters	Parameters to pass to the interface. See the Interface Guide to learn more about the interfaces provided by COSMOS.	Interface Specific

COLLECT_METADATA

COLLECT_METADATA keyword prompts the user for meta data when starting the CmdTlmServer.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Target Name of the Metadata telemetry packet	Yes
Packet Name	Packet Name of the Metadata telemetry packet	Yes

Example Usage:

```
COLLECT_METADATA META META
```

BACKGROUND_TASK

Create a background task in the Server. The Server instantiates the class which must inherit from BackgroundTask and then calls the call() method which the class must implement. The call() method is only called once so if your background task is supposed to live on while the Server is running, you must implement your code in a loop with a sleep to not use all the CPU.

PARAMETER	DESCRIPTION	REQUIRED
Filename	Ruby file which contains the background task implementation. Must inherit from BackgroundTask and implement the call method.	Yes
Optional Arguments	Optional arguments to the background task constructor	No

Example Usage:

```
BACKGROUND_TASK example_background_task.rb
```

example_background_task.rb:

```
require 'cosmos/tools/cmd_tlm_server/background_task'
module Cosmos
  class ExampleBackgroundTask < BackgroundTask
    def call
      while true
        # Call COSMOS API methods
        sleep 1 # 1Hz
      end
    end
  end
end
```

Interface and Router Modifiers

The following keywords modify an interface and are only applicable after the INTERFACE or ROUTER keywords. They are indented to show ownership to the previously defined interface.

TARGET

REQUIRED and only applicable to the INTERFACE keyword. Maps a target name to this interface which causes all the command and telemetry definitions to apply to the data being processed on the interface.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Target name to map to this interface. Must match a known target name.	Yes

DONT_CONNECT

The Command and Telemetry Server will not try to connect to the given interface when starting up. Default is to connect on startup.

DONT_RECONNECT

The Command and Telemetry Server will not try to reconnect to the given interface if the connection is lost. Default is auto reconnect.

RECONNECT_DELAY

If DONT_RECONNECT is not present the Server will try to reconnect to an interface if the connection is lost. Reconnect delays sets the interval in seconds between reconnect tries.

PARAMETER	DESCRIPTION	REQUIRED
Seconds	Delay in seconds between reconnect attempts. The default is 15 seconds.	Yes

DISABLE_DISCONNECT

Disable the Disconnect button on the Interfaces tab in the Server. This prevents the user from disconnecting from the interface.

LOG

Enable logging on the interface by the specified log writer. This is only required if you want a log writer other than the default to log commands and telemetry on this interface.

PARAMETER	DESCRIPTION	REQUIRED
Log Writer Name	Log writer name as defined by PACKET_LOG_WRITER	Yes

DONT_LOG

Disable logging commands and telemetry on this interface. Note this prevents logging from the Server on the Logging tab.

LOG_RAW

Log all data on the interface exactly as it is sent and received. This does not add any COSMOS headers and thus can not be read by COSMOS tools. It is primarily useful for low level debugging of an interface.

OPTION

Pass a specific option to the interface or router.

PARAMETER	DESCRIPTION	REQUIRED
Option Name	Name of the option.	Yes
Option Value 1	Value of the option.	Yes
Additional Option Values	0 or more additional values given to the option	Option Specific

ROUTE

Only applies to routers. ROUTE declares which interfaces should use the current router. The given interface will then route all of its commands and telemetry through the router and out the interface defined by the router.

PARAMETER	DESCRIPTION	REQUIRED
Interface Name	A previously defined interface name given by the INTERFACE keyword.	Yes

Example Usage:

```
PACKET_LOG_WRITER COSMOS_LOG packet_log_writer.rb cosmos

INTERFACE COSMOS_INT cmd_tlm_server_interface.rb
  TARGET COSMOS
  DISABLE_DISCONNECT
  RECONNECT_DELAY 5
  LOG COSMOS_LOG
```

Project CRC Checking

The COSMOS Launcher will check CRCs on project files if a data/crc.txt file is present. The file is made up of filename, a space character, and the expected CRC for the file. If the user updates the file from the Launcher legal dialog, the keyword USER_MODIFIED will be added to the top. This line should be deleted for an official release.

Example File:

```
lib/example_background_task.rb 0xCF0A70AF
lib/example_target.rb 0x5B7507D3
lib/user_version.rb 0x8F282EE9
```

BACK NEXT

Command and Telemetry Configuration

Improve this page

Table of Contents

APPEND_ARRAY_ITEM
SELECT_ITEM
MACRO_APPEND_START and MACRO_APPEND_END
META
PROCESSOR
ALLOW_SHORT

Item Modifiers

FORMAT_STRING
UNITS
DESCRIPTION
STATE
READ_CONVERSION
POLY_READ_CONVERSION
SEG_POLY_READ_CONVERSION
GENERIC_READ_CONVERSION_START and GENERIC_READ_CONVERSION_END
LIMITS
LIMITS_RESPONSE
META

Example File

Command Definition Files

Command definition files define the command packets that can be sent to COSMOS targets. One large file can be used to define the command packets, or multiple files can be used at the user's discretion. Command definition files are placed in the config/TARGET/cmd_tlm directory and are processed alphabetically. Therefore if you have some command files that depend on others, e.g. they override or extend existing commands, they must be named last. Due to the way the ASCII Table is structured, files beginning with capital letters are processed before lower case letters. To force a file to be processed last either prepend it with 'z' or '~'.

Command Keywords:

COMMAND

The COMMAND keyword designates the start of a new command packet.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target this command is associated with	Yes
Command Name	Name of this command. Also referred to as its mnemonic. Must be unique to commands to this target. Ideally will be as short and clear as possible.	Yes
BIG_ENDIAN or LITTLE_ENDIAN	Indicates if the data in this command is to be sent in Big Endian or Little Endian format.	Yes
Description	Description of this command. Must be enclosed with '"	No

Example Usage:

```
COMMAND COSMOS START_LOGGING BIG_ENDIAN "Starts logging"
```

SELECT_COMMAND

The SELECT_COMMAND keyword selects an existing command packet for editing.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target this command is associated with	Yes
Command Name	Name of this command. Also referred to as its mnemonic.	Yes

Example Usage:

```
SELECT_COMMAND COSMOS START_LOGGING
```

Command Modifiers

The following keywords modify a command and are only applicable after the COMMAND or SELECT_COMMAND keywords. They are typically indented within the definition file to show ownership to the previously defined command.

HAZARDOUS

Designates the current command as a hazardous command. This affects scripts and the Command Sender tool by popping up a dialog asking for confirmation before sending the command.

PARAMETER	DESCRIPTION	REQUIRED
Description	Why the command is hazardous. Must be enclosed with ""	No

PARAMETER

The PARAMETER keyword defines a command parameter in the current command.

<table>
<tr><th>PARAMETER</th><th>DESCRIPTION</th><th>REQUIRED</th></tr>
<tr><td>Name</td><td>Name of the parameter. Must be unique within the command.</td><td>Yes</td></tr>
<tr><td>Bit Offset</td><td>Bit offset into the command packet of the Most Significant Bit of this parameter. May be negative to indicate on offset from the end of the packet. Always use a bit offset of 0 for derived parameters.</td><td>Yes</td></tr>
<tr><td>Bit Size</td><td>Bit size of this parameter. Zero or Negative values may be used to indicate that a string fills the packet up to the offset from the end of the packet specified by this value. If Bit Offset = 0 and Bit Size = 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.</td><td>Yes</td></tr>
<tr><td>Data Type</td><td>Data Type of this parameter. Possible types: INT = Integer, UINT = Unsigned Integer, FLOAT = IEEE Floating point data, STRING = Character string data, BLOCK = Non-Ascii Data Block, DERIVED = Bit Offset and Bit Size of 0.</td><td>Yes</td></tr>
<tr><td>Minimum Value</td><td>Minimum allowed value for this parameter (Not given if Data Type = STRING or BLOCK) The following special constants may be given for common values:
<table>
<tr><th>CONSTANT</th><th>VALUE</th></tr>
<tr><td>MIN_INT8</td><td>-128</td></tr>
<tr><td>MAX_INT8</td><td>127</td></tr>
<tr><td>MIN_INT16</td><td>-32768</td></tr>
<tr><td>MAX_INT16</td><td>32767</td></tr>
<tr><td>MIN_INT32</td><td>-2147483648</td></tr>
<tr><td>MAX_INT32</td><td>2147483647</td></tr>
<tr><td>MIN_INT64</td><td>-9223372036854775808</td></tr>
<tr><td>MAX_INT64</td><td>9223372036854775807</td></tr>
<tr><td>MIN_UINT8</td><td>0</td></tr>
<tr><td>MAX_UINT8</td><td>255</td></tr>
<tr><td>MIN_UINT16</td><td>0</td></tr>
<tr><td>MAX_UINT16</td><td>65535</td></tr>
</table></td><td>Yes</td></tr>
</table>

	MIN_UINT32	0
	MAX_UINT32	4294967295
	MIN_UINT64	0
	MAX_UINT64	18446744073709551615
	MIN_FLOAT32	-3.402823e38
	MAX_FLOAT32	3.402823e38
	MIN_FLOAT64	-1.7976931348623157e308
	MAX_FLOAT64	1.7976931348623157e308
	NEG_INFINITY	-Float::INFINITY
	POS_INFINITY	Float::INFINITY
Maximum Value	Maximum allowed value for this parameter (Not given if Data Type = STRING or BLOCK) See PARAMETER#Minimum Value for a list of special constants like MIN_UINT16 that may be used for this field.	Yes
Default Value	Default value for this parameter. You must provide a default but if you mark the parameter REQUIRED then scripts will be forced to specify a value. See PARAMETER#Minimum Value for a list of special constants like MIN_UINT16 that may be used for this field.	Yes
Description	Description for this parameter. Must be enclosed with ""	No

Example Usage:

```
PARAMETER SYNC 0 32 UINT 0xDEADBEEF 0xDEADBEEF 0xDEADBEEF "Sync pattern"
PARAMETER VALUE 32 32 FLOAT 0 10.5 2.5
PARAMETER LABEL 64 0 STRING "COSMOS" "The label to apply"</pre>
```

APPEND_PARAMETER

The APPEND_PARAMETER keyword appends a command parameter to the end of the current command. Parameter details are the same as for the PARAMETER keyword except Bit Offset is not used. This is the preferred way to declare parameters as it allows for adding and removing parameters without having to recalculate bit offsets for all other parameters.

Example Usage:

```
APPEND_PARAMETER SYNC 32 UINT 0xDEADBEEF 0xDEADBEEF 0xDEADBEEF"Sync pattern"
APPEND_PARAMETER VALUE 32 FLOAT 0 10.5 2.5
APPEND_PARAMETER LABEL 0 STRING "COSMOS" "The label to apply"
```

ID_PARAMETER

Much like the PARAMETER keyword, the ID_PARAMETER keyword defines a command parameter in the current command. In addition, ID_PARAMETER(s) are used to identify a command given a binary array of data. A command packet may have one or more ID_PARAMETERs, all of which must match the binary data for the command to be identified.

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the parameter. Must be unique within the command.	Yes
Bit Offset	Bit offset into the command packet of the Most Significant Bit of this parameter. May be negative to indicate on offset from the end of the packet. Always use a bit offset of 0 for derived parameters.	Yes

Bit Size	Bit size of this parameter. Zero or Negative values may be used to indicate that a string fills the packet up to the offset from the end of the packet specified by this value. If Bit Offset = 0 and Bit Size = 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	Yes
Data Type	Data Type of this parameter. Possible types: INT = Integer, UINT = Unsigned Integer, FLOAT = IEEE Floating point data, STRING = Character string data, BLOCK = Non-Ascii Data Block, DERIVED = Bit Offset and Bit Size of 0.	Yes
Minimum Value	Minimum allowed value for this parameter (Not given if Data Type = STRING or BLOCK) See PARAMETER#Minimum Value for a list of special constants like MIN_UINT16 that may be used for this field.	Yes
Maximum Value	Maximum allowed value for this parameter (Not given if Data Type = STRING or BLOCK) See PARAMETER#Minimum Value for a list of special constants like MIN_UINT16 that may be used for this field.	Yes
ID Value	Identification value for this parameter. The binary data must match this value for the buffer to be identified as this packet.	Yes
Description	Description for this parameter. Must be enclosed with '"	No

Example Usage:

```
ID_PARAMETER OPCODE 32 32 UINT 2 2 2 "Opcode identifier"
```

APPEND_ID_PARAMETER

The APPEND_ID_PARAMETER keyword appends an ID command parameter to the end of the current command. Parameter details are the same as for the ID_PARAMETER keyword except Bit Offset is not used.

Example Usage:

```
APPEND_ID_PARAMETER OPCODE 32 UINT 2 2 2 "Opcode identifier"
```

ARRAY_PARAMETER

The ARRAY_PARAMETER keyword defines a command parameter in the current command that is an array.

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the parameter. Must be unique within the command.	Yes
Bit Offset	Bit offset into the command packet of the Most Significant Bit of this parameter. May be negative to indicate on offset from the end of the packet. Always use a bit offset of 0 for derived parameters.	Yes
Bit Size of Each Item	Bit size of each array item. Must be greater than or equal to 0.	Yes
Data Type of Each Item	Data Type of each array item. Possible types: INT = Integer, UINT = Unsigned Integer, FLOAT = IEEE Floating point data, STRING = Character string data, BLOCK = Non-Ascii Data Block, DERIVED = Bit Offset and Bit Size of 0.	Yes
Total Bit Size of Array	Total Bit Size of the Array. Zero or Negative values may be used to indicate the array fills the packet up to the offset from the end of the packet specified by this value.	Yes
Description	Description for this parameter. Must be enclosed with '".	No

Example Usage:

```
ARRAY_PARAMETER ARRAY 64 64 FLOAT 640 "Array of 10 64bit floats"
```

APPEND_ARRAY_PARAMETER

The APPEND_ARRAY_PARAMETER keyword appends an array command parameter to the end of the current command. Parameter details are the same as for the ARRAY_PARAMETER keyword except Bit Offset is not used.

Example Usage:

```
APPEND_ARRAY_PARAMETER ARRAY 64 FLOAT 640 "Array of 10 64bit floats"
```

SELECT_PARAMETER

The SELECT_PARAMETER keyword selects an existing command parameter for editing in the current command.

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the parameter to select for modification	Yes

Example Usage:

```
SELECT_COMMAND COSMOS START_LOGGING
  SELECT_PARAMETER LABEL
```

MACRO_APPEND_START and MACRO_APPEND_END

The MACRO_APPEND_START keyword is used to create a list of command parameters which are appended to the current command. Each of these parameters will be repeated with a number appended to their names to form a list of command parameters with a unique mnemonic for each parameter.

PARAMETER	DESCRIPTION	REQUIRED
First Number in Range	First value that will be appended to the parameter mnemonic	Yes
Last Number in Range	Last number that will be appended to the parameter mnemonic	Yes
Format String	An optional format string that defaults to “%s%d”. Where the %s represents the constant portion of the items mnemonic and the %d represents the number in the range that is appended. For example, with a range of 1 to 2 and two items named VALUE and DATA, the format string “%s_%d” would create the mnemonics VALUE_1, DATA_1, VALUE_2, and DATA_2.	No

Example Usage:

```
MACRO_APPEND_START 1 2 "%s_%d"
  APPEND_PARAMETER VALUE 16 UINT 0 5 1 "Value"
  APPEND_PARAMETER DATA 16 UINT 0 5 2 "Data"
MACRO_APPEND_END
```

This results in a command with the following mnemonics: VALUE_1, DATA_1, VALUE_2, and DATA_2. All VALUE parameters will have the same default of 1 and all DATA parameters have the same default of 2. Note that due to the dynamic nature of this keyword you must use the APPEND variation of the PARAMETER keywords.

DISABLE_MESSAGES

Disable the COSMOS Command and Telemetry Server from printing cmd(…) messages for this command. Commands are still logged.

META

The META keyword stores metadata for the current command that can be used by custom tools for various purposes (For example to store additional information needed to generate FSW header files).

PARAMETER	DESCRIPTION	REQUIRED
Meta Name	Name of the metadata to store	Yes
Meta Values	One or more values to be stored for this Meta Name	No

Example Usage:

```
META FSW_TYPE "struct command"
```

Parameter Modifiers

The following keywords modify a parameter and are only applicable after the various PARAMETER keywords as defined above. They are typically indented within the definition file to show ownership to the previously defined parameter.

REQUIRED

Declares that when sending the command via Script Runner a value must always be given for the current command parameter. This prevents the user from relying on a default value. Note that this does not affect Command Sender which will still populate the field with the default value provided in the PARAMETER definition.

FORMAT_STRING

The FORMAT_STRING keyword adds printf style formatting to a command parameter. This can be used to set how default command parameter values are displayed in Command Sender.

PARAMETER	DESCRIPTION	REQUIRED
Format	How to format the command parameter. For example: "0x%0X" will display a parameter in hex.	Yes

Example Usage:

```
FORMAT_STRING "0x%0X"
```

UNITS

The UNITS keyword add units knowledge to a parameter.

PARAMETER	DESCRIPTION	REQUIRED
Full Name	Full name of the units type. For example: Celcius	Yes
Abbreviated Name	Abbreviation for the units. For example: C	Yes

Example Usage:

```
UNITS Celcius C
UNITS Kilometers KM
```

MINIMUM_VALUE, MAXIMUM_VALUE, DEFAULT_VALUE, DESCRIPTION

These keywords allow you to override an existing value. This is useful for changing things in conjunction with SELECT_COMMAND and SELECT_PARAMETER.

PARAMETER	DESCRIPTION	REQUIRED
Value	The new value to replace the previously defined value	Yes

Example Usage:

```
SELECT_COMMAND COSMOS START_LOGGING
  SELECT_PARAMETER OPCODE
    MINIMUM_VALUE 1
    MAXIMUM_VALUE 1
    DEFAULT_VALUE 1
    DESCRIPTION "Opcode is the command identifier"
```

STATE

The STATE keyword defines a key/value pair for the current command parameter. For example, you might define states for ON = 1 and OFF = 0. This allows the word ON to be used rather than the number 1 when sending the command parameter and allows for much greater clarity and less chance for user error.

PARAMETER	DESCRIPTION	REQUIRED
Key	The state name	Yes
Value	The state value	Yes
HAZARDOUS	Keyword the indicates the state is hazardous. This will cause a popup to ask for user confirmation when sending this command.	No
Hazardous Description	Description about why this state is hazardous.	No

Example Usage:

```
APPEND_PARAMETER ENABLE 32 UINT 0 1 0 "Enable setting"
  STATE FALSE 0
  STATE TRUE 1
APPEND_PARAMETER STRING 1024 STRING "NOOP" "String parameter"
  STATE "NOOP" "NOOP"
  STATE "ARM LASER" "ARM LASER" HAZARDOUS "Arming the laser is an eye safety hazard"
  STATE "FIRE LASER" "FIRE LASER" HAZARDOUS "WARNING! Laser will be fired!"
```

WRITE_CONVERSION

The WRITE_CONVERSION keyword applies a conversion to the current command parameter. This conversion is implemented in a custom Ruby file which should be located in the target's lib folder and required by the target's target.txt file. See the documentation in . The class must require 'cosmos/conversions/conversion' and inherit from Conversion. It must implement the initialize method if it takes extra parameters and must always implement the call method. The conversion factor is applied to the value entered by the user before it is written into the binary command packet and sent.

PARAMETER	DESCRIPTION	REQUIRED
Class file name	The file name which contains the Ruby class. The file name must be named after the class such that the class is a CamelCase version of the underscored file name. For example: 'the_great_conversion.rb' should contain 'class TheGreatConversion'.	Yes
Param X	Parameter #x. Additional parameter values for the conversion which are passed to the class constructor.	No

Example Usage:

```
WRITE_CONVERSION the_great_conversion.rb 1000
```

the_great_conversion.rb:

```
require 'cosmos/conversions/conversion'
module Cosmos
  class TheGreatConversion &lt; Conversion
    def initialize(multiplier)
      super()
      @multiplier = multiplier
    end
    def call(value, packet, buffer)
      return value * multiplier
    end
  end
end
```

POLY_WRITE_CONVERSION

The POLY_WRITE_CONVERSION keyword adds a polynomial conversion factor to the current command parameter. This conversion factor is applied to the value entered by the user before it is written into the binary command packet and sent.

PARAMETER	DESCRIPTION	REQUIRED
C0	Coefficient #0	Yes
Cx	Coefficient #x. Additional coefficient values for the conversion. Any order polynomial conversion may be used so the value of 'x' will vary with the order of the polynomial. Note that larger order polynomials take longer to process than shorter order polynomials, but are sometimes more accurate.	No

Example Usage:

```
POLY_WRITE_CONVERSION 10 0.5 0.25
```

SEG_POLY_WRITE_CONVERSION

The SEG_POLY_WRITE_CONVERSION keyword adds a segmented polynomial conversion factor to the current command parameter. This conversion factor is applied to the value entered by the user before it is written into the binary command packet and sent.

PARAMETER	DESCRIPTION	REQUIRED
Lower Bound	Defines the lower bound of the range of values that this segmented polynomial applies to. Is ignored for the segment with the smallest lower bound.	Yes
C0	Coefficient #0	Yes
Cx	Coefficient #x. Additional coefficient values for the conversion. Any order polynomial conversion may be used so the value of 'x' will vary with the order of the polynomial. Note that larger order polynomials take longer to process than shorter order polynomials, but are sometimes more accurate.	No

Example Usage:

```
SEG_POLY_WRITE_CONVERSION 0 10 0.5 0.25 # Apply the conversion to all values < 50
SEG_POLY_WRITE_CONVERSION 50 11 0.5 0.275 # Apply the conversion to all values >= 50 and < 100
SEG_POLY_WRITE_CONVERSION 100 12 0.5 0.3 # Apply the conversion to all values >= 100
```

GENERIC_WRITE_CONVERSION_START and GENERIC_WRITE_CONVERSION_END

**NOTE: Generic conversions are not a good long term solution. Consider creating a conversion class and using WRITE_CONVERSION instead. WRITE_CONVERSION is easier to debug and higher performance. **

The GENERIC_WRITE_CONVERSION_START keyword adds a generic conversion function to the current command parameter. This conversion factor is applied to the value entered by the user before it is written into the binary command packet and sent. The conversion is specified as ruby code that receives two implied parameters: 'value' which is the raw value being written, and 'packet' which is a reference to the command packet class (Note: referencing the packet as 'myself' is still supported for backwards compatibility). The last line of ruby code given should return the converted value. The GENERIC_WRITE_CONVERSION_END keyword specifies that all lines of ruby code for the conversion have been given.

Example Usage:

```
APPEND_PARAMETER ITEM1 32 UINT 0 0xFFFFFFFF 0
  GENERIC_WRITE_CONVERSION_START
    (value * 1.5).to_i # Convert the value by a scale factor
  GENERIC_WRITE_CONVERSION_END
```

Note: If you need to apply a conversion that depends on another parameter being set then special code must be added to check the other parameter's value. COSMOS processes the parameters as a Hash and the order or processing is NOT guaranteed. For example:

```
COMMAND INST SETPOINT BIG_ENDIAN "Control Setpoint"
  ..
  APPEND_PARAMETER ZONE 8 UINT 1 10 1 "Heater zone"
    STATE ONE 1
    STATE TWO 2
  APPEND_PARAMETER SETPOINT 32 FLOAT 0 1000 0 "Setpoint"
    GENERIC_WRITE_CONVERSION_START
      result = nil
      case myself.given_values['ZONE'] # Access the zone value
      when 'ONE'
        result = value * 1
      when 'TWO'
        result = value * 2
      end
      return result
    GENERIC_WRITE_CONVERSION_END
```

META

The META keyword stores metadata for the current command parameter that can be used by custom tools for various purposes (For example to store additional information needed to generate FSW header files).

PARAMETER	DESCRIPTION	REQUIRED
Meta Name	Name of the metadata to store	Yes
Meta Values	One or more values to be stored for this Meta Name	No

Example Usage:

```
META TEST "This parameter is for test purposes only"
```

Example File

Example File: <COSMOSPATH>/config/MY_TARGET/cmd_tlm/cmds.txt

```
COMMAND MY_TARGET COLLECT_DATA BIG_ENDIAN "Commands my target to collect data"
  PARAMETER CCSDSVER 0 3 UINT 0 0 0 "CCSDS PRIMARY HEADER VERSION NUMBER"
  PARAMETER CCSDSTYPE 3 1 UINT 1 1 1 "CCSDS PRIMARY HEADER PACKET TYPE"
  PARAMETER CCSDSSHF 4 1 UINT 0 0 0 "CCSDS PRIMARY HEADER SECONDARY HEADER FLAG"
  ID_PARAMETER CCSDSAPID 5 11 UINT 0 2047 100 "CCSDS PRIMARY HEADER APPLICATION ID"
  PARAMETER CCSDSSEQFLAGS 16 2 UINT 3 3 3 "CCSDS PRIMARY HEADER SEQUENCE FLAGS"
  PARAMETER CCSDSSEQCNT 18 14 UINT 0 16383 0 "CCSDS PRIMARY HEADER SEQUENCE COUNT"
  PARAMETER CCSDSLENGTH 32 16 UINT 4 4 4 "CCSDS PRIMARY HEADER PACKET LENGTH"
  PARAMETER ANGLE 48 32 FLOAT -180.0 180.0 0.0 "ANGLE OF INSTRUMENT IN DEGREES"
    POLY_WRITE_CONVERSION 0 0.01745 0 0
  PARAMETER MODE 80 8 UINT 0 1 0 "DATA COLLECTION MODE"
    STATE NORMAL 0
    STATE DIAG 1

COMMAND MY_TARGET NOOP BIG_ENDIAN "Do Nothing"
  PARAMETER CCSDSVER 0 3 UINT 0 0 0 "CCSDS PRIMARY HEADER VERSION NUMBER"
  PARAMETER CCSDSTYPE 3 1 UINT 1 1 1 "CCSDS PRIMARY HEADER PACKET TYPE"
  PARAMETER CCSDSSHF 4 1 UINT 0 0 0 "CCSDS PRIMARY HEADER SECONDARY HEADER FLAG"
  ID_PARAMETER CCSDSAPID 5 11 UINT 0 2047 101 "CCSDS PRIMARY HEADER APPLICATION ID"
  PARAMETER CCSDSSEQFLAGS 16 2 UINT 3 3 3 "CCSDS PRIMARY HEADER SEQUENCE FLAGS"
  PARAMETER CCSDSSEQCNT 18 14 UINT 0 16383 0 "CCSDS PRIMARY HEADER SEQUENCE COUNT"
  PARAMETER CCSDSLENGTH 32 16 UINT 0 0 0 "CCSDS PRIMARY HEADER PACKET LENGTH"
  PARAMETER DUMMY 48 8 UINT 0 0 0 "DUMMY PARAMETER BECAUSE CCSDS REQUIRES 1 BYTE OF DATA"

COMMAND MY_TARGET SETTINGS BIG_ENDIAN "Set the Settings"
  PARAMETER CCSDSVER 0 3 UINT 0 0 0 "CCSDS PRIMARY HEADER VERSION NUMBER"
  PARAMETER CCSDSTYPE 3 1 UINT 1 1 1 "CCSDS PRIMARY HEADER PACKET TYPE"
  PARAMETER CCSDSSHF 4 1 UINT 0 0 0 "CCSDS PRIMARY HEADER SECONDARY HEADER FLAG"
  ID_PARAMETER CCSDSAPID 5 11 UINT 0 2047 102 "CCSDS PRIMARY HEADER APPLICATION ID"
  PARAMETER CCSDSSEQFLAGS 16 2 UINT 3 3 3 "CCSDS PRIMARY HEADER SEQUENCE FLAGS"
  PARAMETER CCSDSSEQCNT 18 14 UINT 0 16383 0 "CCSDS PRIMARY HEADER SEQUENCE COUNT"
  PARAMETER CCSDSLENGTH 32 16 UINT 0 0 0 "CCSDS PRIMARY HEADER PACKET LENGTH"
  MACRO_APPEND_START 1 5
    APPEND_PARAMETER SETTING 16 UINT 0 5 0 "Setting #x"
  MACRO_APPEND_END
```

Telemetry Definition Files

Telemetry definition files define the telemetry packets that can be received and processed from COSMOS targets. One large file can be used to define the telemetry packets, or multiple files can be used at the user's discretion. Telemetry definition files are placed in the config/TARGET/cmd_tlm directory and are processed alphabetically. Therefore if you have some telemetry files that depend on others, e.g. they override or extend existing telemetry, they must be named last. Due to the way the ASCII Table is structured, files beginning with capital letters are processed before lower case letters. To force a file to be processed last either prepend it with 'z' or '~'.

Telemetry Keywords:

TELEMETRY

The TELEMETRY keyword designates the start of a new telemetry packet.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target this command is associated with	Yes

Packet Name	Name of this telemetry packet. Also referred to as its mnemonic. Must be unique to telemetry packets in this target. Ideally will be as short and clear as possible.	Yes
BIG_ENDIAN or LITTLE_ENDIAN	Indicates if the data in this packet is in Big Endian or Little Endian format.	Yes
Description	Description of this telemetry packet. Must be enclosed with ""	No

Example Usage:

```
TELEMETRY COSMOS VERSION BIG_ENDIAN "Version information"
```

SELECT_TELEMETRY

The SELECT_TELEMETRY keyword selects an existing telemetry packet for editing.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target this telemetry packet is associated with	Yes
Packet Name	Name of this telemetry packet	Yes

Example Usage:

```
SELECT_TELEMETRY COSMOS VERSION
```

LIMITS_GROUP

The LIMITS_GROUP keyword defines a related group of limits that can be enabled and disabled together. It can be used to group related limits as a subsystem that can be enabled or disabled as that particular subsystem is powered (for example). To enable a group call the enable_limits_group("NAME") method in Script Runner. To disable a group call the disable_limits_group("NAME") in Script Runner. Items can belong to multiple groups but the last enabled or disabled group "wins". For example, if an item belongs to GROUP1 and GROUP2 and you first enable GROUP1 and then disable GROUP2 the item will be disabled. If you then enable GROUP1 again it will be enabled.

PARAMETER	DESCRIPTION	REQUIRED
Group Name	Name of the limits group	Yes

LIMITS_GROUP_ITEM

The LIMITS_GROUP_ITEM keyword adds the specified telemetry item to the last defined LIMITS_GROUP.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target	Yes
Packet Name	Name of the packet	Yes
Item Name	Name of the telemetry item to add to the group	Yes

Example Usage:

```
LIMITS_GROUP SUBSYSTEM
  LIMITS_GROUP_ITEM INST HEALTH_STATUS TEMP1
  LIMITS_GROUP_ITEM INST HEALTH_STATUS TEMP2
  LIMITS_GROUP_ITEM INST HEALTH_STATUS TEMP3
```

This information is typically kept in a separate configuration file in the config/TARGET/cmd_tlm folder named limits_groups.txt. If you want to configure multiple target items in a particular group you should put this information in the config/SYSTEM/cmd_tlm/limits_groups.txt file. The SYSTEM target is processed last and contains information that

crosses target boundaries.

Telemetry Modifiers

The following keywords modify a telemetry packet and are only applicable after the TELEMETRY or SELECT_TELEMETRY keywords. They are typically indented within the definition file to show ownership to the previously defined telemetry packet.

ITEM

The ITEM keyword defines a telemetry item in the current telemetry packet.

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the telemetry item. Also referred to as its mnemonic. Must be unique within the packet.	Yes
Bit Offset	Bit offset into the telemetry packet of the Most Significant Bit of this parameter. May be negative to indicate on offset from the end of the packet. Always use a bit offset of 0 for derived parameters.	Yes
Bit Size	Bit size of this telemetry item. Zero or Negative values may be used to indicate that a string fills the packet up to the offset from the end of the packet specified by this value. If Bit Offset = 0 and Bit Size = 0 then this is a derived item and the Data Type must be set to 'DERIVED'.	Yes
Data Type	Data Type of this telemetry item. Possible types: INT = Integer, UINT = Unsigned Integer, FLOAT = IEEE Floating point data, STRING = Character string data, BLOCK = Non-Ascii Data Block, DERIVED = Bit Offset and Bit Size of 0.	Yes
Description	Description for this telemetry item. Must be enclosed with "".	No

Example Usage:

```
ITEM PKTID 112 16 UINT "Packet ID"
```

APPEND_ITEM

The APPEND_ITEM keyword appends a new telemetry item to the end of the current telemetry packet. Parameter details are the same as the ITEM keyword except that Bit Offset is not used. This is the preferred way to declare items as it allows for adding and removing items without having to recalculate bit offsets for all other items.

Example Usage:

```
APPEND_ITEM PKTID 16 UINT "Packet ID"
```

ID_ITEM

Much like the ITEM keyword, the ID_ITEM keyword defines a telemetry item in the current telemetry packet. However, ID_ITEMs are used to identify a telemetry packet from a binary array of data. The COSMOS Commmand and Telemetry Server identifies packets after they are received by an interface. A telemetry packet may have one or more ID_ITEMS, all of which must match the binary data for the telemetry packet to be identified.

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the telemetry item. Also referred to as its mnemonic. Must be unique within the packet.	Yes
Bit Offset	Bit offset into the telemetry packet of the Most Significant Bit of this parameter. May be negative to indicate on offset from the end of the packet. Always use a bit offset of 0 for derived parameters.	Yes
Bit Size	Bit size of this telemetry item. Zero or Negative values may be used to indicate that a string fills the packet up to the offset from the end of the packet specified by this	Yes

Bit Size	value. If Bit Offset = 0 and Bit Size = 0 then this is a derived item and the Data Type must be set to 'DERIVED'.	Yes
Data Type	Data Type of this telemetry item. Possible types: INT = Integer, UINT = Unsigned Integer, FLOAT = IEEE Floating point data, STRING = Character string data, BLOCK = Non-Ascii Data Block, DERIVED = Bit Offset and Bit Size of 0.	Yes
ID Value	The value of this telemetry item that uniquely identifies this telemetry packet	Yes
Description	Description for this telemetry item. Must be enclosed with "".	No

Example Usage:

```
ID_ITEM PKTID 112 16 UINT 1 "Packet ID which must be 1"
```

APPEND_ID_ITEM

The APPEND_ID_ITEM keyword appends a new id telemetry item to the end of the current telemetry packet. Parameter details are the same as the ID_ITEM keyword except that Bit Offset is not used.

Example Usage:

```
APPEND_ID_ITEM PKTID 16 UINT 1 "Packet ID which must be 1"
```

ARRAY_ITEM

The ARRAY_ITEM keyword defines a telemetry item in the current telemetry packet that is an array.

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the telemetry item. Also referred to as its mnemonic. Must be unique within the packet.	Yes
Bit Offset	Bit offset into the telemetry packet of the Most Significant Bit of this parameter. May be negative to indicate on offset from the end of the packet. Always use a bit offset of 0 for derived parameters.	Yes
Bit Size of Each Item	Bit size of each array item. Must be greater than or equal to 0. If Bit Offset = 0 and Bit Size = 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	Yes
Data Type of Each Item	Data Type of each array item. Possible types: INT = Integer, UINT = Unsigned Integer, FLOAT = IEEE Floating point data, STRING = Character string data, BLOCK = Non-Ascii Data Block, DERIVED = Bit Offset and Bit Size of 0.	Yes
Total Bit Size of Array	Total Bit Size of the Array. Zero or Negative values may be used to indicate the array fills the packet up to the offset from the end of the packet specified by this value.	Yes
Description	Description for this parameter. Must be enclosed with "".	No

Example Usage:

```
ARRAY_ITEM 64 32 FLOAT 320 "Array of 10 floats"
```

APPEND_ARRAY_ITEM

The APPEND_ARRAY_ITEM keyword appends a new array telemetry item to the end of the current telemetry packet. Parameter details are the same as the ARRAY_ITEM keyword except that Bit Offset is not used.

Example Usage:

```
APPEND_ARRAY_ITEM 32 FLOAT 320 "Array of 10 floats"
```

SELECT_ITEM

The SELECT_ITEM keyword selects an existing telemetry item for editing in the current telemetry packet.

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the telemetry item to select for modification	Yes

Example Usage:

```
SELECT_TELEMETRY COSMOS VERSION
  SELECT_ITEM RUBY
```

MACRO_APPEND_START and MACRO_APPEND_END

The MACRO_APPEND_START keyword is used to create a list of telemetry items which are appended to the current telemetry packet. Each of these items will be repeated with a number appended to their names to form a list of items with a unique mnemonic for each.

PARAMETER	DESCRIPTION	REQUIRED
First Number in Range	First value that will be appended to the telemetry item mnemonic	Yes
Last Number in Range	Last number that will be appended to the telemetry item mnemonic	Yes
Format String	An optional format string that defaults to "%s%d". Where the %s represents the constant portion of the items mnemonic and the %d represents the number in the range that is appended. For example, with a range of 1 to 2 and two items named VALUE and DATA, the format string "%s_%d" would create the mnemonics VALUE_1, DATA_1, VALUE_2, and DATA_2.	No

Example Usage:

```
MACRO_APPEND_START 1 2 "%s_%d"
  APPEND_ITEM VALUE 16 UINT "Value"
  APPEND_ITEM DATA 16 UINT "Data"
MACRO_APPEND_END
```

This results in a telemtry packet with the following mnemonics: VALUE_1, DATA_1, VALUE_2, and DATA_2. Note that due to the dynamic nature of this keyword you must use the APPEND variation of the ITEM keywords.

META

The META keyword stores metadata for the current telemetry packet that can be used by custom tools for various purposes (For example to store additional information needed to generate FSW header files).

PARAMETER	DESCRIPTION	REQUIRED
Meta Name	Name of the metadata to store	Yes
Meta Values	One or more values to be stored for this Meta Name	No

Example Usage:

```
META FSW_TYPE "struct tlm_packet"
```

PROCESSOR

The PROCESSOR keyword defines a processor class that execute code every time a packet is received.

PARAMETER	DESCRIPTION	REQUIRED
Processor Name	The name of the processor	Yes
Processor Class Filename	Name of the Ruby file which implements the processor. This file should be in the config/TARGET/lib directory so it can be found by COSMOS.	Yes
Processor Specific Options	Variable length number of options that will be passed to the class constructor.	Processor Specific

Example Usage:

```
PROCESSOR TEMP1HIGH watermark_processor.rb TEMP1
```

ALLOW_SHORT

Allows the telemetry packet to be received with a data portion that is smaller than the defined size without warnings. Any extra space in the packet will be filled in with zeros by COSMOS.

Item Modifiers

The following keywords modify a telemetry item and are only applicable after the various ITEM keywords as defined above. They are typically indented within the definition file to show ownership to the previously defined telemetry item.

FORMAT_STRING

The FORMAT_STRING keyword adds printf style formatting to a telemetry item.

PARAMETER	DESCRIPTION	REQUIRED
Format	How to format the command parameter. For example: "0x%0X" will display an item in hex.	Yes

Example Usage:

```
FORMAT_STRING "0x%0X"
```

UNITS

The UNITS keyword add units knowledge to a telemetry item.

PARAMETER	DESCRIPTION	REQUIRED
Full Name	Full name of the units type. For example: Celcius	Yes
Abbreviated Name	Abbreviation for the units. For example: C	Yes

Example Usage:

```
UNITS Celcius C
UNITS Kilometers KM
```

DESCRIPTION

The DESCRIPTION keywords allow you to override an existing description. This is useful for changing things in conjunction with SELECT_TELEMETRY and SELECT_ITEM.

PARAMETER	DESCRIPTION	REQUIRED
Description	The new description	Yes

Example Usage:

```
SELECT_TELEMETRY COSMOS VERSION
  SELECT_PARAMETER RUBY
    DESCRIPTION "The Ruby version"
```

STATE

The STATE keyword defines a key/value pair for the current telemetry item. For example, you might define states for ON = 1 and OFF = 0. This allows the word ON to be used rather than the number 1 when checking the telemetry item and allows for much greater clarity and less chance for user error.

PARAMETER	DESCRIPTION	REQUIRED
Key	The state name	Yes
Value	The state value	Yes
Color	The color the state should be displayed as. Default is black. Choices are GREEN, YELLOW, RED.	No

Example Usage:

```
APPEND_ITEM ENABLE 32 UINT "Enable setting"
  STATE FALSE 0
  STATE TRUE 1
APPEND_ITEM STRING 1024 STRING "String"
  STATE "NOOP" "NOOP" GREEN
  STATE "ARM LASER" "ARM LASER" YELLOW
  STATE "FIRE LASER" "FIRE LASER" RED
```

READ_CONVERSION

The READ_CONVERSION keyword applies a conversion to the current telemetry item. This conversion is implemented in a custom Ruby file which should be located in the target's lib folder and required by the target's target.txt file. See the documentation in . The class must require 'cosmos/conversions/conversion' and inherit from Conversion. It must implement the initialize method if it takes extra parameters and must always implement the call method. This conversion factor is applied to the raw value in the telemetry packet before it is displayed to the user. The user still has the ability to see the raw unconverted value in a details dialog.

PARAMETER	DESCRIPTION	REQUIRED
Class file name	The file name which contains the Ruby class. The file name must be named after the class such that the class is a CamelCase version of the underscored file name. For example: 'the_great_conversion.rb' should contain 'class TheGreatConversion'.	Yes
Param X	Parameter #x. Additional parameter values for the conversion which are passed to the class constructor.	No

Example Usage:

```
READ_CONVERSION the_great_conversion.rb 1000
```

the_great_conversion.rb:

```
require 'cosmos/conversions/conversion'
module Cosmos
  class TheGreatConversion &lt; Conversion
    def initialize(multiplier)
      super()
      @multiplier = multiplier
    end
    def call(value, packet, buffer)
      return value * multiplier
    end
  end
end
```

POLY_READ_CONVERSION

The POLY_READ_CONVERSION keyword adds a polynomial conversion factor to the current telemetry item. This conversion factor is applied to the raw value in the telemetry packet before it is displayed to the user. The user still has the ability to see the raw unconverted value in a details dialog.

PARAMETER	DESCRIPTION	REQUIRED
C0	Coefficient #0	Yes
Cx	Coefficient #x. Additional coefficient values for the conversion. Any order polynomial conversion may be used so the value of 'x' will vary with the order of the polynomial. Note that larger order polynomials take longer to process than shorter order polynomials, but are sometimes more accurate.	No

Example Usage:

```
POLY_READ_CONVERSION 10 0.5 0.25
```

SEG_POLY_READ_CONVERSION

The SEG_POLY_READ_CONVERSION keyword adds a segmented polynomial conversion factor to the current telemetry item. This conversion factor is applied to the raw value in the telemetry packet before it is displayed to the user. The user still has the ability to see the raw unconverted value in a details dialog.

PARAMETER	DESCRIPTION	REQUIRED
Lower Bound	Defines the lower bound of the range of values that this segmented polynomial applies to. Is ignored for the segment with the smallest lower bound.	Yes
C0	Coefficient #0	Yes
Cx	Coefficient #x. Additional coefficient values for the conversion. Any order polynomial conversion may be used so the value of 'x' will vary with the order of the polynomial. Note that larger order polynomials take longer to process than shorter order polynomials, but are sometimes more accurate.	No

Example Usage:

```
SEG_POLY_READ_CONVERSION 0 10 0.5 0.25 # Apply the conversion to all values < 50
SEG_POLY_READ_CONVERSION 50 11 0.5 0.275 # Apply the conversion to all values >= 50 and < 100
SEG_POLY_READ_CONVERSION 100 12 0.5 0.3 # Apply the conversion to all values >= 100
```

GENERIC_READ_CONVERSION_START and GENERIC_READ_CONVERSION_END

**NOTE: Generic conversions are not a good long term solution. Consider creating a conversion class and using READ_CONVERSION instead. READ_CONVERSION is easier to debug and higher performance. **

The GENERIC_READ_CONVERSION_START keyword adds a generic conversion function to the current telemetry item. This conversion factor is applied to the raw value in the telemetry packet before it is displayed to the user. The user still has the ability to see the raw unconverted value in a details dialog. The conversion is specified as ruby code that receives two implied parameters: 'value' which is the raw value being read, and 'myself' which is a reference to the telemetry packet class. The last line of ruby code given should return the converted value. The GENERIC_READ_CONVERSION_END keyword specifies that all lines of ruby code for the conversion have been given.

Example Usage:

```
APPEND_ITEM ITEM1 32 UINT
  GENERIC_READ_CONVERSION_START
    value * 1.5  # Convert the value by a scale factor
  GENERIC_READ_CONVERSION_END
```

You can also create a conversion that depends on another parameter For example:

```
TELEMETRY INST HEALTH_STATUS BIG_ENDIAN "Health and status"
  ..
  APPEND_ITEM ZONE 8 UINT "Heater zone"
    STATE ONE 1
    STATE TWO 2
  ITEM ZONE_CONV 0 0 DERIVED
    GENERIC_READ_CONVERSION_START
      result = myself.read('ZONE', :RAW) # Access the raw zone value
      return result * 1.5
    GENERIC_READ_CONVERSION_END
```

LIMITS

The LIMITS keywords defines a set of limits for a telemetry item. If limits are violated a message is printed in the Command and Telemetry Server to indicate an item went out of limits. Other tools also use this information to update displays with different colored telemetry items or other useful information. The concept of "limits sets" is defined to allow for different limits values in different environments. For example, you might want tighter or looser limits on telemetry if your environment changes such as during thermal vacuum testing.

PARAMETER	DESCRIPTION	REQUIRED
Limits Set	Name of the limits set. If you have no unique limits sets use the keyword DEFAULT.	Yes
Persistence	Number of consecutive times the telemetry item must be within a different limits range before changing limits state.	Yes
ENABLED or DISABLED	Whether limits monitoring for this telemetry item is initially enabled or disabled.	Yes
Red Low Limit	If the telemetry value is less than or equal to this value a Red Low condition will be detected.	Yes
Yellow Low Limit	If the telemetry value is less than or equal to this value, but greater than the Red Low Limit, a Yellow Low condition will be detected.	Yes
Yellow High Limit	If the telemetry value is greater than or equal to this value, but less than the Red High Limit, a Yellow High condition will be detected.	Yes
Red High Limit	If the telemetry value is greater than or equal to this value a Red High condition will be detected.	Yes
Green Low Limit	Setting the Green Low and Green High limits defines an "operational limit" which is colored blue by COSMOS. This allows for a distinct desired operational range which is narrower than the green safety limit. If the telemetry value is greater than or equal to this value, but less than the Green High Limit, a Blue operational condition	No

	will be detected.	
Green High Limit	See above. If the telemetry value is less than or equal to this value, but greater than the Green Low Limit, a Blue operational condition will be detected.	No

Example Usage:

```
LIMITS DEFAULT 3 ENABLED -80.0 -70.0 60.0 80.0 -20.0 20.0
LIMITS TVAC 3 ENABLED -80.0 -30.0 30.0 80.0
```

LIMITS_RESPONSE

The LIMITS_RESPONSE keyword defines a response class that will be called when the limits state of the current item changes.

PARAMETER	DESCRIPTION	REQUIRED
Response Class Filename	Name of the Ruby file which implements the limits response. This file should be in the config/TARGET/lib directory so it can be found by COSMOS.	Yes
Response Specific Options	Variable length number of options that will be passed to the class constructor.	Response Specific

Example Usage:

```
LIMITS_RESPONSE example_limits_response.rb 10
```

META

The META keyword stores metadata for the current item that can be used by custom tools for various purposes (For example to store additional information needed to generate FSW header files).

PARAMETER	DESCRIPTION	REQUIRED
Meta Name	Name of the metadata to store	Yes
Meta Values	One or more values to be stored for this Meta Name	No

Example Usage:

```
META TEST "This item is for test purposes only"
```

Example File

Example File: <COSMOSPATH>/config/MY_TARGET/cmd_tlm/tlm.txt

```
TELEMETRY MY_TARGET HS BIG_ENDIAN "Health and Status for My Target"
  ITEM CCSDSVER 0 3 UINT "CCSDS PACKET VERSION NUMBER (SEE CCSDS 133.0-B-1)"
  ITEM CCSDSTYPE 3 1 UINT "CCSDS PACKET TYPE (COMMAND OR TELEMETRY)"
    STATE TLM 0
    STATE CMD 1
  ITEM CCSDSSHF 4 1 UINT "CCSDS SECONDARY HEADER FLAG"
    STATE FALSE 0
    STATE TRUE 1
  ID_ITEM CCSDSAPID 5 11 UINT 102 "CCSDS APPLICATION PROCESS ID"
  ITEM CCSDSSEQFLAGS 16 2 UINT "CCSDS SEQUENCE FLAGS"
    STATE FIRST 0
    STATE CONT 1
    STATE LAST 2
    STATE NOGROUP 3
  ITEM CCSDSSEQCNT 18 14 UINT "CCSDS PACKET SEQUENCE COUNT"
  ITEM CCSDSLENGTH 32 16 UINT "CCSDS PACKET DATA LENGTH"
  ITEM CCSDSDAY 48 16 UINT "DAYS SINCE EPOCH (JANUARY 1ST, 1958, MIDNIGHT)"
  ITEM CCSDSMSOD 64 32 UINT "MILLISECONDS OF DAY (0 - 86399999)"
  ITEM CCSDSUSOMS 96 16 UINT "MICROSECONDS OF MILLISECOND (0-999)"
  ITEM ANGLEDEG 112 16 INT "Instrument Angle in Degrees"
    POLY_READ_CONVERSION 0 57.295
  ITEM MODE 128 8 UINT "Instrument Mode"
    STATE NORMAL 0 GREEN
    STATE DIAG 1 YELLOW
  MACRO_APPEND_START 1 5
    APPEND_ITEM SETTING 16 UINT "SETTING #x"
  MACRO_APPEND_END
  ITEM TIMESECONDS 0 0 FLOAT "DERIVED TIME SINCE EPOCH IN SECONDS"
    GENERIC_READ_CONVERSION_START
      ((myself.ccsdsday.to_f * 86400.0) + (myself.ccsdsmsod.to_f / 1000.0) + (myself.ccsdsusoms.to_f / 1000000.0))
    GENERIC_READ_CONVERSION_END
  ITEM TIMEFORMATTED 0 0 STRING "DERIVED TIME SINCE EPOCH AS A FORMATTED STRING"
    GENERIC_READ_CONVERSION_START
      require 'time_util'
      time = TimeUtil.cds_to_mdy(myself.ccsdsday, myself.ccsdsmsod, myself.ccsdsusoms)
      sprintf('%04u/%02u/%02u %02u:%02u:%02u.%06u', time[0], time[1], time[2], time[3], time[4], time[5], time[6])
    GENERIC_READ_CONVERSION_END
```

BACK NEXT

Interface Configuration

Improve this page

Table of Contents

Interface classes provide the code that COSMOS uses to receive real-time telemetry from targets and to send commands to targets. The interface that a target uses could be anything (TCP/IP, serial, GPIB, Firewire, etc.), therefore it is important that this is a customize-able portion of any reusable Command and Telemetry System. Fortunately the most common form of interfaces are over TCP/IP sockets, and COSMOS provides interface solutions for these. This guide will discuss how to use these interface classes, and how to create your own.

Interfaces have the following methods that must be implemented:

1. **connect** - Open the socket or port or somehow establish the connection to the target
2. **connected?** - Return true or false depending on the connection state
3. **disconnect** - Close the socket or port of somehow disconnect from the target
4. **read** - Return the next packet of data from the target
5. **write** - Send a packet of data to the target
6. **write_raw** - Send a raw binary string of data to the target
7. **read_allowed?** - Whether reading from the target over the interface is allowed
8. **write_allowed?** - Whether writing a packet to the target over the interface is allowed
9. **write_raw_allowed?** - Whether writing raw data to the target over the interface is allowed

Provided Interfaces

Cosmos provides the following interfaces for use: TCPIP Client, TCPIP Server, UDP, Serial, Command Telemetry Server, and LINC.

TCPIP Client Interface

The TCPIP client interface connects to a TCPIP socket to send commands and receive telemetry. This interface is used

for targets which open a socket and wait for a connection. This is the most common type of interface.

PARAMETER	DESCRIPTION	REQUIRED
Host	Machine name to connect to	Yes
Write Port	Port to write commands to (can be the same as read port)	Yes
Read Port	Port to read telemetry from (can be the same as write port)	Yes
Write Timeout	Number of seconds to wait before aborting the write. Pass 'nil' to block on write.	Yes
Read Timeout	Number of seconds to wait before aborting the read. Pass 'nil' to block on read.	Yes
Stream Protocol Type	See Streams and Stream Protocols.	Yes
Stream Protocol Arguments	See Streams and Stream Protocols for the arguments each stream protocol takes.	Yes

cmd_tlm_server.txt Examples:

```
INTERFACE INTERFACE_NAME tcpip_client_interface.rb localhost 8080 8081 10.0 nil LENGTH 0 16 0 1 BIG_ENDIAN 4 0xBA5EBA11
INTERFACE INTERFACE_NAME tcpip_client_interface.rb localhost 8080 8080 10.0 nil BURST 4 0xDEADBEEF
INTERFACE INTERFACE_NAME tcpip_client_interface.rb localhost 8080 8080 10.0 nil FIXED 6 0 nil true
INTERFACE INTERFACE_NAME tcpip_client_interface.rb localhost 8080 8080 10.0 nil PREIDENTIFIED 0xCAFEBABE
INTERFACE INTERFACE_NAME tcpip_client_interface.rb localhost 8080 8080 10.0 10.0 TERMINATED 0x0D0A 0x0D0A true 0 0xF005BA1
```

TCPIP Server Interface

The TCPIP server interface creates a TCPIP server which listens for incoming connections and dynamically creates sockets which communicate with the target. This interface is used for targets which open a socket and try to connect to a server.

PARAMETER	DESCRIPTION	REQUIRED
Write Port	Port to write commands to (can be the same as read port)	Yes
Read Port	Port to read telemetry from (can be the same as write port)	Yes
Write Timeout	Number of seconds to wait before aborting the write. Pass 'nil' to block on write.	Yes
Read Timeout	Number of seconds to wait before aborting the read. Pass 'nil' to block on read.	Yes
Stream Protocol Type	See Streams and Stream Protocols.	Yes
Stream Protocol Arguments	See Streams and Stream Protocols for the arguments each stream protocol takes.	Yes

cmd_tlm_server.txt Examples:

```
INTERFACE INTERFACE_NAME tcpip_server_interface.rb 8080 8081 10.0 nil LENGTH 0 16 0 1 BIG_ENDIAN 4 0xBA5EBA11
INTERFACE INTERFACE_NAME tcpip_server_interface.rb 8080 8080 10.0 nil BURST 4 0xDEADBEEF
INTERFACE INTERFACE_NAME tcpip_server_interface.rb 8080 8080 10.0 nil FIXED 6 0 nil true
INTERFACE INTERFACE_NAME tcpip_server_interface.rb 8080 8080 10.0 nil PREIDENTIFIED 0xCAFEBABE
INTERFACE INTERFACE_NAME tcpip_server_interface.rb 8080 8080 10.0 10.0 TERMINATED 0x0D0A 0x0D0A true 0 0xF005BA11
```

UDP Interface

The UDP interface uses UDP packets to send and receive telemetry from the target. It can not use any stream protocols.

PARAMETER	DESCRIPTION	REQUIRED
Host	Host name or IP address of the machine to send and receive data with	Yes
Write Dest Port	Port on the remote machine to send commands to	Yes
Read Port	Port on the remote machine to read telemetry from	Yes
Write Source Port	Port on the local machine to send commands from. This can be 'nil' in which case the socket will not be bound to a outgoing port.	No
Interface Address	If the remote machine supports multicast the interface address is used to configure the outgoing multicast address. This can be 'nil' if unused.	No
TTL	Time to Live. The number of intermediate routers allowed before dropping the packet. The default on Windows platforms is 128.	No
Write Timeout	Number of seconds to wait before aborting the write. Pass 'nil' to block on write.	No
Read Timeout	Number of seconds to wait before aborting the read. Pass 'nil' to block on read.	No

cmd_tlm_server.txt Example:

```
INTERFACE INTERFACE_NAME udp_interface.rb localhost 8080 8081 8082 nil 128 10.0 nil
```

Serial Interface

The serial interface connects to a target over a serial port. COSMOS provides drivers for both Windows and POSIX drivers for UNIX based systems.

PARAMETER	DESCRIPTION	REQUIRED
Write Port	Name of the serial port to write, e.g. 'COM1' or '/dev/ttyS0'. Pass 'nil' to disable writing.	Yes
Read Port	Name of the serial port to read, e.g. 'COM1' or '/dev/ttyS0'. Pass 'nil' to disable reading.	Yes
Baud Rate	Baud rate to read and write	Yes
Parity	Serial port parity. Must be 'NONE', 'EVEN', or 'ODD'.	Yes
Stop Bits	Number of stop bits, e.g. 1.	Yes
Write Timeout	Number of seconds to wait before aborting the write. Pass 'nil' to block on write.	Yes
Read Timeout	Number of seconds to wait before aborting the read. Pass 'nil' to block on read.	Yes
Stream Protocol Type	See Streams and Stream Protocols.	Yes
Stream Protocol Arguments	See Streams and Stream Protocols for the arguments each stream protocol takes.	Yes

cmd_tlm_server.txt Examples:

```
INTERFACE INTERFACE_NAME serial_interface.rb COM1 COM1 9600 NONE 1 10.0 nil LENGTH 0 16 0 1 BIG_ENDIAN 4 0xBA5EBA11
INTERFACE INTERFACE_NAME serial_interface.rb /dev/ttyS1 /dev/ttyS1 38400 ODD 1 10.0 nil BURST 4 0xDEADBEEF
INTERFACE INTERFACE_NAME serial_interface.rb COM2 COM2 19200 EVEN 1 10.0 nil FIXED 6 0 nil true
INTERFACE INTERFACE_NAME serial_interface.rb /dev/ttyS0 /dev/ttyS0 57600 NONE 1 10.0 nil PREIDENTIFIED 0xCAFEBABE
INTERFACE INTERFACE_NAME serial_interface.rb COM4 COM4 115200 NONE 1 10.0 10.0 TERMINATED 0x0D0A 0x0D0A true 0 0xF005B
```

CmdTlmServer Interface

The CmdTlmServer interface provides a connection to the COSMOS Command and Telemetry Server. This allows scripts and other COSMOS tools to send commands to the CmdTlmServer to enable and disable logging. It also allows scripts and other tools to receive a COSMOS version information packet and a limits change packet which is sent when any telemetry items change limits states. The CmdTlmServer interface can be used by any COSMOS configuration.

cmd_tlm_server.txt Example:

```
INTERFACE COSMOSINT cmd_tlm_server_interface.rb
```

LINC Interface

The LINC interface uses a single TCPIP socket to talk to a Ball Aerospace LINC Labview target.

PARAMETER	DESCRIPTION	REQUIRED
Host	Machine name to connect to	Yes
Port	Port to write commands to and read telemetry from	Yes
Handshake Enabled	Enable command handshaking where commands block until the corresponding handshake message is received. The default is true.	No
Response Timeout	Number of seconds to wait for a handshaking response. The default is 5 seconds.	No
Read Timeout	Number of seconds to wait before aborting the write. Pass 'nil' to block on read. The default is nil.	No
Write Timeout	Number of seconds to wait before aborting the read. Pass 'nil' to block on write. The default is 5 seconds.	No
Length Bit Offset	The bit offset of the length field. Every packet using this interface must have the same structure such that the length field is the same size at the same location. The default is 0.	No
Length Bit Size	The size in bits of the length field. The default is 16.	No
Length Value Offset	The offset to apply to the length field value. For example if the length field indicates packet length minus one, this value should be one. The default is 4.	No
Fieldname GUID	Fieldname of the GUID field. The default is 'HDR_GUID'	No
Length Endianness	The endianness of the length field. Must be either 'BIG_ENDIAN' or 'LITTLE_ENDIAN'. The default is 'BIG_ENDIAN'.	No
Fieldname Cmd Length	Fieldname of the length field. The default is 'HDR_LENGTH'	No

cmd_tlm_server.txt Examples:

```
INTERFACE INTERFACE_NAME linc_interface.rb localhost 8080
INTERFACE INTERFACE_NAME linc_client_interface.rb localhost 8080 true 5 nil 5 0 16 4 HDR_GUID BIG_ENDIAN HDR_LENGTH
```

Streams and Stream Protocols

Streams are simplified interfaces that only implement the read, read_nonblock, write, connected? and disconnect methods. They are basically just data sinks and sources which are further manipulated by stream protocols. COSMOS provides the following streams: SerialStream, TcpipClientStream, and TcpipSocketStream. As you might guess, the SerialInterface, TcpipClientInterface, TcpipServerInterface directly use the SerialStream, TcpipClientStream, and TcpipSocketStream respectively. In addition, these interfaces require a StreamProtocol to process the data on the stream.

StreamProtocols process a stream of data on behalf of an interface. Once they are connected to their streams, they will

continuously read from the stream to amass a buffer of raw data which is then processed according to the protocol type. COSMOS provides the following stream protocols: Burst, Fixed, Length, Terminated, Preidentified, and Templated.

Burst Stream Protocol

The Burst Stream Protocol simply reads as much data as it can from the stream before returning the data as a COSMOS Packet. This Protocol relies on regular bursts of data delimited by time and thus is not very robust. However it can utilize a sync pattern which does allow it to re-sync from the stream if necessary.

PARAMETER	DESCRIPTION	REQUIRED
Discard Leading Bytes	The number of bytes to discard from the binary data after reading from the stream. Note that this applies to bytes starting with the sync pattern if the sync pattern is being used. The default is 0 which means to not discard any bytes.	No
Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw stream. This pattern represents a packet delimiter and all data found including the sync pattern will be returned. The default is 'nil' which means no sync pattern is used.	No
Fill Sync Pattern	Whether or not to fill in the sync pattern on outgoing packets. Defaults to false.	No

Fixed Stream Protocol

The Fixed Stream Protocol reads a preset minimum amount of data from the stream which is necessary to properly identify all the defined packets using the interface. It then identifies the packet and proceeds to read as much data from the stream as necessary to create the packet which it then returns. This stream relies on all the packets on the interface being fixed in length. For example, all the packets using the interface are a fixed size and contain a simple header with a 32 bit sync pattern followed by a 16 bit ID. The Fixed Stream Protocol would elegantly handle this case with a minimum read size of 6 bytes.

PARAMETER	DESCRIPTION	REQUIRED
Minimum ID Size	The minimum amount of bytes needed to identify a packet. All the packet definitions must declare their ID_ITEM(s) within this given amount of bytes.	Yes
Discard Leading Bytes	The number of bytes to discard from the binary data after reading from the stream. Note that this applies to bytes starting with the sync pattern if the sync pattern is being used. The default is 0 which means to not discard any bytes.	No
Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw stream. This pattern represents a packet delimiter and all data found including the sync pattern will be returned. The default is 'nil' which means no sync pattern is used.	No
Telemetry Stream	Whether the stream is returning telemetry. The default is true which means this is a telemetry stream. Pass false to declare a command stream.	No
Fill Sync Pattern	Whether or not to fill in the sync pattern on outgoing packets. Defaults to false.	No

Length Stream Protocol

The Length Stream Protocol depends on a length field at a fixed location in the defined packets using the interface. It then reads enough data to grab the length field, decodes it, and reads the remaining length of the packet. For example, all the packets using the interface contain a CCSDS header with a length field. The Length Stream Protocol can be set up to handle the length field and even the "length - 1" offset the CCSDS header uses.

PARAMETER	DESCRIPTION	REQUIRED
Length Bit Offset	The bit offset of the length field. Every packet using this interface must have the same structure such that the length field is the same size at the same location. The default is 0. Be sure to account for the length of the Sync Pattern in this value (if present).	No
Length Bit Size	The size in bits of the length field. The default is 16.	No

Length Value Offset	The offset to apply to the length field value. For example if the length field indicates packet length minus one, this value should be one. The default is 0. Be sure to account for the length of the Sync Pattern in this value (if present).	No
Bytes per Count	The number of bytes per each length field 'count'. This is used if the units of the length field is something other than bytes, for example if the length field count is in words. The default is 1.	No
Length Endianness	The endianness of the length field. Must be either 'BIG_ENDIAN' or 'LITTLE_ENDIAN'. The default is 'BIG_ENDIAN'.	No
Discard Leading Bytes	The number of bytes to discard from the binary data after reading from the stream. Note that this applies to bytes including the sync pattern if the sync pattern is being used. The default is 0 which means to not discard any bytes. Discarding is one of the very last steps so any size and offsets above need to account for all the data before discarding.	No
Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw stream. This pattern represents a packet delimiter and all data found including the sync pattern will be returned. The default is 'nil' which means no sync pattern is used.	No
Max Length	The maximum allowed value in the length field. The default is nil which means there is no maximum length.	No
Fill Length and Sync Pattern	Setting this flag to true causes the length field and sync pattern (if present) to be filled automatically on outgoing packets. Defaults to false.	No

Terminated Stream Protocol

The Terminated Stream Protocol delineates packets using termination characters found at the end of every packet. It continuously reads data from the stream until the termination characters are found at which point it returns the packet data. For example, all the packets using the interface are followed by 0xABCD. This data can either be a part of each packet that is kept or something which is known only by the Terminated Stream Protocol and simply thrown away.

PARAMETER	DESCRIPTION	REQUIRED
Write Termination Characters	The data to write to the stream after writing a command packet. Given as a hex string such as 0xABCD. The default is the empty string '' which means to write no termination characters.	No
Read Termination Characters	The characters at the end of the stream which delineate the end of a telemetry packet. Given as a hex string such as 0xABCD. The default is the empty string '' which won't work.	No
Strip Read Termination	Whether to remove the read termination characters from the stream before returning the telemetry packet. The default is true.	No
Discard Leading Bytes	The number of bytes to discard from the binary data after reading from the stream. Note that this applies to bytes including the sync pattern if the sync pattern is being used. The default is 0 which means to not discard any bytes.	No
Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw stream. This pattern represents a packet delimiter and all data found including the sync pattern will be returned. The default is 'nil' which means no sync pattern is used.	No
Fill Sync Pattern	Whether or not to fill in the sync pattern on outgoing packets. Defaults to false.	No

Preidentified Stream Protocol

The Preidentified Stream Protocol is used internally by the COSMOS Command and Telemetry Server only and delineates packets using a custom COSMOS header. This stream Protocol is configured by default on port 7779 and is created by the Command and Telemetry Server to allow tools to connect and receive the entire packet stream. The Telemetry Grapher uses this port to receive all the packets following through the Command and Telemetry Server in case any need to be graphed.

PARAMETER	DESCRIPTION	REQUIRED

Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw stream. This pattern represents a packet delimiter and all data found AFTER the sync pattern will be returned. The sync pattern itself is discarded. The default is 'nil' which means no sync pattern is used.	No
Max Length	The maximum allowed value in the length field. The default is nil which means there is no maximum length.	No

Templated Stream Protocol

The Templated Stream Protocol works much like the Terminated Stream Protocol except it designed for text-based command and response type interfaces. It delineates packets in the same way as the terminated stream protocol except each packet is referred to as a line (because each usually contains a line of text). For outgoing packets a CMD_TEMPLATE field is expected to exist in the packet. This field contains a template string with items to be filled in deliniated within HTML tag style brackets "". The Templated Stream Protocol will read the named items from within the packet fill in the CMD_TEMPLATE. This filled in string is then sent out rather than the originally passed in packet. Correspondingly, if a response is expected the outgoing packet should include a RSP_TEMPLATE and RSP_PACKET field. The RSP_TEMPLATE is used to extract data from the response string and build a corresponding RSP_PACKET. See the TEMPLATED target within the COSMOS Demo configuration for an example of usage.

PARAMETER	DESCRIPTION	REQUIRED
Write Termination Characters	The data to write to the stream after writing a command packet. Given as a hex string such as 0xABCD. The default is the empty string '' which means to write no termination characters.	No
Read Termination Characters	The characters at the end of the stream which delineate the end of a telemetry packet. Given as a hex string such as 0xABCD. The default is the empty string '' which won't work.	No
Ignore Lines	Number of response lines to ignore (completely drop). Defaults to 0.	No
Initial Read Delay	An initial delay after connecting after which the stream will be read till empty and data dropped. Useful for discarding connect headers and initial prompts. Defaults to nil which means no initial read.	No
Response Lines	The number of lines that make up expected responses. Defaults to 1.	No
Strip Read Termination	Whether to remove the read termination characters from the stream before returning the telemetry packet. The default is true.	No
Discard Leading Bytes	The number of bytes to discard from the binary data after reading from the stream. Note that this applies to bytes including the sync pattern if the sync pattern is being used. The default is 0 which means to not discard any bytes.	No
Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw stream. This pattern represents a packet delimiter and all data found including the sync pattern will be returned. The default is 'nil' which means no sync pattern is used.	No
Fill Sync Pattern	Whether or not to fill in the sync pattern on outgoing packets. Defaults to false.	No

Tool Configuration

Improve this page

Table of Contents

Keywords:

- REQUIRE_UTILITY
- RESULTS_WRITER
- ALLOW_DEBUG
- PAUSE_ON_ERROR
- CONTINUE_TEST_CASE_AFTER_ERROR
- ABORT_TESTING_AFTER_ERROR
- MANUAL
- LOOP_TESTING
- BREAK_LOOP_AFTER_ERROR
- IGNORE_TEST
- IGNORE_TEST_SUITE
- CREATE_DATA_PACKAGE
- AUTO_CYCLE_LOGS
- COLLECT_META_DATA
- Script Runner Configuration Keywords
- Example File

Test Runner Usage Notes

Table Manager Configuration

Keywords:

- TABLEFILE
- TABLE
- PARAMETER
- STATE
- DEFAULT
- POLY_WRITE_CONVERSION
- POLY_READ_CONVERSION
- GENERIC_WRITE_CONVERSION_START / GENERIC_WRITE_CONVERSION_END
- GENERIC_READ_CONVERSION_START / GENERIC_READ_CONVERSION_END
- CONSTRAINT_START / CONSTRAINT_END

Example File

Please see Telemetry Screen Configuration for instructions on configuring Telemetry Viewer.

Launcher Configuration

Launcher configuration files define the icons and buttons presented in the Launcher and define how programs are launched. These files are expected to be placed in the config/tools/launcher directory and have a .txt extension. The default configuration file is named launcher.txt.

Keywords:

TITLE

The TITLE keyword changes the title of the COSMOS Launcher.

PARAMETER	DESCRIPTION	REQUIRED
Title	Launcher title. Defaults to "COSMOS Launcher" without this keyword.	Yes

Example Usage:

```
TITLE 'Program Launcher'
```

TOOL_FONT

The TOOL_FONT keyword sets the font used for tool buttons. It should only be used once as the last encountered setting will apply to all button labels. The default font if this keyword is not used is 12px Arial.

PARAMETER	DESCRIPTION	REQUIRED
Font Family	The font family to use. Valid choices are "Arial", "Calibri", "Cambria", "Candara", "Castellar", "Centaur", "Century", "Chiller", "Consolas", "Constantia", "Courier", "Courier New", "Dotum", "Elephant", "Euphemia", "Fixedsys", "Georgia", "Impact", "Lucida", "Papyrus", "Rockwell", "Rod", "System", "Tahoma", "Terminal", "Times New Roman", "Verdana", "Wide Latin".	Yes
Font Size	The size of the font in standard points.	Yes

Example Usage:

```
TOOL_FONT Courier 10
```

LABEL_FONT

The LABEL_FONT keyword sets the font used for labels. It should only be used once as the last encountered setting will apply to all labels. The default font if this keyword is not used is 16px Tahoma.

PARAMETER	DESCRIPTION	REQUIRED
Font Family	The font family to use. Valid choices are "Arial", "Calibri", "Cambria", "Candara", "Castellar", "Centaur", "Century", "Chiller", "Consolas", "Constantia", "Courier", "Courier New", "Dotum", "Elephant", "Euphemia", "Fixedsys", "Georgia", "Impact", "Lucida", "Papyrus", "Rockwell", "Rod", "System", "Tahoma", "Terminal", "Times New Roman", "Verdana", "Wide Latin".	Yes
Font Size	The size of the font in standard points.	Yes

```
Example Usage:
LABEL_FONT Arial 20
```

LABEL

The LABEL keyword creates a label of text in the current font style.

PARAMETER	DESCRIPTION	REQUIRED
Text	The text of the label.	Yes

Example Usage:

```
LABEL Utilities
```

DIVIDER

The DIVIDER keyword creates a horizontal line between tools. It takes no parameters and is purely for decoration.

NUM_COLUMNS

The NUM_COLUMNS keyword specifies how launcher buttons should be created per row in the GUI. The default is 4.

PARAMETER	DESCRIPTION	REQUIRED
Columns	The number of launcher buttons per row before Launcher automatically creates a new row of buttons.	Yes

TOOL

The TOOL keyword specifies one tool that can be launched. The syntax varies if it is being used standalone or for a multi-tool.

Syntax when standalone:

PARAMETER	DESCRIPTION	REQUIRED
Button Text	Label that is put on the button that launches the tool	Yes
Shell Command	Command that is executed to launch the tool. (The same thing you would type at a command terminal). Note that you can include tool parameters here which will be applied when the tool starts.	Yes
Icon Filename	Filename of a an icon located in the data directory. Passing 'nil' or an empty string '' will result in Launcher using the default COSMOS icon.	No
Tool Parameters	Tool parameters as you would type on the command line. Specifying parameters here rather than in "Shell Command" will cause a dialog box to appear which allows the user to edit parameters if desired. Expected to be in parameter name/parameter value pairs. i.e. —config filename.txt. NOTE: The full configuration option name must be used rather than the short name. NOTE: These parameters will override any parameters specified in the Shell Command. Also, multiple options can be specified by seperating options with a pipe character |	No

Example Usage:

```
TOOL "Command and Telemetry Server" "RUBYW tools/CmdTlmServer" cts.png --config cmd_tlm_server.txt
TOOL "Script Runner" "RUBYW tools/ScriptRunner" nil --width 600 --height 800
```

Syntax when used within the MULTITOOL keywords:

PARAMETER	DESCRIPTION	REQUIRED
Shell Command	Command that is executed to launch the tool. (The same thing you would type at a command terminal)	Yes

Example Usage: See MULTITOOL_START

MULTITOOL_START

The MULTITOOL_START keyword starts the creation of a single icon/button that will launch multiple tools.

PARAMETER	DESCRIPTION	REQUIRED
Button Text	Label that is put on the button that launches the tools	Yes
Icon Filename	Filename of a an icon located in the data directory. Passing 'nil' or an empty string '' will result in Launcher using the default COSMOS icon.	No

```
Example Usage:
MULTITOOL_START "COSMOS"
  TOOL "RUBYW tools/CmdTlmServer -x 827 -y 2 -w 756 -t 475 -c cmd_tlm_server.txt"
  DELAY 5
  TOOL "RUBYW tools/TlmViewer -x 827 -y 517 -w 424 -t 111"
MULTITOOL_END
```

MULTITOOL_END

The MULTITOOL_END keyword ends the creation of a multi-tool button.

Example Usage: See MULTITOOL_START

DELAY

The DELAY keyword inserts a delay between launching multiple tools. It is only valid within the MULTITOOL keywords.

PARAMETER	DESCRIPTION	REQUIRED
Delay	Time to delay in seconds	Yes

Example Usage: See MULTITOOL_START

Example File

Example File: <Cosmos::USERPATH>/config/tools/launcher/launcher.txt

```
TITLE "Demo Launcher"
FONT tahoma 12
NUM_COLUMNS 4
MULTITOOL_START "COSMOS" NULL
 TOOL "RUBYW tools/CmdTlmServer -x 827 -y 2 -w 756 -t 475 -c cmd_tlm_server.txt"
 DELAY 5
 TOOL "RUBYW tools/TlmViewer -x 827 -y 517 -w 424 -t 111"
 TOOL "RUBYW tools/PacketViewer -x 827 -y 669 -w 422 -t 450"
 TOOL "RUBYW tools/ScriptRunner -x 4 -y 2 -w 805 -t 545"
 TOOL "RUBYW tools/CmdSender -x 4 -y 586 -w 805 -t 533"
MULTITOOL_END
TOOL "Command and Telemetry Server" "RUBYW tools/CmdTlmServer" "cts.png" --config cmd_tlm_server.txt
TOOL "Limits Monitor" "RUBYW tools/LimitsMonitor" "limits_monitor.png"
DIVIDER
LABEL "Commanding and Scripting"
TOOL "Command Sender" "RUBYW tools/CmdSender" "cmd_sender.png"
TOOL "Script Runner" "RUBYW tools/ScriptRunner" "script_runner.png"
TOOL "Test Runner" "RUBYW tools/TestRunner" "test_runner.png"
DIVIDER
LABEL Telemetry
TOOL "Packet Viewer" "RUBYW tools/PacketViewer" "packet_viewer.png"
TOOL "Telemetry Viewer" "RUBYW tools/TlmViewer" "tlm_viewer.png"
TOOL "Telemetry Grapher" "RUBYW tools/TlmGrapher" "tlm_grapher.png"
TOOL "Data Viewer" "RUBYW tools/DataViewer" "data_viewer.png"
DIVIDER
LABEL Utilities
TOOL "Telemetry Extractor" "RUBYW tools/TlmExtractor" "tlm_extractor.png"
TOOL "Command Extractor" "RUBYW tools/CmdExtractor" "cmd_extractor.png"
TOOL "Handbook Creator" "RUBYW -Ku tools/HandbookCreator" "handbook_creator.png"
TOOL "Table Manager" "RUBYW tools/TableManager" "table_manager.png"
```

Limits Monitor Configuration

Limits Monitor has the ability to ignore telemetry items and no longer monitor them for transition to the yellow and red state. This is useful for known telemetry item configuration problems. Note that while telemetry items may be ignored in Limits Monitor, their states are still being recorded by the Command and Telemetry Server.

Keywords:

IGNORE

The IGNORE keyword instructs Limits Monitor to ignore a particular telemetry item when reporting the overall system

limits state.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the telemetry target	Yes
Packet Name	Name of the telemetry packet	Yes
Item Name	Name of the telemetry item	Yes

Example Usage:

```
IGNORE INST HEALTH_STATUS TEMP1
IGNORE INST HEALTH_STATUS TEMP2
```

Script Runner Configuration

The Script Runner Configuration File affects both the Script Runner tool and the Test Runner tool. It defines where to look for procedures and other scripting settings. The configuration file is named script_runner.txt.

Keywords:

LINE_DELAY

The LINE_DELAY keyword specifies the amount of time in seconds before the next line of a script will be executed. This allows the user to easily watch as a script progresses.

PARAMETER	DESCRIPTION	REQUIRED
Delay	Delay in seconds before the next line is executed. A value of 0 means to execute the scripts as fast as possible.	Yes

Example Usage:

```
LINE_DELAY 1.0 # Delay for 1 second between lines
```

MONITOR_LIMITS

The MONITOR_LIMITS keyword specifies that Script Runner should log limits events while a script is running. These limit evens will be printed in the script runner log file. Note that this has no effect of the Command and Telemetry Server which always logs limits events.

PAUSE_ON_RED

The PAUSE_ON_RED keyword specifies that Script Runner should pause a running script if a red limit occurs

Example File

Example File: <Cosmos::USERPATH>/config/tools/script_runner/script_runner.txt

```
LINE_DELAY 0.1
MONITOR_LIMITS
PAUSE_ON_RED
```

Telemetry Extractor Configuration

Telemetry Extractor configuration files define what telemetry points that telemetry extractor should pull out of a log file. These files are expected to be placed in the tools/config/tlm_extractor directory and have a .txt extension. The default configuration file is named tlm_extractor.txt.

Keywords:

DELIMITER

The DELIMITER keyword specifies an alternative column delimiter over the default of a tab character.

PARAMETER	DESCRIPTION	REQUIRED
Delimiter	Character or string to use as a delimiter. For example ','.	Yes

Example Usage:

```
DELIMITER ','
```

FILL_DOWN

The FILL_DOWN keyword causes Telemetry Extractor it to insert a value into every row of the output. For example, if you have the following telemetry extractor configuration file:

```
ITEM INST HEALTH_STATUS PKTID
ITEM INST HEALTH_STATUS COLLECTS
ITEM INST ADCS POSX
ITEM INST ADCS POSX
```

Your normal output might look something like this:

TARGET	PACKET	PKTID	COLLECTS	POSX	POSX
INST	ADCS			-579296	-579296
INST	HEALTH_STATUS	1	0		
INST	ADCS			-578683	-578683

with FILL_DOWN enabled it would like this this:

TARGET	PACKET	PKTID	COLLECTS	POSX	POSX
INST	ADCS			-579296	-579296
INST	HEALTH_STATUS	1	0	-579296	-579296
INST	ADCS	1	0	-578683	-578683

Note that in the second INST ADCS packet the PKTID and COLLECTS values are "filled down" even though they were not present in that packet. This makes it easier to graph multiple values across packets in Excel.

DOWNSAMPLE_SECONDS

The DOWNSAMPLE_SECONDS keyward causes Telemetry Extractor to downsample data to only output a value every X seconds of time.

PARAMETER	DESCRIPTION	REQUIRED
Seconds	Number of seconds to skip between values output	Yes

Example Usage:

```
DOWNSAMPLE_SECONDS 5
```

MATLAB_HEADER

The MATLAB_HEADER keyword cause Telemetry Extractor to prepend the Matlab comment symbol of '%' to the header lines in the output file.

UNIQUE_ONLY

The UNIQUE_ONLY keyword causes Telemetry Extractor to only output a row if one of the extracted values has changed. This can be used to extract telemetry items over a large time period by only outputting those values where items have changed.

UNIQUE_IGNORE

The UNIQUE_IGNORE keyword is used in conjunction with UNIQUE_ONLY to control which items should be checked for changing values. This list of telemetry items (not target names or packet names) always includes the COSMOS metadata items named RECEIVED_TIMEFORMATTED and RECEIVED_SECONDS. This is because these items will always change from packet to packet which would cause them to ALWAYS be printed if UNIQUE_ONLY was used. To avoid this, but still include time stamps in the output, UNIQUE_IGNORE includes these items. If you have a similar telemetry item that you want to display in the output, but not be used to determine uniqueness, use this keyword.

PARAMETER	DESCRIPTION	REQUIRED
Item Name	Name of the item to exclude from the uniqueness criteria (see above). Note that all items with this name in all target packets are affected.	Yes

SHARE_COLUMNS

The SHARE_COLUMNS keyword causes Telemetry Extractor to put telemetry items with the same name into the same column in the output. For example if you have the following configuration file:

```
ITEM INST HEALTH_STATUS PKTID
ITEM INST ADCS PKTID
```

Your normal output would look something like this:

TARGET	PACKET	PKTID	PKTID
INST	ADCS		1
INST	HEALTH_STATUS	1	
INST	ADCS		1

Note how both telemetry packets got their own unique PKTID column. With SHARE_COLUMNS enabled you would get something like this:

TARGET	PACKET	PKTID
INST	ADCS	1
INST	HEALTH_STATUS	1
INST	ADCS	1

Note how both packets share the one PKTID column. This applies to all telemetry items with identical names.

DONT_OUTPUT_FILENAMES

The DONT_OUTPUT_FILENAMES keyword prevents Telemetry Extractor from outputing the list of input filenames at the top of each output file.

TEXT

The TEXT keyword allows you to place arbitrary text in the Telemetry Extractor output. It also allows you to dynamically create Excel formulas using a special syntax.

PARAMETER	DESCRIPTION	REQUIRED
Header	The column header text	Yes
Text	Text to put in the output file. The special character '%' will be translated to the current row of the output file. This is useful for Excel formulas which need a reference to a cell. Remember the first two columns are always the TARGET and PACKET and telemetry items start in column 'C' in Excel.	Yes

Example Usage:

```
TEXT "Calc" "=C%*D%" # Excel will calculate the C column times the D column
```

ITEM

The ITEM keyword specifies a telemetry item to extract.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the telemetry target	Yes
Packet Name	Name of the telemetry packet	Yes
Item Name	Name of the telemetry item	Yes
Item Type	RAW, FORMATTED, or WITH_UNITS (CONVERTED is implied if the parameter is omitted)	No

Example File

Example File: <Cosmos::USERPATH>/config/tools/tlm_extractor/tlm_extractor.txt

```
FILL_DOWN
MATLAB_HEADER
DELIMITER ","
SHARE_COLUMNS
DOWNSAMPLE_SECONDS 5
```

DONT_OUTPUT_FILENAMES

```
UNIQUE_ONLY
UNIQUE_IGNORE TEMP1
ITEM INST HEALTH_STATUS TIMEFORMATTED
ITEM INST HEALTH_STATUS TEMP1 RAW
ITEM INST HEALTH_STATUS TEMP2 FORMATTED
ITEM INST HEALTH_STATUS TEMP3 WITH_UNITS
ITEM INST HEALTH_STATUS TEMP4
TEXT "Calc" "=D%*G%" # Calculate TEMP1 (RAW) times TEMP4
```

Test Runner Configuration

Test Runner configuration files define what tests should be run with what options. These files are expected to be placed in the tools/config/test_runner directory and have a .txt extension. The default configuration file is named

test_runner.txt.

Keywords:

REQUIRE_UTILITY

The REQUIRE_UTILITY keyword specifies a test procedure to run. This procedure will be found automatically in the procedures directory or can be given by a path relative to the COSMOS install directory or by an absolute path.

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of the test file	Yes

Example Usage:

```
REQUIRE_UTILITY example_test # .rb is optional
REQUIRE_UTILITY ../../example_test # Relative path from the base of the COSMOS configuration
REQUIRE_UTILITY C:/procedures/example_test # Absolute path (not cross platform)
```

RESULTS_WRITER

The RESULTS_WRITER keyword allows you to specify a different Ruby file to interpret and print the Test Runner results. This file must define a class which implements the Cosmos::ResultsWriter API.

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of the Ruby file which implements a results writer	Yes
Class Parameters	Parameters to pass to the constructor of the results writer	Class specific

Example Usage:

```
RESULTS_WRITER my_results_writer.rb
```

ALLOW_DEBUG

Whether to allow the user to enable the debug line where the user can enter arbitrary statements.

PAUSE_ON_ERROR

The PAUSE_ON_ERROR keyword sets or clears the pause on error checkbox. If this is checked, Test Runner will pause if the test encounters an error. Otherwise the error will be logged but the script will continue.

PARAMETER	DESCRIPTION	REQUIRED
True or False	Whether to pause when the script encounters an error	Yes

Example Usage:

```
PAUSE_ON_ERROR TRUE # default
```

CONTINUE_TEST_CASE_AFTER_ERROR

The CONTINUE_TEST_CASE_AFTER_ERROR keyword sets or clears the continue test case after error checkbox. If this is checked, Test Runner will continue executing the current test case after encountering an error. Otherwise the test case will stop at the error and the next test case will execute.

PARAMETER	DESCRIPTION	REQUIRED
True or False	Whether to continue the test case when the script encounters an error	Yes

Example Usage:

```
CONTINUE_TEST_CASE_AFTER_ERROR TRUE # default
```

ABORT_TESTING_AFTER_ERROR

The ABORT_TESTING_AFTER_ERROR keyword sets or clears the abort testing after error checkbox. If this is checked, Test Runner will stop executing after the current test case completes (how it completes depends on CONTINUE_TEST_CASE_AFTER_ERROR). Otherwise the next test case will execute.

PARAMETER	DESCRIPTION	REQUIRED
True or False	Whether to continue to the next test case when the script encounters an error	Yes

Example Usage:

```
ABORT_TESTING_AFTER_ERROR FALSE # default
```

MANUAL

The MANUAL keyword sets the $manual global variable for all executing scripts. This variable can be checked during tests to allow for fully automated tests if it is not set, or for user input if it is set.

PARAMETER	DESCRIPTION	REQUIRED
True or False	Whether to set the $manual global to true	Yes

Example Usage:

```
MANUAL TRUE # default
```

LOOP_TESTING

The LOOP_TESTING keyword sets or clears the loop testing checkbox. If this is checked, Test Runner will continue to run whatever level of tests that were initially started. If either "Abort Testing after Error" or "Break Loop after Error" are checked, then the loop testing will stop if an error is encountered. The difference is that the "Abort Testing after Error" will stop testing immediately after the current test case completes. "Break Loop after Error" continues the current loop by executing the remaining suite or group before stopping. In the case of executing a single test case the options effectively do the same thing.

PARAMETER	DESCRIPTION	REQUIRED
True or False	Whether to loop the selected test level	Yes

Example Usage:

```
LOOP_TESTING FALSE # default
```

BREAK_LOOP_AFTER_ERROR

The BREAK_LOOP_AFTER_ERROR keyword sets or clears the break loop after error checkbox. If this is checked, Test Runner continues the current loop by executing the remaining suite or group before stopping.

PARAMETER	DESCRIPTION	REQUIRED
True or False	Whether to break the loop after encountering an error	Yes

Example Usage:

```
BREAK_LOOP_AFTER_ERROR FALSE # default
```

IGNORE_TEST

The IGNORE_TEST keyword ignores the given test class name when parsing the tests.

PARAMETER	DESCRIPTION	REQUIRED
Test Class Name	The test class to ignore when building the list of available tests	Yes

Example Usage:

```
IGNORE_TEST ExampleTest
```

IGNORE_TEST_SUITE

The IGNORE_TEST_SUITE keyword ignores the given test suite name when parsing the tests.

PARAMETER	DESCRIPTION	REQUIRED
Test Suite Name	The test suite to ignore when building the list of available test suites	Yes

Example Usage:

```
IGNORE_TEST_SUITE ExampleTestSuite
```

CREATE_DATA_PACKAGE

The CREATE_DATA_PACKAGE keyword creates a data package of every file created during the test.

Example Usage:

```
CREATE_DATA_PACKAGE
```

AUTO_CYCLE_LOGS

The AUTO_CYCLE_LOGS automatically starts a new server message log and cmd/tlm logs at the beginning and end of each test. Very useful when coupled with the CREATE_DATA_PACKAGE keyword.

Example Usage:

```
AUTO_CYCLE_LOGS
```

COLLECT_META_DATA

The COLLECT_META_DATA keyword tells TestRunner to prompt for Meta Data before starting tests.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Meta Data Target Name	Yes
Packet Name	Meta Data Packet Name	Yes

Example Usage:

```
COLLECT_META_DATA META DATA
```

Script Runner Configuration Keywords

Test Runner also responds to all the keywords in the Script Runner Configuration.

Example File

Example File: <Cosmos::USERPATH>/config/tools/test_runner/test_runner.txt

```
REQUIRE_UTILITY example_test
ALLOW_DEBUG
PAUSE_ON_ERROR TRUE
CONTINUE_TEST_CASE_AFTER_ERROR TRUE
ABORT_TESTING_AFTER_ERROR FALSE
MANUAL TRUE
LOOP_TESTING TRUE
BREAK_LOOP_AFTER_ERROR TRUE
IGNORE_TEST ExampleTest
IGNORE_TEST_SUITE ExampleTestSuite

CREATE_DATA_PACKAGE
COLLECT_META_DATA META DATA

LINE_DELAY 0
MONITOR_LIMITS
PAUSE_ON_RED
```

Test Runner Usage Notes

Arbitrary text can be written to the test report using the following syntax:

```
Cosmos::Test.puts "This test verifies requirement 2"
```

You can determine the currently executing suite/group/case using the following syntax:

```
Cosmos::Test.current_test_suite
Cosmos::Test.current_test_group
Cosmos::Test.current_test_case
```

Table Manager Configuration

Table definition files define the binary table format so the Table Manager tool knows how to create, load, and display the binary data file. Table definition files are expected to be placed in the config/tools/table_manager directory and have a .txt extension.

Keywords:

TABLEFILE

The TABLEFILE keyword specifies another file to open and process for table definitions

PARAMETER	DESCRIPTION	REQUIRED
File Name	Name of the file. The file will be looked for in the directory of the current definition file.	Yes

Example Usage:

```
TABLEFILE "MCConfigurationTable_def.txt"
```

TABLE

The TABLE keyword designates the start of a new table definition.

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the table in double quotes. The name will appear on the GUI tab.	Yes
Description	Description of this table in double quotes. The description is used in mouseover popups and status line information.	Yes
Dimension	Indicates the table is a ONE_DIMENSIONAL table which is a two column table consisting of unique rows, or a TWO_DIMENSIONAL table with multiple columns and identical rows with unique values.	Yes
Endianness	Whether to packet the table data as BIG_ENDIAN or LITTLE_ENDIAN.	Yes
Identifier	A unique numerical Table ID.	Yes

Example Usage:

```
TABLE "MC Configuration" "Memory Control Configuration Table" ONE_DIMENSIONAL BIG_ENDIAN 9
```

PARAMETER

The PARAMETER keyword defines a table parameter in the current table.

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of this parameter in double quotes. Must be unique within this table.	Yes
Description	Description of this parameter in double quotes. The description is used in mouseover popups and status line information.	Yes
Data Type	Data Type of this parameter. Possible types: INT = Integer, UINT = Unsigned Integer, FLOAT = IEEE Floating point data (32 or 64 bit), STRING = ASCII string, BLOCK = Binary block of data	Yes
Bit Size	Bit size of this parameter. Must be greater than 0.	Yes
Display Type	Display Type of this parameter. Possible types: DEC = Decimal, HEX = Hex, STATE = States (must be given later), CHECK = Checkbox, STRING = Must be used for STRING data types, NONE = Must be used for BLOCK types. Appending a -U to the display type makes the field uneditable.	Yes
Minimum Value	Minimum allowed value for this parameter. For STRING data types this indicates the minimum number of characters.	Yes
Maximum Value	Maximum allowed value for this parameter. For STRING data types this indicates the maximum number of characters.	Yes
Default Value	Default value for this parameter	Yes

Example Usage:

```
PARAMETER "Throttle" "Seconds to wait" FLOAT 32 DEC 0 0x0FFFFFFFF 2
PARAMETER "Scrubbing" "Memory Scrubbing" UINT 8 CHECK 0 1 1
PARAMETER "Pad" "Pad" UINT 16 HEX-U 0 0 0
```

STATE

The STATE keyword defines a key/value pair for the current table parameter (the one most recently defined). For example, you might define states for ON = 1 and OFF = 0. This allows the word ON to be used rather than the number 1 when setting the table parameter and allows for much greater clarity and less chance for user error.

PARAMETER	DESCRIPTION	REQUIRED

Key	The named state key. This can also be a string enclosed in double quotes.	Yes
Value	Value the key translates into	Yes

Example Usage:

```
PARAMETER "Scrubbing" "Memory Scrubbing" UINT 8 STATE 0 1 1
  STATE DISABLE 0
  STATE ENABLE 1
```

DEFAULT

The DEFAULT keyword defines the default values for a row in a TWO_DIMENSIONAL table. Therefore, the number of DEFAULT lines defines the number of rows in the table. If no values are given after the keyword, the default as defined in the PARAMETER(s) will apply.

PARAMETER	DESCRIPTION	REQUIRED
Value1	The default value for the first defined PARAMETER in the TWO_DIMENSIONAL table.	No
Value2	The default value for the second defined PARAMETER in the TWO_DIMENSIONAL table.	No
ValueN	The default value for the last defined PARAMETER in the TWO_DIMENSIONAL table.	No

Example Usage:

```
TABLE "TLM Monitoring" "Telemetry Monitoring Table" TWO_DIMENSIONAL BIG_ENDIAN 4
  PARAMETER "Threshold" "Telemetry item threshold at which point persistance is incremented" UINT 32 HEX 0 4294967295 0
  PARAMETER "Offset" "Offset into the telemetry packet to monitor" UINT 32 DEC 0 4294967295 0
  PARAMETER "Data Size" "Amount of data to monitor (bytes)" UINT 32 STATE 0 3 0
    STATE BITS 0
    STATE BYTE 1
    STATE WORD 2

DEFAULT          # Defaults of 0, 0, 0(BITS) will be used
DEFAULT 0x2      # Override Threshold default of 0
DEFAULT 0x3 30 WORD # Note the use of STATE names
```

POLY_WRITE_CONVERSION

The POLY_WRITE_CONVERSION keyword adds a polynomial conversion factor to the current table parameter (the one most recently defined). This conversion factor is applied to the value entered by the user before it is written into the binary table file. All parameters with a POLY_WRITE_CONVERSION must also have a POLY_READ_CONVERSION. This read conversion should be the inverse function of the write conversion or the value might be inadvertently changed every time it is loaded and saved.

PARAMETER	DESCRIPTION	REQUIRED
C0	Coefficient #0.	Yes
Cx	Coefficient #x. This is the final coefficient value for the conversion. Any order polynomial conversion may be used so the value of 'x' will vary with the order of the polynomial. Note that larger order polynomials take longer to process than shorter order polynomials, but are sometimes more accurate.	Yes

Example Usage:

```
POLY_WRITE_CONVERSION 1 2 # 2x + 1 when writing to the table binary
POLY_READ_CONVERSION -0.5 0.5 # 0.5x - 0.5 when reading from the table binary
```

POLY_READ_CONVERSION

The POLY_READ_CONVERSION keyword adds a polynomial conversion factor to the current table parameter (the one most recently defined). This conversion factor is applied to a table parameter read from the binary table file before it is displayed. Read conversions can be applied independently to read-only table parameters. If a table parameter is writable it must have both a read and write conversion.

PARAMETER	DESCRIPTION	REQUIRED
C0	Coefficient #0.	Yes
Cx	Coefficient #x. This is the final coefficient value for the conversion. Any order polynomial conversion may be used so the value of 'x' will vary with the order of the polynomial. Note that larger order polynomials take longer to process than shorter order polynomials, but are sometimes more accurate.	Yes

Example Usage:

```
POLY_WRITE_CONVERSION -0.5 0.25 # 0.25x - 0.5 when writing to the table binary
POLY_READ_CONVERSION 2 4 # 2x + 4 when reading from the table binary
```

GENERIC_WRITE_CONVERSION_START / GENERIC_WRITE_CONVERSION_END

GENERIC_READ_CONVERSION_START / GENERIC_READ_CONVERSION_END

The generic conversion keywords add generic conversion functions to the current table parameter (the one most recently defined). This conversion factor is applied to the value entered by the user before it is written into the binary table file (for WRITE) and after it is read from the binary (for READ). The conversion is specified as ruby code that receives two implied parameters: 'value' which is the raw table parameter, and 'myself' which is a reference to the TableManager class. The last line of ruby code given should return the converted value. The conversion END keywords specify that all lines of ruby code for the conversion have been given. If a generic write conversion is created a generic read conversion must also be created. This read conversion should be the inverse of the write conversion or the value might be inadvertently changed every time it is loaded and saved.

Example Usage:

```
GENERIC_WRITE_CONVERSION_START
  value / 2
GENERIC_WRITE_CONVERSION_END
GENERIC_READ_CONVERSION_START
  value * 2
GENERIC_READ_CONVERSION_END
```

CONSTRAINT_START / CONSTRAINT_END

The constraint keyword allows the user to modify the current parameter's allowable value range, default value, and/or conversion based on the value of another parameter.

Example Usage:

```
TABLE "Mechanism Control" "Description" ONE_DIMENSIONAL BIG_ENDIAN 1
  PARAMETER "Total Steps" "Total number of steps" UINT 16 HEX-U 0 2000 0
    CONSTRAINT_START
      if myself.get_table("Mechanism Control").get_packet_item("Other Param") == "OLD"
        value.range = 0..1.0\n")
        value.default = 1.0
      end
    CONSTRAINT_END
  PARAMETER "Other Param" "Yet another" INT 32 STATE -100 100 10
    STATE "OLD" 0
    STATE "NEW" 1
```

Example File

Example File: /config/table_manager/metable_def.txt

```
TABLE "Mechanism Control" "Mechanism Control Table" ONE_DIMENSIONAL BIG_ENDIAN 1
  PARAMETER "Total Steps" "Total number of steps" UINT 16 HEX-U 0 2000 0
    CONSTRAINT_START
      if myself.get_table("Mechanism Control").get_packet_item("Other Param") == "OLD"
        value.range = 0..1.0\n")
        value.default = 1.0
      end
    CONSTRAINT_END
    POLY_READ_CONVERSION 1 2
    POLY_WRITE_CONVERSION 2 3
  PARAMETER "My Param" "Description" UINT 16 HEX-U 0 2000 0
    GENERIC_READ_CONVERSION_START
      value * 2
    GENERIC_READ_CONVERSION_END
    GENERIC_WRITE_CONVERSION_START
      value / 2
    GENERIC_WRITE_CONVERSION_END
  PARAMETER "Other Param" "Yet another" INT 32 STATE -100 100 10
    STATE "OLD" 0
    STATE "NEW" 1
  PARAMETER "Next Param" "Another param" FLOAT 64 STATE -Float::MAX Float::MAX 0.0
    STATE OFF 0.0
    STATE ON  1.0
  PARAMETER "String Param" "This is a string" STRING 32 STRING 2 4 "TEST"
  PARAMETER "Block Param" "Raw data" BLOCK 64 NONE 0 0 0xAAAA5555AAAA5555

TABLE "Event Action" "Event Action Table" TWO_DIMENSIONAL LITTLE_ENDIAN 2
  PARAMETER "Event" "The event" UINT 16 HEX 0 65535
  PARAMETER "Action" "The action" INT 32 STATE 1 2
    STATE OLD 1
    STATE NEW 2
  DEFAULT 0 OLD
  DEFAULT 0x100 2
  DEFAULT 1000 NEW
```

Telemetry Screens

Improve this page

Table of Contents

TITLE

Telemetry widgets

ARRAY
BLOCK
FORMATFONTVALUE
FORMATVALUE
LABELPROGRESSBAR
LABELTRENDLIMITSBAR
LABELVALUE
LABELVALUEDESC
LABELVALUELIMITSBAR
LIMITSBAR
LINEGRAPH
PROGRESSBAR
RANGEBAR
TEXTBOX
TIMEGRAPH
TRENDBAR
TRENDLIMITSBAR
VALUE

Interactive Widgets

BUTTON
CHECKBUTTON
COMBOBOX
RADIOBUTTON
TEXTFIELD

Canvas Widgets

CANVAS
CANVASLABEL
CANVASLABELVALUE
CANVASIMAGE
CANVASIMAGEVALUE
CANVASLINE
CANVASLINEVALUE

Widget Settings

Common Settings

BACKCOLOR
TEXTCOLOR
WIDTH
HEIGHT

Widget-Specific Settings

BORDERCOLOR
COLORBLIND
ENABLE_AGING
GRAY_RATE / GREY_RATE
GRAY_TOLERANCE / GREY_TOLERANCE
MIN_GRAY / MIN_GREY
TREND_SECONDS
VALUE_EQ
VALUE_GT
VALUE_GTEQ
VALUE_LT
VALUE_LTEQ
TLM_AND
TLM_OR

This document provides the information necessary to generate and use COSMOS Telemetry Screens, which are displayed by the COSMOS Telemetry Viewer application.

Definitions

NAME	DEFINITION
Widget	A widget is a graphical element on a COSMOS telemetry screen. It could display text, graph data, provide a button, or perform any other display/user input task.
Screen	A screen is a single window that contains any number of widgets which are organized and layed-out in a useful fashion.
Screen Definition File	A screen definition file is an ASCII file that tells COSMOS Telemetry Viewer how to draw a screen. It is made up of a series of keyword/parameter lines that define the telemetry points that are displayed on the screen and how to display them.

Telemetry Viewer Configuration

Two different types of configuration files are used to configure the COSMOS Telemetry Viewer; the screen definition files and a configuration file that lets the tool know what screens are available and how they are organized.

Telemetry Screen Definition Files

Telemetry screen definition files define the the contents of telemetry screens. They take the general form of a SCREEN keyword followed by a series of widget keywords that define the telemetry screen. Screen definition files specific to a particular target go in that targets configuration folder. For example: config/targets/COSMOS/screens/version.txt. Screen definition files that combine telemetry from multiple targets typically go in the system target's screens folder. For example: config/targets/SYSTEM/screens/overall.txt.

Keywords:

SCREEN

The SCREEN keyword is the first keyword in any telemetry screen definition. It defines the name of the screen and parameters that affect the screen overall.

PARAMETER	DESCRIPTION	REQUIRED
Width	Width in pixels or AUTO to let Telemetry Viewer automatically layout the screen	Yes
Height	Height in pixels or AUTO to let Telemetry Viewer automatically layout the screen	Yes
Polling Period	Number of seconds between screen updates	Yes

Example Usage:

```
SCREEN AUTO AUTO 1.0
```

END

The END keyword is used to indicate the close of a layout widget. For example a VERTICALBOX keyword must be matched with an END keyword to indicate where the VERTICALBOX ends.

GLOBAL_SETTING

The GLOBAL_SETTING keyword is used to apply a widget setting to allow widgets of a certain type. (See SETTING)

PARAMETER	DESCRIPTION	REQUIRED
Widget Class Name	The name of the class of widgets that this setting will be applied to. For example: LABEL	Yes
Setting Name	Widget specific setting name	Yes

Setting Value(s)	Widget specific value(s) to set	Varies

Example Usage:

```
GLOBAL_SETTING LABELVALUELIMITSBAR COLORBLIND TRUE
```

GLOBAL_SUBSETTING

The GLOBAL_SUBSETTING keyword is used to apply a widget subsetting to allow widgets of a certain type. (See SUBSETTING)

PARAMETER	DESCRIPTION	REQUIRED
Widget Class Name	The name of the class of widgets that this setting will be applied to. For example: LABEL	Yes
Subwidget Index	Index to the desired subwidget or 'ALL'	Yes
Setting Name	Widget specific setting name	Yes
Setting Value(s)	Widget specific value(s) to set	Varies

Example Usage:

```
GLOBAL_SUBSETTING LABELVALUELIMITSBAR 1 COLORBLIND TRUE
GLOBAL_SUBSETTING LABELVALUELIMITSBAR 0:0 TEXTCOLOR white # Set all text color to white for labelvaluelimitsbars
```

SETTING

The SETTING keyword is used to apply a widget setting to the widget that was specified immediately before it.

PARAMETER	DESCRIPTION	REQUIRED
Setting Name	Widget specific setting name	Yes
Setting Value(s)	Widget specific value to set	Varies

Example Usage:

```
VERTICALBOX
  LABEL ... # Various other widgets
END
SETTING BACKCOLOR 163 185 163 # RGB color for the box background
```

SUBSETTING

The SUBSETTING keyword is used to apply a widget subsetting to the widget that was specified immediately before it. Subsettings are only valid for widgets that are made up of more than one subwidget. For example, LABELVALUE is made up of a LABEL at subwidget index 0 and a VALUE at subwidget index 1. This allows for passing settings to specific subwidgets. Some widgets are made up of multiple subwidgets, e.g. LABELVALUELIMITSBAR. To set the label text color, pass '0:0' as the Subwidget Index to first index the LABELVALUE and then to the LABEL.

PARAMETER	DESCRIPTION	REQUIRED
Subwidget Index	Index to the desired subwidget or 'ALL'	Yes
Setting Name	Widget specific setting name	Yes
Setting Value(s)	Widget specific value to set	Varies

Example Usage:

```
VERTICALBOX
  LABELVALUE ...
    SUBSETTING 0 TEXTCOLOR blue # Change only the label's color
  LABELVALUELIMITSBAR ...
    SUBSETTING 0:0 TEXTCOLOR white # Change the label's text color to white
END
```

NAMED_WIDGET

The NAMED_WIDGET keyword is used to give a name to a widget that allows it to be accessed from other widgets using the get_named_widget method of Cosmos::Screen. Note that get_named_widget returns the widget itself and thus must be operated on using methods native to that widget.

PARAMETER	DESCRIPTION	REQUIRED
Widget Name	The unique name applied to the following widget instance. Names must be unique per screen.	Yes
Widget Type	One of the widget types listed in Widget Descriptions	Yes
Widget Parameters	The unique parameters for the given widget type	Yes

Example Usage:

```
NAMED_WIDGET heading TITLE "Main Heading"
BUTTON "Push" 'puts get_named_widget("heading").text'
```

WIDGETNAME

All other keywords in a telemetry screen definition define the name of a widget and its unique parameters. These aren't really keywords at all and widgets can have any name besides the real keywords listed above. Whenever a keyword is encountered that is unrecognized, it is assumed that a file of the form widgetname_widget.rb exists, and contains a class called WidgetnameWidget. Because of this convention, new widgets can be added to the system without any change to the telemetry screen definition format Please see the Widget Descriptions section below for the details on all widgets supplied with the COSMOS core system.

PARAMETER	DESCRIPTION	REQUIRED
Widget Type	One of the widget types listed in Widget Descriptions	Yes
Widget Parameters	The unique parameters for the given widget type	Yes

Example Usage: See the Example File

Example File

Example File: /config/targets//myscreen.txt

```
SCREEN AUTO AUTO 0.5
GLOBAL_SETTING LABELVALUELIMITSBAR COLORBLIND TRUE
VERTICAL
  TITLE "Instrument Health and Status"
    SETTING BACKCOLOR 162 181 205
    SETTING TEXTCOLOR black
  VERTICALBOX
    SECTIONHEADER "General Telemetry"
    BUTTON 'Start Collect' 'target_name = get_target_name("INST"); cmd("#{target_name} COLLECT with TYPE NORMAL, DURATION 5"
    SETTING BACKCOLOR 54 95 58
    SETTING TEXTCOLOR white
    FORMATVALUE INST HEALTH_STATUS COLLECTS "0x%08X"
    LABELVALUE INST HEALTH_STATUS COLLECT_TYPE
    LABELVALUE INST HEALTH_STATUS DURATION
    LABELVALUE INST HEALTH_STATUS ASCIICMD WITH_UNITS 30
  END
  SETTING BACKCOLOR 163 185 163
  VERTICALBOX
    SECTIONHEADER "Temperatures"
    LABELTRENDLIMITSBAR INST HEALTH_STATUS TEMP1 WITH_UNITS 5
    LABELVALUELIMITSBAR INST HEALTH_STATUS TEMP2
    LABELVALUELIMITSBAR INST HEALTH_STATUS TEMP3
    LABELVALUELIMITSBAR INST HEALTH_STATUS TEMP4
    SETTING GRAY_TOLERANCE 0.1
  END
  SETTING BACKCOLOR 203 173 158
  VERTICALBOX
    SECTIONHEADER "Ground Station"
    LABELVALUE INST HEALTH_STATUS GROUND1STATUS
    LABELVALUE INST HEALTH_STATUS GROUND2STATUS
  END
  VERTICALBOX
    LABELVALUE INST HEALTH_STATUS TIMEFORMATTED WITH_UNITS 30
    SCREENSHOTBUTTON
  END
  SETTING BACKCOLOR 207 171 169
END
SETTING BACKCOLOR 162 181 205
```

Telemetry Viewer Settings Files

A telemetry viewer settings file tells telemetry viewer what screens exist and how they should be categorized. The default setting files is called tlm_viewer.txt and is located in config/tools/tlm_viewer/tlm_viewer.txt.

Keywords:

AUTO_TARGETS

The AUTO_TARGETS keyword tells Telemetry Viewer to add all the screens defined in the screens directory of each target folder in the config/targets directory. Screens are grouped by target name in the display. For example: all the screens defined in config/targets/COSMOS/screens will be added to a single drop down selection labeled COSMOS.

Example Usage:

```
AUTO_TARGETS
```

AUTO_TARGET

The AUTO_TARGET keyword tells Telemetry Viewer to add all the screens defined in the screens directory of the specified target folder in the config/targets directory. Screens are grouped by target name in the display. For example: all the screens defined in config/targets/COSMOS/screens will be added to a single drop down selection labeled COSMOS. If AUTO_TARGETS is used this keyword does nothing.

Example Usage:

```
AUTO_TARGET COSMOS
```

NEW_COLUMN

The NEW_COLUMN keywords creates a new column of drop down selections in Telemetry Viewer. All the AUTO_TARGET or SCREEN keywords after this keyword will be added to a new column in the GUI.

TARGET

The TARGET keyword is used to call out individual screens within a targets screen directory. It is used in conjunction with the SCREEN keyword.

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the target directory to look for screens.	Yes

Example Usage:

```
TARGET COSMOS
```

SCREEN

The SCREEN keyword adds the specified screen from the specified target. It must follow the TARGET keyword and is typically indented to show ownership to the target.

PARAMETER	DESCRIPTION	REQUIRED
File name	Name of the file containing the telemetry screen definition. The filename will be upcased and used in the drop down selection.	Yes
X position	Position in pixels to draw the left edge of the screen on the display. If not supplied the screen will be centered. If supplied, the Y position must also be supplied.	No
Y position	Position in pixels to draw the top edge of the screen on the display. If not supplied the screen will be centered. If supplied, the X position must also be supplied.	No

Example Usage:

```
TARGET COSMOS
  SCREEN version.txt 50 50
```

SHOW_ON_STARTUP

The SHOW_ON_STARTUP keyword causes the previously defined SCREEN to be automatically displayed when Telemetry Viewer starts. It must be preceeded by the SCREEN keyword.

ADD_SHOW_ON_STARTUP

The ADD_SHOW_ON_STARTUP keyword adds show on startup to any screen that has already been defined. This is

useful for adding show on startup to screens defined with AUTO_TARGETS.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Target Name of the screen	Yes
Screen Name	Base Name of the screen. This is equal to the screens filename with the .txt extension.	Yes
X position	Position in pixels to draw the left edge of the screen on the display. If not supplied the screen will be centered. If supplied, the Y position must also be supplied.	No
Y position	Position in pixels to draw the top edge of the screen on the display. If not supplied the screen will be centered. If supplied, the X position must also be supplied.	No

Example Usage:

```
ADD_SHOW_ON_STARTUP INST HS 500 300
ADD_SHOW_ON_STARTUP INST ADCS
```

Example File

Example File: /config/tools/tlm_viewer/tlm_viewer.txt

```
TARGET INST
  SCREEN "adcs.txt"
  SCREEN "array.txt"
TARGET INST2
  SCREEN "commanding.txt" 898 317
    SHOW_ON_STARTUP
  SCREEN "hs.txt"
TARGET COSMOS
  SCREEN "version.txt"
```

Widget Descriptions

This section describes the usage of all the telemetry screen widgets that are provided by the core COSMOS system.

Layout Widgets

Layout widgets are used to position other widgets on the screen. For example, the HORIZONTAL layout widget places the widgets it encapsulates horizontally on the screen.

VERTICAL

The VERTICAL widget places the widgets it encapsulates vertically on the screen. The screen defaults to a vertical layout, so if no layout widgets are specified, all widgets will be automatically placed within a VERTICAL layout widget. The VERTICAL widget sizes itself to fit its contents.

PARAMETER	DESCRIPTION	REQUIRED
Vertical spacing	Vertical spacing between widgets in pixels (default = 1)	No

Example Usage:

```
VERTICAL 50
  LABEL "TEST"
  LABEL "SCREEN"
END
```

VERTICALBOX

The VERTICALBOX widget places the widgets it encapsulates vertically on the screen inside of a thin border. The VERTICALBOX widget sizes itself to fit its contents vertically and to fit the screen horizontally.

PARAMETER	DESCRIPTION	REQUIRED
Title	Text to place within the border to label the box	No
Vertical spacing	Vertical spacing between widgets in pixels (default = 1)	No

Example Usage:

```
VERTICALBOX Info
  LABEL "TEST"
  LABEL "SCREEN"
END
```

HORIZONTAL

The HORIZONTAL widget places the widgets it encapsulates horizontally on the screen. The HORIZONTAL widget sizes itself to fit its contents.

PARAMETER	DESCRIPTION	REQUIRED
Horizontal spacing	Horizontal spacing between widgets in pixels (default = 1)	No

Example Usage:

```
HORIZONTAL 100
  LABEL "TEST"
  LABEL "SCREEN"
END
```

HORIZONTALBOX

The HORIZONTALBOX widget places the widgets it encapsulates horizontally on the screen inside of a thin border. The HORIZONTALBOX widget sizes itself to fit its contents.

PARAMETER	DESCRIPTION	REQUIRED
Title	Text to place within the border to label the box	No
Horizontal spacing	Horizontal spacing between widgets in pixels (default = 1)	No

Example Usage:

```
HORIZONTALBOX Info 10
  LABEL "TEST"
  LABEL "SCREEN"
END
```

MATRIXBYCOLUMNS

The MATRIXBYCOLUMNS widget places the widgets into a table-like matrix. The MATRIXBYCOLUMNS widget sizes itself to fit its contents.

PARAMETER	DESCRIPTION	REQUIRED
Columns	The number of columns to create	Yes

Example Usage:

```
MATRIXBYCOLUMNS 3
  LABEL "COL 1"
  LABEL "COL 2"
  LABEL "COL 3"

  LABEL "100"
  LABEL "200"
  LABEL "300"
END
```

SCROLLWINDOW

The SCROLLWINDOW widget places the widgets inside of it into a scrollable area. The SCROLLWINDOW widget sizes itself to fit the screen in which it is contained.

Example Usage:

```
SCROLLWINDOW
VERTICAL
 LABEL "100"
 LABEL "200"
 LABEL "300"
 LABEL "400"
 LABEL "500"
 LABEL "600"
 LABEL "700"
 LABEL "800"
 LABEL "900"
END
END
```

TABBOOK

The TABBOOK widget creates a tabbed area in which to place TABITEM widgets to form a tabbed layout.

TABITEM

The TABITEM widget creates a tab into which to place widgets. The tab automatically acts like a VERTICAL widget.

PARAMETER	DESCRIPTION	REQUIRED
Tab text	Text to diplay in the tab	Yes

Example Usage:

```
TABBOOK
  TABITEM "Tab 1"
    LABEL "100"
    LABEL "200"
  END
  TABITEM "Tab 2"
    LABEL "300"
    LABEL "400"
  END
END
```

Decoration Widgets

Decoration widgets are used to enhance the appearance of the screen. They do not respond to input, nor does the output vary with telemetry.

LABEL

The LABEL widget displays text on the screen. Generally, label widgets contain a telemetry mnemonic and are placed next to the telemetry VALUE widget.

PARAMETER	DESCRIPTION	REQUIRED
Text	Text to display on the label	Yes

Example Usage:

```
LABEL "Note: This is only a warning"
```

HORIZONTALLINE

The HORIZONTALLINE widget displays a horizontal line on the screen that can be used as a separator.

SECTIONHEADER

The SECTIONHEADER widget displays a label that is underlined with a horizontal line. Generally, SECTIONHEADER widgets are the first widget placed inside of a VERTICALBOX widget.

PARAMETER	DESCRIPTION	REQUIRED
Text	Text to display above the horizontal line	Yes

Example Usage:

```
SECTIONHEADER Mechanisms
```

TITLE

The TITLE widget displays a large centered title on the screen.

PARAMETER	DESCRIPTION	REQUIRED
Text	Text to display above the horizontal line	Yes

Example Usage:

```
TITLE "Title"
HORIZONTALLINE
SECTIONHEADER "Section Header"
LABEL "Label"
```

Telemetry widgets

Telemetry widgets are used to display telemetry values. The first parameters to each of these widgets is a telemetry mnemonic. Depending on the type and purpose of the telemetry item, the screen designer may select from a wide selection of widgets to display the value in the most useful format. They are listed here in alphabetical order.

ARRAY

The ARRAY widget is used to display data from an array telemetry item. Data is organized into rows and by default space separated.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Width	Width of the widget (default = 200)	No
Height	Height of the widget (default = 100)	No
Format string	Format string applied to each array item (default = nil)	No
Items per row	Number of array items per row (default = 4)	No
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No

Example Usage:

```
ARRAY INST HEALTH_STATUS ARY 250 50 "0x%x" 6 FORMATTED
ARRAY INST HEALTH_STATUS ARY2 200 60 nil 4 WITH_UNITS
```

BLOCK

The BLOCK widget is used to display data from a block telemetry item. Data is organized into rows and space separated.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Width	Width of the widget (default = 200)	No
Height	Height of the widget (default = 100)	No
Format string	Format string applied to byte of the block (default = "%02X")	No
Bytes per word	Number of bytes per word (default = 4)	No
Words per row	Number of words per row (default = 4)	No
Address format	Format for the address printed at the beginning of each line (default = nil which means do not print an address)	No

Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = RAW)	No

Example Usage:

```
BLOCK INST IMAGE IMAGE 400 130 "%02X" 4 4 "0x%08X:"
```

FORMATFONTVALUE

The FORMATFONTVALUE widget displays a box with a value printed inside that is formatted by the specified string rather than by a format string given in the telemetry definition files. Additionally, this widget can use a specified font. The white portion of the box darkens to gray while the value remains stagnant, then brightens to white each time the value changes. Additionally the value is colored based on the items limits state (Red for example if it is out of limits).

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Format string	Printf style format string to apply to the telemetry item	Yes
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No
Number of characters	The number of characters wide to make the value box (default = 12)	No
Font name	The font to use. (default = arial)	No
Font size	The font size. (default = 100)	No

Example Usage:

```
FORMATFONTVALUE INST LATEST TIMESEC %012u CONVERTED 12 arial 15
```

FORMATVALUE

The FORMATVALUE widget displays a box with a value printed inside that is formatted by the specified string rather than by a format string given in the telemetry definition files. The white portion of the box darkens to gray while the value remains stagnant, then brightens to white each time the value changes. Additionally the value is colored based on the items limits state (Red for example if it is out of limits).

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Format string	Printf style format string to apply to the telemetry item	Yes
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No
Number of characters	The number of characters wide to make the value box (default = 12)	No

Example Usage:

```
FORMATVALUE INST LATEST TIMESEC %012u CONVERTED 12
```

LABELPROGRESSBAR

The LABELPROGRESSBAR widget displays a LABEL widget showing the items name followed by a PROGRESSBAR widget to show the items value.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Scale factor	Value to multiple the telemetry item by before displaying the in the progress bar. Final value should be in the range of 0 to 100. (default 1.0)	No
Width	Width of the progress bar (default = 80 pixels)	No
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No

Example Usage:

```
LABELPROGRESSBAR INST ADCS POSPROGRESS 2 200 RAW
LABELPROGRESSBAR INST ADCS POSPROGRESS
```

LABELTRENDLIMITSBAR

The LABELTRENDLIMITSBAR widget displays a LABEL widget to show the item's name, a VALUE widget to show the telemetry items current value, a VALUE widget to display the value of the item X seconds ago, and a TRENDBAR widget to display the items value within its limits ranges and its trend.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No
Trend seconds	The number of seconds in the past to display the trend value (default = 60)	No
Characters	The number of characters to display the telemetry value (default = 12)	No
Width	Width of the limits bar (default = 160)	No
Height	Height of the limits bar (default = 25)	No

Example Usage

```
LABELTRENDLIMITSBAR INST HEALTH_STATUS TEMP1 CONVERTED 5 20 200 50
LABELTRENDLIMITSBAR INST HEALTH_STATUS TEMP1
```

LABELVALUE

The LABELVALUE widget displays a LABEL widget to shows the telemetry items name followed by a VALUE widget to display the items value.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes

Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = WITH_UNITS)	No
Number of characters	The number of characters wide to make the value box (default = 12)	No

Example Usage:

```
LABELVALUE INST LATEST TIMESEC CONVERTED 18
LABELVALUE INST LATEST COLLECT_TYPE
```

LABELVALUEDESC

The LABELVALUEDESC widget displays a LABEL widget to shows the telemetry items description followed by a VALUE widget to display the items value.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Description	The description to display in the label (default is to display the description text associated with the telemetry item)	No
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = WITH_UNITS)	No
Number of characters	The number of characters wide to make the value box (default = 12)	No

Example Usage:

```
LABELVALUEDESC INST LATEST TIMESEC "Time in seconds" CONVERTED 18
LABELVALUEDESC INST LATEST COLLECT_TYPE
```

LABELVALUELIMITSBAR

The LABELVALUELIMITSBAR widget displays a LABEL widget to shows the telemetry item's name, followed by a VALUE widget to display the items value, followed by a LIMITSBAR widget.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = WITH_UNITS)	No
Number of characters	The number of characters wide to make the value box (default = 12)	No

Example Usage:

```
LABELVALUELIMITSBAR INST HEALTH_STATUS TEMP1 CONVERTED 18
LABELVALUELIMITSBAR INST HEALTH_STATUS TEMP1
```

LIMITSBAR

The LIMITSBAR widget displays a graphical representation of where an items value falls withing its limits ranges.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No
Width	The width of the range bar (default = 160)	No
Height	The height of the range bar (default = 25)	No

Example Usage:

```
LIMITSBAR INST HEALTH_STATUS TEMP1 CONVERTED 200 50
LIMITSBAR INST HEALTH_STATUS TEMP1
```

LINEGRAPH

The LINEGRAPH widget displays a line graph of a telemetry items value verses sample number.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Num samples	The number of samples to display on the graph (default = 100)	No
Width	The width of the graph (default = 300)	No
Height	The height of the graph (default = 200)	No
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No

Example Usage:

```
LINEGRAPH INST HEALTH_STATUS TEMP1
LINEGRAPH INST HEALTH_STATUS TEMP1 10 400 100 RAW
```

PROGRESSBAR

The PROGRESSBAR widget displays a progress bar that is useful for displaying percentages.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Scale factor	Value to multiple the telemetry item by before displaying the in the progress bar. Final value should be in the range of 0 to 100. (default 1.0)	No
Width	Width of the progress bar (default = 80 pixels)	No

Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No

Example Usage:

```
PROGRESSBAR INST ADCS POSPROGRESS 0.5 200
PROGRESSBAR INST ADCS POSPROGRESS
```

RANGEBAR

The RANGEBAR widget displays a graphical representation of where an items value falls withing a range.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Low Value	Minimum value to display on the range bar. If the telemetry item goes below this value the bar is "pegged" on the low end.	Yes
High Value	Maximum value to display on the range bar. If the telemetry item goes above this value the bar is "pegged" on the high end.	Yes
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No
Width	Width of the range bar (default = 160)	No
Height	Height of the range bar (default = 25)	No

Example Usage:

```
RANGEBAR INST HEALTH_STATUS TEMP1 0 100 CONVERTED 200 50
RANGEBAR INST HEALTH_STATUS TEMP1 -1000 1000
```

TEXTBOX

The TEXTBOX widget provides a large box for multiline text.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Width	Width of the text box (default = 200)	No
Height	Height of the text box (default = 100)	No

Example Usage:

```
TEXTBOX INST HEALTH_STATUS TIMEFORMATTED 150 50
TEXTBOX INST HEALTH_STATUS TIMEFORMATTED
```

TIMEGRAPH

The TIMEGRAPH widget displays a line graph of a telemetry items value verses time.

PARAMETER	DESCRIPTION	REQUIRED

Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Num samples	The number of samples to display on the graph (default = 100)	No
Width	The width of the graph (default = 300)	No
Height	The height of the graph (default = 200)	No
Show points	Whether to show points or just draw lines between points (default = true)	No
Time item name	The telemetry item to use as the time on the X axis (default = TIMESECONDS)	No
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No

Example Usage:

```
TIMEGRAPH INST HEALTH_STATUS TEMP1
TIMEGRAPH INST HEALTH_STATUS TEMP1 10 400 100 false TIMESECONDS CONVERTED
```

TRENDBAR

The TRENDBAR widget provides the same functionality as the LIMITSBAR widget except that it also keeps a history of the telemetry item and graphically shows where the value was X seconds ago.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Value type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No
Trend seconds	The number of seconds in the past to display the trend value (default = 60)	No
Width	Width of the limits bar (default = 160)	No
Height	Height of the limits bar (default = 25)	No

Example Usage

```
TRENDBAR INST HEALTH_STATUS TEMP1 CONVERTED 20 200 50
TRENDBAR INST HEALTH_STATUS TEMP1
```

TRENDLIMITSBAR

The TRENDLIMITSBAR widget displays a VALUE widget to show the telemetry items current value, a VALUE widget to display the value of the item X seconds ago, and a TRENDBAR widget to display the items value within its limits ranges and its trend.

PARAMETER	DESCRIPTION	REQUIRED
Target name	Target name portion of the telemetry mnemonic	Yes
Packet name	Packet name portion of the telemetry mnemonic	Yes
Item name	Item name portion of the telemetry mnemonic	Yes
Value type	Type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS	No

Value type	(default = WITH_UNITS)	No
Trend seconds	Number of seconds in the past to display the trend value (default = 60)	No
Characters	Number of characters to display the value (default = 12)	No
Width	Width of the limits bar (default = 160)	No
Height	Height of the limits bar (default = 25)	No

Example Usage

```
TRENDLIMITSBAR INST HEALTH_STATUS TEMP1 CONVERTED 20 20 200 50
TRENDLIMITSBAR INST HEALTH_STATUS TEMP1
```

VALUE

The VALUE widget displays a box with a value printed inside. The white portion of the box darkens to gray while the value remains stagnant, then brightens to white each time the value changes. Additionally the value is colored based on the items limits state (Red for example if it is out of limits).

PARAMETER	DESCRIPTION	REQUIRED
Target name	Target name portion of the telemetry mnemonic	Yes
Packet name	Packet name portion of the telemetry mnemonic	Yes
Item name	Item name portion of the telemetry mnemonic	Yes
Value type	Type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = WITH_UNITS)	No
Characters	Number of characters to display the value (default = 12)	No

Example Usage:

```
VALUE INST HEALTH_STATUS TEMP1 CONVERTED 18
VALUE INST HEALTH_STATUS TEMP1
```

Interactive Widgets

Interactive widgets are used to gather input from the user. Unlike all other widgets, which only output some graphical representation, interactive widgets permit input either from the keyboard or mouse.

BUTTON

The BUTTON widget displays a rectangular button that is clickable by the mouse. Upon clicking, the button executes the Ruby code assigned. Buttons can be used to send commands and perform other tasks.

If you want your button to use values from other widgets, define them as named widgets and read their values using the `get_named_widget("WIDGET_NAME").text` method. See the example in CHECKBUTTON. If your button logic gets complex it's recommended to `require` a separate script and pass the screen to the script using self such as `require utility.rb; utility_method(self)`.

PARAMETER	DESCRIPTION	REQUIRED
Button text	Text displayed on the button	Yes
String to eval	Ruby code to execute when the button is pressed	Yes

Example Usage:

```
BUTTON 'Start Collect' 'cmd("INST COLLECT with TYPE NORMAL, DURATION 5")'
```

CHECKBUTTON

The CHECKBUTTON widget displays a check box. Note this is of limited use by itself and is primarily used in conjunction with NAMED_WIDGET.

PARAMETER	DESCRIPTION	REQUIRED
Checkbox text	Text displayed next to the checkbox	Yes

Example Usage:

```
NAMED_WIDGET CHECK CHECKBUTTON 'Ignore Hazardous Checks'
BUTTON 'Send' 'if get_named_widget("CHECK").checked? then cmd_no_hazardous_check("INST CLEAR") else cmd("INST CLEAR") end'
```

COMBOBOX

The COMBOBOX widget displays a drop down list of text items that the user can choose from. Note this is of limited use by itself and is primarily used in conjunction with NAMED_WIDGET.

PARAMETER	DESCRIPTION	REQUIRED
Option text	Text to display in the selection drop down	Yes

Example Usage:

```
BUTTON 'Start Collect' 'cmd("INST COLLECT with TYPE #{get_named_widget("COLLECT_TYPE").text}, DURATION 10.0")'
NAMED_WIDGET COLLECT_TYPE COMBOBOX NORMAL SPECIAL
```

RADIOBUTTON

The RADIOBUTTON widget a radio button and text. Note this is of limited use by itself and is primarily used in conjunction with NAMED_WIDGET.

PARAMETER	DESCRIPTION	REQUIRED
Text	Text to display next to the radio button	Yes

Example Usage:

```
NAMED_WIDGET ABORT RADIOBUTTON 'Abort'
NAMED_WIDGET CLEAR RADIOBUTTON 'Clear'
BUTTON 'Send' 'if get_named_widget("ABORT").checked? then cmd("INST ABORT") else cmd("INST CLEAR") end'
```

TEXTFIELD

The TEXTFIELD widget displays a rectangular box that the user can enter text into.

PARAMETER	DESCRIPTION	REQUIRED
Characters	Width of the text field in characters (default = 12)	No
Text	Default text to put in the text field (default is blank)	No

Example Usage:

```
NAMED_WIDGET DURATION TEXTFIELD 12 "10.0"
BUTTON 'Start Collect' 'cmd("INST COLLECT with TYPE NORMAL, DURATION #{get_named_widget("DURATION").text.to_f}")'
```

Canvas Widgets

Canvas Widgets are used to draw custom displays into telemetry screens. The canvas coordinate frame places (0,0) in the upper-left corner of the canvas.

CANVAS

The CANVAS widget is the layout widget for the other canvas widgets. All canvas widgets must be enclosed within a CANVAS widget. It is included with the other CANVAS widgets rather than in the layout section for simplicity.

PARAMETER	DESCRIPTION	REQUIRED
Width	Width of the canvas	Yes
Height	Height of the canvas	Yes

Example Usage: See the other Canvas examples

CANVASLABEL

The CANVASLABEL widget draws text onto the canvas.

PARAMETER	DESCRIPTION	REQUIRED
X	X position of the upper-left corner of the text on the canvas	Yes
Y	Y position of the upper-left corner of the text on the canvas	Yes
Text	Text to draw onto the canvas	Yes
Font size	Font size of the text (default = 12)	No
Color	Color of the text (default = 'black')	No

Example Usage:

```
CANVAS 100 100
  CANVASLABEL 5 34 "Label1" 24 red
  CANVASLABEL 5 70 "Label2" 18 blue
END
```

CANVASLABELVALUE

The CANVASLABELVALUE widget draws the text value of a telemetry item onto the canvas in an optional frame.

PARAMETER	DESCRIPTION	REQUIRED
target name	The target name portion of the telemetry mnemonic	Yes
packet name	The packet name portion of the telemetry mnemonic	Yes
item name	The item name portion of the telemetry mnemonic	Yes
X	X position of the upper-left corner of the text on the canvas	Yes
Y	Y position of the upper-left corner of the text on the canvas	Yes
Font size	Font size of the text (default = 12)	No
Color	Color of the text (default = 'black')	No
Frame	Whether to draw a frame around the value in the same color as the font	No
Frame width	Width in pixels of the frame	No

Value type	Type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	No

Example Usage:

```
CANVAS 200 100
  CANVASLABELVALUE INST HEALTH_STATUS TEMP1 5 34 12 red true 5
  CANVASLABELVALUE INST HEALTH_STATUS TEMP2 5 70 10 blue false 0 WITH_UNITS
END
```

CANVASIMAGE

The CANVASIMAGE widget displays a GIF image on the canvas.

PARAMETER	DESCRIPTION	REQUIRED
Image name	Name of a image file. The file must be located in the /data directory.	Yes
X	Left X position to draw the image	Yes
Y	Top Y position to draw the image	Yes

Example Usage:

```
CANVAS 300 300
  CANVASIMAGE "satellite.gif" 0 0
END
```

CANVASIMAGEVALUE

The CANVASIMAGEVALUE widget displays a GIF image on the canvas that changes with a telemetry value.

PARAMETER	DESCRIPTION	REQUIRED
Target name	Target name portion of the telemetry mnemonic	Yes
Packet name	Packet name portion of the telemetry mnemonic	Yes
Item name	Item name portion of the telemetry mnemonic	Yes
Filename prefix	The prefix part of the filename of the gif images (expected to be in the user's data directory). The actual filenames will be this value plus the word "on" or the word "off" and ".gif"	Yes
X	X position of the upper-left corner of the image on the canvas	Yes
Y	Y position of the upper-left corner of the image on the canvas	Yes
Value type	Type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = RAW)	No

Example Usage:

```
CANVAS 150 200
  CANVASLABELVALUE INST HEALTH_STATUS GROUND1STATUS 0 12 12 black false
  CANVASIMAGEVALUE INST HEALTH_STATUS GROUND1STATUS "ground" 0 20 # Uses groundon.gif and groundoff.gif
END
```

CANVASLINE

The CANVASLINE widget draws a line onto the canvas.

PARAMETER	DESCRIPTION	REQUIRED

X1	X position of the first endpoint of the line on the canvas	Yes
Y1	Y position of the first endpoint of the line on the canvas	Yes
X2	X position of the second endpoint of the line on the canvas	Yes
Y2	Y position of the second endpoint of the line on the canvas	Yes
Color	Color of the line(default = 'black')	No
Width	Width of the line in pixels (default = 1)	No
Connector	Indicates whether or not to draw a circle at the second endpoint of the line: NO_CONNECTOR or CONNECTOR (default = NO_CONNECTOR)	No

Example Usage:

```
CANVAS 100 50
  CANVASLINE 5 5 95 5
  CANVASLINE 5 5 5 45 green 2 CONNECTOR
  CANVASLINE 95 5 95 45 blue 3 CONNECTOR
END
```

CANVASLINEVALUE

The CANVASLINEVALUE widget draws a line onto the canvas in one of two colors based on the value of the associated telemetry item.

PARAMETER	DESCRIPTION	REQUIRED
Target name	Target name portion of the telemetry mnemonic	Yes
Packet name	Packet name portion of the telemetry mnemonic	Yes
Item name	Item name portion of the telemetry mnemonic	Yes
X1	X position of the first endpoint of the line on the canvas	Yes
Y1	Y position of the first endpoint of the line on the canvas	Yes
X2	X position of the second endpoint of the line on the canvas	Yes
Y2	Y position of the second endpoint of the line on the canvas	Yes
Color on	Color of the line when the telemtry point is considered on (default = 'green')	No
Color off	Color of the line when the telemtry point is considered off (default = 'blue')	No
Width	Width of the line in pixels (default = 3)	No
Connector	Indicates whether or not to draw a circle at the second endpoint of the line: NO_CONNECTOR or CONNECTOR (default = NO_CONNECTOR)	No
Value type	Type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = RAW)	No

Example Usage:

```
CANVAS 120 50
  CANVASLABELVALUE INST HEALTH_STATUS GROUND1STATUS 0 12 12 black false
  CANVASLINEVALUE INST HEALTH_STATUS GROUND1STATUS 5 25 115 25
  CANVASLINEVALUE INST HEALTH_STATUS GROUND1STATUS 5 45 115 45 purple red 3 CONNECTOR
END
```

Widget Settings

Settings allow for additional tweaks and options to be applied to widgets that are not available in their constructors. These settings are all configured through the SETTING and GLOBAL_SETTING keywords. SETTING applies only to the widget defined immediately before it. GLOBAL_SETTING applies to all widgets.

Common Settings

The following settings are available to all widgets if their underlying Qt GUI object supports them.

BACKCOLOR

The BACKCOLOR setting sets the background color for a widget.

PARAMETER	DESCRIPTION	REQUIRED
Color name	Common name for the color, e.g. 'black', 'red', etc	Yes

or

PARAMETER	DESCRIPTION	REQUIRED
Red value	Red portion of an RGB value (0-255)	Yes
Green value	Green portion of an RGB value (0-255)	Yes
Blue value	Blue portion of an RGB value (0-255)	Yes

Example Usage:

```
SETTING BACKCOLOR red
SETTING BACKCOLOR 162 181 205
```

TEXTCOLOR

The TEXTCOLOR setting sets the text color for a widget.

PARAMETER	DESCRIPTION	REQUIRED
Color name	Common name for the color, e.g. 'black', 'red', etc	Yes

PARAMETER	DESCRIPTION	REQUIRED
Red value	Red portion of an RGB value (0-255)	Yes
Green value	Green portion of an RGB value (0-255)	Yes
Blue value	Blue portion of an RGB value (0-255)	Yes

Example Usage:

```
SETTING TEXTCOLOR red
SETTING TEXTCOLOR 162 181 205
```

WIDTH

The WIDTH setting forces the height of a widget to a certain size.

PARAMETER	DESCRIPTION	REQUIRED
Width	Desired with in pixels	Yes

Example Usage:

```
SETTING WIDTH 100
```

HEIGHT

The HEIGHT setting forces the height of a widget to a certain size.

PARAMETER	DESCRIPTION	REQUIRED
Height	Desired height in pixels	Yes

Example Usage:

```
SETTING HEIGHT 100
```

Widget-Specific Settings

The following settings are only available to the widgets listed.

BORDERCOLOR

The BORDERCOLOR setting changes the color of a layout widgets border.

PARAMETER	DESCRIPTION	REQUIRED
Color name	Common name for the color, e.g. 'black', 'red', etc	Yes

or

PARAMETER	DESCRIPTION	REQUIRED
Red value	Red portion of an RGB value (0-255)	Yes
Green value	Green portion of an RGB value (0-255)	Yes
Blue value	Blue portion of an RGB value (0-255)	Yes

Example Usage:

```
HORIZONTALBOX
  LABEL "Label 1"
  LABEL "Label 2"
END
SETTING BORDERCOLOR red
```

COLORBLIND

The COLORBLIND setting enables/disables providing clues in visualization for users that are colorblind. Supported by all VALUE widgets.

PARAMETER	DESCRIPTION	REQUIRED
Enable	TRUE or FALSE	Yes

Example Usage:

```
LABELVALUELIMITSBAR INST HEALTH_STATUS TEMP1
  SETTING COLORBLIND TRUE
```

ENABLE_AGING

The ENABLE_AGING setting enables/disables graying of widgets if there value doesn't change. Supported by ARRAY, BLOCK, and all VALUE widgets.

PARAMETER	DESCRIPTION	REQUIRED
Enable	TRUE or FALSE	Yes

Example Usage:

```
LABELVALUE INST HEALTH_STATUS COLLECTS
  SETTING ENABLE_AGING FALSE
```

GRAY_RATE / GREY_RATE

The GRAY_RATE and GREY_RATE settings change the rate at which graying occurs in widgets. Supported by ARRAY, BLOCK, and all VALUE widgets.

PARAMETER	DESCRIPTION	REQUIRED
Gray rate	The number of shades of gray that are subtracted at each polling period if the value hasn't changed	Yes

Example Usage:

```
LABELVALUE INST HEALTH_STATUS COLLECTS
  SETTING GRAY_RATE 5
```

GRAY_TOLERANCE / GREY_TOLERANCE

The GRAY_TOLERANCE and GREY_TOLERANCE settings set the maximum change in value that will not cause the widget to recognize an items value as changing. Supported by all VALUE widgets.

PARAMETER	DESCRIPTION	REQUIRED
Tolerance value	The maximum change in value that will cause the widget to not recognize an items value as changing.	Yes

Example Usage:

```
LABELVALUE INST HEALTH_STATUS COLLECTS
  SETTING GRAY_TOLERANCE 1
```

MIN_GRAY / MIN_GREY

The MIN_GRAY and MIN_GREY settings set the minimum shade of a gray that a widget will decay to if its value doesn't change. Supported by ARRAY, BLOCK, and all VALUE widgets.

PARAMETER	DESCRIPTION	REQUIRED
Minimum gray	The minimum shade of a gray that a widget will decay to if its value doesn't change. Must be a value between 0 (black) and 255 (white). (default = 200)	Yes

Example Usage:

```
LABELVALUE INST HEALTH_STATUS TEMP1
  SETTING GRAY_TOLERANCE 1000 # Prevent the widget from refreshing by choosing a high tolerance
  SETTING MIN_GRAY 0 # Set the minimum gray to black
```

TREND_SECONDS

The TREND_SECONDS setting changes the number of seconds using during trending. Supported by the TREND widgets.

PARAMETER	DESCRIPTION	REQUIRED
Seconds	The number of seconds to trend across	Yes

Example Usage:

```
TRENDBAR INST HEALTH_STATUS TEMP1
  SETTING TREND_SECONDS 10
```

VALUE_EQ

The VALUE_EQ setting configures for an equal to comparison for a canvas value widget to determine 'ON' state. Supported widgets: CANVASIMAGEVALUE, CANVASLABELVALUE, CANVASLINEVALUE.

PARAMETER	DESCRIPTION	REQUIRED
Value	The value to compare against with ==	Yes

Example Usage:

```
CANVASIMAGEVALUE INST HEALTH_STATUS GROUND1STATUS "ground" 400 100
  SETTING VALUE_EQ 0
```

VALUE_GT

The VALUE_GT setting configures for a greater than comparison for a canvas value widget to determine 'ON' state. Supported widgets: CANVASIMAGEVALUE, CANVASLABELVALUE, CANVASLINEVALUE.

PARAMETER	DESCRIPTION	REQUIRED
Value	The value to compare against with >	Yes

Example Usage:

```
CANVASIMAGEVALUE INST HEALTH_STATUS TEMP1 "ground" 400 100
  SETTING VALUE_GT 10.0
```

VALUE_GTEQ

The VALUE_GTEQ setting configures for a greater than or equal to comparison for a canvas value widget to determine 'ON' state. Supported widgets: CANVASIMAGEVALUE, CANVASLABELVALUE, CANVASLINEVALUE.

PARAMETER	DESCRIPTION	REQUIRED
Value	The value to compare against with >=	Yes

Example Usage:

```
CANVASIMAGEVALUE INST HEALTH_STATUS TEMP1 "ground" 400 100
  SETTING VALUE_GTEQ 10.0
```

VALUE_LT

The VALUE_LT setting configures for a less than comparison for a canvas value widget to determine 'ON' state. Supported widgets: CANVASIMAGEVALUE, CANVASLABELVALUE, CANVASLINEVALUE.

PARAMETER	DESCRIPTION	REQUIRED
Value	The value to compare against with <	Yes

Example Usage:

```
CANVASIMAGEVALUE INST HEALTH_STATUS TEMP1 "ground" 400 100
  SETTING VALUE_LT 10.0
```

VALUE_LTEQ

The VALUE_LTEQ setting configures for a less than or equal to comparison for a canvas value widget to determine 'ON' state. Supported widgets: CANVASIMAGEVALUE, CANVASLABELVALUE, CANVASLINEVALUE.

PARAMETER	DESCRIPTION	REQUIRED
Value	The value to compare against with <=	Yes

Example Usage:

```
CANVASIMAGEVALUE INST HEALTH_STATUS TEMP1 "ground" 400 100
SETTING VALUE_LTEQ 10.0
```

TLM_AND

The TLM_AND setting allows added another comparison that is anded with the original comparison for a canvas value widget to determine 'ON' state. Supported widgets: CANVASIMAGEVALUE, CANVASLABELVALUE, CANVASLINEVALUE.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Comparison Type	The comparison type: VALUE_EQ, VALUE_GT, VALUE_GTEQ, VALUE_LT, or VALUE_LTEQ	Yes
Value	The value to compare against	Yes

Example Usage:

```
CANVASIMAGEVALUE INST HEALTH_STATUS TEMP1 "ground" 400 100
  SETTING VALUE_LTEQ 10.0
  SETTING TLM_AND INST HEALTH_STATUS TEMP2 VALUE_GT 20.0
```

TLM_OR

The TLM_OR setting allows added another comparison that is ored with the original comparison for a canvas value widget to determine 'ON' state. Supported widgets: CANVASIMAGEVALUE, CANVASLABELVALUE, CANVASLINEVALUE.

PARAMETER	DESCRIPTION	REQUIRED
Target name	The target name portion of the telemetry mnemonic	Yes
Packet name	The packet name portion of the telemetry mnemonic	Yes
Item name	The item name portion of the telemetry mnemonic	Yes
Comparison Type	The comparison type: VALUE_EQ, VALUE_GT, VALUE_GTEQ, VALUE_LT, or VALUE_LTEQ	Yes

Value	The value to compare against	Yes

Example Usage:

```
CANVASIMAGEVALUE INST HEALTH_STATUS TEMP1 "ground" 400 100
  SETTING VALUE_LTEQ 10.0
  SETTING TLM_OR INST HEALTH_STATUS TEMP2 VALUE_GT 20.0
```

BACK NEXT

Requirements and Design

Improve this page

Table of Contents

COSMOS is a command and control system providing commanding, scripting, and data visualization capabilities for embedded systems and systems of systems. COSMOS is intended for use during all phases of testing (board, box, integrated system) and during operations.

COSMOS is made up of the following 15 applications:

1. **Command and Telemetry Server** - Provides realtime commanding, telemetry reception, logging, limits monitoring, and packet routing.
2. **Replay** - Simulates Command and Telemetry Server for telemetry packet log file playback.
3. **Command Sender** - Provides a graphical interface for manually sending individual commands.

4. **Script Runner** - Executes test scripts and provides highlighting of the currently executing line.
5. **Test Runner** - Provides a high level framework for system level testing including test report generation.
6. **Packet Viewer** - Provides realtime visualization of every telemetry packet that has been defined.
7. **Telemetry Viewer** - Provides custom telemetry screen functionality with advanced layout and visualization widgets.
8. **Telemetry Grapher** - Provides realtime and offline graphing of telemetry data.
9. **Data Viewer** - Provides text based telemetry visualization for items.
10. **Limits Monitor** - Monitors telemetry with defined limits and shows items that currently are or have violated limits.
11. **Telemetry Extractor** - Extracts telemetry packet log files into CSV data.
12. **Command Extractor** - Extracts command packet log files into human readable text.
13. **Table Manager** - A binary file editor that can be used to build up configuration tables or other binary data.
14. **Handbook Creator** - Creates html and pdf documentation of available commands and telemetry packets.
15. **Launcher** - Provides a graphical interface for launching each of the tools that make up the COSMOS system.

More detailed descriptions, requirements, and design for each tool are found later in this document. Additionally, each of the above applications is built using COSMOS libraries that are available for use as a framework to develop custom program/project applications.

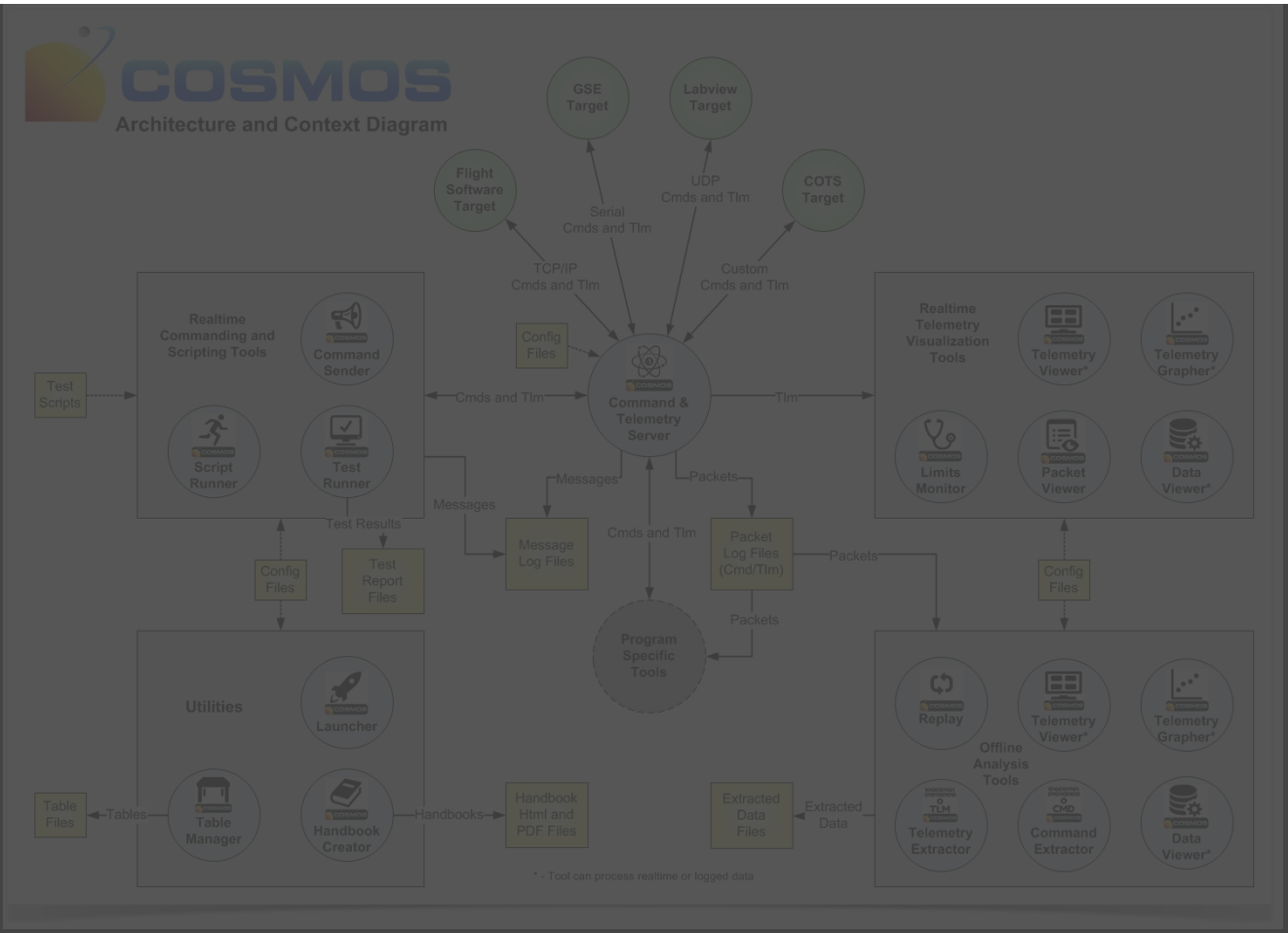
Terminology

The COSMOS system uses several terms that are important to understand. The following table defines these terms.

TERM	DEFINITION
Target	A COSMOS target is an embedded system that the COSMOS Command and Telemetry Server connects using and interface in in order to send commands to and/or receive telemetry from.
Command	A packet of information telling a target to perform an action of some sort.
Telemetry Packet	A packet of information providing status from a target.
Interface	A Ruby class that knows how to send commands to and/or receive telemetry from a target. COSMOS comes with interfaces that support TCP/IP, UDP, and serial connections. Custom interfaces are easy to add to the system.
Ruby	The powerful dynamic programming language used to write the COSMOS applications and libraries as well as COSMOS scripts and test procedures.
Configuration Files	COSMOS uses simple plain text configuration files to define commands and telemetry packets, and to configure each COSMOS application. These files are easily human readable/editable and machine readable/editable.
Packet Log Files	Binary files containing either logged commands or telemetry packets.
Message Log Files	Text files containing messages generated by a tool.
Tool	Another name for a COSMOS application.

Overall Architecture and Context Diagram

The following diagram shows how the 15 applications that make up the COSMOS system relate to each other and to the targets that COSMOS is controlling.



Key aspects of this architecture:

- The COSMOS tools are grouped into four broad categories:
  - Realtime Commanding and Scripting
  - Realtime Telemetry Visualization
  - Offline Analysis
  - Utilities
- COSMOS can connect to many different kinds of targets. The examples shown in this diagram include Flight software (FSW), Ground Support Equipment (GSE), Labview, and a COTS target such as an Agilent power supply. Any embedded system that provides a communication interface can be connected to COSMOS.
- COSMOS ships with interfaces for connecting over TCP/IP, UDP, and serial connections. This covers most systems, but custom interfaces can also be written to connect to anything that a computer can talk to.
- All realtime communication with targets flows through the Command and Telemetry Server. This ensures all commands and telemetry are logged.
- Every tool is configured with plain text configuration files.
- Program specific tools can be written using the COSMOS libraries that can interact with the realtime command and telemetry streams through the COSMOS Command and Telemetry Server and can process packet log files.

Overall Requirements

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
ALL-1	All COSMOS tools shall launch and run on Windows 7.	Start Launcher and launch every COSMOS tool on Windows 7.	All autohotkey scripts.
ALL-2	All COSMOS tools shall launch and run on CentOS Linux 6.5.	Start launcher and launch every COSMOS tool on CentOS Linux 6.5.	Manually verified on Linux 5-30-14.

ALL-3	All COSMOS tools shall launch and run on Mac OSX Mavericks.	Start launcher and launch every COSMOS tool on Mac OSX Mavericks.	Manually verified on Mac OSX 5-30-14.

Command and Telemetry Server

The Command and Telemetry server connects COSMOS to targets for real-time commanding and telemetry processing. All real-time COSMOS tools communicate with targets through the Command and Telemetry Server ensuring that all communications are logged.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
CTS-1	The Command and Telemetry Server shall display a list of all interfaces.	View the Interfaces tab.	cmd_tlm_server.ahk
CTS-2	The Command and Telemetry Server shall allow manual connection and disconnection of interfaces.	Press a GUI button to disconnect and connect an interface.	cmd_tlm_server.ahk
CTS-3	The Command and Telemetry Server shall allow scripted connection and disconnection of interfaces.	Disconnect and connect an interface from a script.	script_spec.rb
CTS-4	The Command and Telemetry Server shall display a list of all targets.	View the Targets tab.	cmd_tlm_server.ahk
CTS-5	The Command and Telemetry Server shall display a list of known commands.	View the Cmd Packets tab.	cmd_tlm_server.ahk
CTS-6	The Command and Telemetry Server shall display raw command data for the most recent command received.	View the Cmd Packets tab and click the View Raw button for a command.	cmd_tlm_server.ahk
CTS-7	The Command and Telemetry Server shall display a list of known telemetry packets.	View the Tlm Packets tab.	cmd_tlm_server.ahk
CTS-8	The Command and Telemetry Server shall display raw telemetry packet data for the most recent telemetry packet received.	View the Tlm Packets tab and click the View Raw button for a telemetry packet.	cmd_tlm_server.ahk
CTS-9	The Command and Telemetry Server shall display a list of all routers.	View the Routers tab.	cmd_tlm_server.ahk
CTS-10	The Command and Telemetry Server shall allow manual connection and disconnection of routers.	Press a GUI button to disconnect and connect a router.	cmd_tlm_server.ahk
CTS-11	The Command and Telemetry Server shall allow scripted connection and disconnection of routers.	Disconnect and connect a router from a script.	script_spec.rb
CTS-12	The Command and Telemetry Server shall display a list of all packet writers.	View the Logging tab.	cmd_tlm_server.ahk
CTS-13	The Command and Telemetry Server shall allow manual starting and stopping of packet logging.	Press the GUI buttons to manually start and stop logging on all, and individual command/telemetry logging.	cmd_tlm_server.ahk
CTS-14	The Command and Telemetry Server shall allow scripted starting and stopping of packet logging.	Start and stop logging on all, and individual command/telemetry logging from a script.	script_spec.rb
CTS-15	The Command and Telemetry Server shall allow manually setting the current limits set.	Select a different limits set from the combobox.	cmd_tlm_server.ahk

CTS-16	The Command and Telemetry Server shall allow scripted setting of the current limits set.	Select a different limits set from a script.	script_spec.rb
CTS-17	The Command and Telemetry Server shall support clearing counters.	Select Edit->Clear Counters and verify counters reset.	cmd_tlm_server.ahk
CTS-18	The Command and Telemetry Server shall support opening telemetry packets in Packet Viewer.	On the Tlm Packets tab click View in Packet Viewer for a telemetry packet.	cmd_tlm_server.ahk
CTS-19	The Command and Telemetry Server shall support autonomously attempting to connect to targets.	Verify targets are connected to upon starting the CTS.	cmd_tlm_server.ahk
CTS-20	The Command and Telemetry Server shall time stamp telemetry packets upon receipt.	Verify logged packets are timestamped.	packet_log_reader_spec.rb
CTS-21	The Command and Telemetry Server shall time stamp telemetry packets to a resolution of 1 millisecond or better. Note: This requirement only refers to resolution. COSMOS does not run on real-time operating systems and accuracy cannot be guaranteed.	View time stamps in log.	cmd_tlm_server.ahk
CTS-22	The Command and Telemetry Server shall time stamp received telemetry with a UTC timestamp.	Verify logged time stamps are as expected.	packet_log_reader_spec.rb
CTS-23	The Command and Telemetry Server shall display time stamps in local time.	View time stamps in log.	cmd_tlm_server.ahk
CTS-24	The Command and Telemetry Server shall support a COSMOS target that outputs the COSMOS version.	View COSMOS Version packet in Targets tab.	cmd_tlm_server.ahk
CTS-25	The Command and Telemetry Server shall maintain a timestamped log of commands received, limits violations, and errors encountered.	View COSMOS message log.	cmd_tlm_server.ahk

Replay

Replay allows the playing back of telemetry log files as if the data was being received in realtime. This allows other COSMOS tools like Packet Viewer and Telemetry Viewer to be used to view logged data.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
RPY-1	Replay shall playback telemetry packet logs	Start playback of a telemetry log file.	replay.ahk
RPY-2	Replay shall support sequential and reverse playback.	Playback in both forwards and reverse.	replay.ahk
RPY-3	Replay shall listen on the same port as the Command and Telemetry Server for API requests.	Verify both the Command and Telemetry Server and Replay cannot be running at the same time (with default settings).	replay.ahk
RPY-4	Replay shall support a variable playback delay.	Playback with differing delay settings.	replay.ahk
RPY-5	Replay shall support single-stepping packets.	Single step packets in forward and reverse.	replay.ahk
RPY-6	Replay shall show the first, current, and final timestamps within the log file.	View displayed timestamps.	replay.ahk

RPY-7	Replay shall support quickly jumping to any point within the log file.	Drag the slider to different points in the file.	replay.ahk

Command Sender

Command Sender provides an easy method to send single commands to targets. The graphical user interface provides simple dropdowns to quickly select the desired command to send organized by target name and command name. After the user has selected the command, they then fill in the desired command parameters and click send to send the command to the target.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
CMD-1	Command Sender shall allow selection of a command by target name and packet name.	Select a specific command by target name and packet name in the drop down menus.	cmd_sender.ahk
CMD-2	Command Sender shall allow sending the selected command.	Send the selected command by pressing the Send button.	cmd_sender.ahk
CMD-3	Command Sender shall display allow non-ignored parameters for the selected command.	Select a specific command and verify the expected parameters are shown.	cmd_sender.ahk
CMD-4	Command Sender shall provide a mechanism to select state values for command parameters with states.	Select a specific state value for a specific command with states.	cmd_sender.ahk
CMD-5	Command Sender shall allow sending a manually entered value for a command parameter with states.	Manually enter a value for a specific command with states.	cmd_sender.ahk
CMD-6	Command Sender shall refuse to send commands if required parameters are not provided.	Attempt to send a command with a required parameter not filled out.	cmd_sender.ahk
CMD-7	Command Sender shall support sending commands while ignoring range checking.	Enter Ignore Range Checking mode and then send a command with an out of range parameter.	cmd_sender.ahk
CMD-8	Command Sender shall optionally display state values in hex.	Enter "Display State Values in Hex" mode and verify state values are displayed as hex.	cmd_sender.ahk
CMD-9	Command Sender shall optionally display ignored command parameters.	Enter "Show Ignored Parameters" mode and verify ignored parameters are displayed.	cmd_sender.ahk
CMD-10	Command Sender shall support sending raw data to a specified interface.	Send a command from a "raw" file using the File->Send Raw menu option.	cmd_sender.ahk
CMD-11	Command Sender shall respect hazardous commands and notify the user before proceeding.	Send a hazardous command and verify a dialog box appears before the command is sent. Verify both accepting the dialog and declining to send.	cmd_sender.ahk
CMD-12	Command Sender shall keep a command history of each command sent.	Send a command and verify that it shows up in the command history box.	cmd_sender.ahk
CMD-13	Command Sender shall allow resending of any command in the command history.	Resend one of the commands in the command history box.	cmd_sender.ahk

Script Runner

Script Runner provides a visual interface for editing and executing test scripts/procedures. A full featured text editor

provided syntax highlighting and code completion while developing test procedures. During script execution, the currently executing line is highlighted and any logged messages are highlighted to the user. If any failure occurs, the script is paused and the user altered. The user can then decide whether to stop the script, or ignore the failure and continue. The user can also retry the failed lines, or other nearby lines before proceeding.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
SR-1	Script Runner shall provide a text editor for developing test scripts.	Open Script Runner and create a simple test script. Perform all standard file operations including New, Open, Reload, Close, Save, and Save As. Perform all standard editing operations such as Cut, Copy, Paste, Undo, Redo, Select All, and Comment/Uncomment lines.	script_runner.ahk
SR-2	Script Runner shall provide search and replace functionality.	Perform all standard search and replace functionality, including search, replace, find next, and find previous.	script_runner.ahk
SR-3	Script Runner shall provide code completion for cmd(), tlm(), and wait_check() COSMOS API methods. Note: Other methods may also be supported.	Create a script and exercise code completion on the mentioned keywords.	script_runner.ahk
SR-4	Script Runner shall execute Ruby-based COSMOS scripts.	Press start and execute a script.	script_runner.ahk
SR-5	Script Runner shall highlight the currently executing line of the script.	Verify that lines are highlighted as a test script executes.	script_runner.ahk
SR-6	Script Runner shall allow pausing an executing script.	Press pause button and verify script is paused. Press start to resume.	script_runner.ahk
SR-7	Script Runner shall allow stopping an executing script.	Press stop and verify script stops.	script_runner.ahk
SR-8	Script Runner shall pause an executing script upon the occurance of an error.	Create a script with a statement that is guaranteed to fail and verify that the script is paused.	script_runner.ahk
SR-9	Script Runner shall log commands sent.	Execute a script that sends a command and verify it is logged.	script_runner.ahk
SR-10	Script Runner shall log text written to STDOUT. Note: Typically through the puts method.	Execute a script that uses puts to write a message and verify it is logged.	script_runner.ahk
SR-11	Script Runner shall log wait times.	Execute a script that includes a wait method and verify wait time is logged.	script_runner.ahk
SR-12	Script Runner shall log errors that occur while the script is executing.	Create a script with a check statement that is guaranteed to fail and verify it is logged.	script_runner.ahk
SR-13	Script Runner shall log check statement success and failure.	Create a script with a check statement that is guaranteed to fail and one that is guaranteed to succeeed. Verify both the success and failure are logged.	script_runner.ahk
SR-14	Script Runner shall support executing selected lines.	Select a set of lines and execute them using Script->Execute Selected Lines.	script_runner.ahk
SR-15	Script Runner shall support executing selected lines while paused.	Select a set of lines and execute them from the right-click context menu.	script_runner.ahk
SR-16	Script Runner shall support	Place the mouse cursor at the desired first line and	

SR-16	starting a script from any line.	then select Script->Execute From Cursor.	script_runner.ahk
SR-17	Script Runner shall support a mnemonic checking function.	Select Script->Mnemonic Check.	script_runner.ahk
SR-18	Script Runner shall support a syntax checking function.	Select Script->Ruby Syntax Check.	script_runner.ahk
SR-19	Script Runner shall support viewing the script instrumentation.	Select Script->View Instrumented Script.	script_runner.ahk
SR-20	Script Runner shall support an disconnected mode to allow for executing scripts without a connection to the Command and Telemetry Server.	Make sure the Command and Telemetry Server is not running. Select Script->Toggle Disconnect. Execute a script with commands and check statements and verify that it runs to completion.	script_runner.ahk
SR-21	Script Runner shall support a Debug terminal to aid in debugging scripts.	Select Script->Toggle Debug.	script_runner.ahk
SR-22	Script Runner shall support a step mode where the script will stop and wait for use interaction after each line.	From the Debug terminal, press the Toggle Run/Step button, then execute a script. Press Go to progress through the script.	script_runner.ahk
SR-23	Script Runner shall support inserting a return statement into a running script.	In a script that calls a subfunction with an infinite loop, then press the insert return button in the debug terminal and ensure the subfunction returns.	script_runner.ahk
SR-24	Script Runner shall support breakpoint functionality.	Create a breakpoint then execute the script and verify it stops at the specified line.	script_runner.ahk
SR-25	Script Runner shall support configuring a delay between executing each line.	From File->Options set the line delay and then execute a script to observe the updated line delay.	script_runner.ahk
SR-26	Script Runner shall support monitoring and logging limits while a script is executing.	From File->Options select monitor limits, and then execute a script.	script_runner.ahk
SR-27	Script Runner shall support pausing an executing script is a red limit occurs.	From File->Options select monitor limits and Pause on red limit, and then execute a script.	script_runner.ahk

Test Runner

Test Runner provides a structured methodology for designing system level testing that mirrors the very successful pattern used in software unit tests. System level tests are built up of test cases that are organized into test groups. For example, you might have one test group that verified all of the requirements associated with a particular mechanism. Ideally you would break this down into individual test cases for different scenarios. One perhaps for opening a shutter, another for closing it, etc. Test cases are ideally small and independent tasks. A number of these test groups are then combined into an overall test suite which would be run to execute a major test such as EMI, or software FQT.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
TR-1	Test Runner shall support executing individual test suites.	Execute an individual test suite.	test_runner.ahk
TR-2	Test Runner shall support executing individual test groups.	Execute an individual test group.	test_runner.ahk
TR-3	Test Runner shall support executing individual test cases.	Execute an individual test case.	test_runner.ahk

TR-4	Test Runner shall support executing test group setup and teardown methods individually.	Execute a test group setup. Execute a test group teardown.	test_runner.ahk
TR-5	Test Runner shall support executing test suite setup and teardown methods individually.	Execute a test suite setup. Execute a test suite teardown.	test_runner.ahk
TR-6	Test Runner shall create a test report after executing any test.	Verify a test report is generated after executing a test suite.	test_runner.ahk
TR-7	Test Runner shall support creating a custom test suite.	Select File->Test Selection and create a custom test suite. Then execute the custom test suite and make sure only the selected test cases are executed.	test_runner.ahk
TR-8	Test Runner shall support redisplaying the most recent test report.	Select File->Show Results to see the most recent test report.	test_runner.ahk
TR-9	Test Runner shall support a debug terminal.	Select Script->Toggle Debug to display the debug terminal.	test_runner.ahk
TR-10	Test Runner shall support pausing when an error occurs.	Execute a test script with the pause on error box checked and without.	test_runner.ahk
TR-11	Test Runner shall support allowing the user to proceed on an error.	Execute a test script with the Allow go/retry on error box checked and without.	test_runner.ahk
TR-12	Test Runner shall support aborting execution on an error.	Execute a test script with the abort on error box checked and without.	test_runner.ahk
TR-13	Test Runner shall support looping a test.	Execute a test script with the loop testing box checked and without.	test_runner.ahk
TR-14	Test Runner shall support breaking the looping of a test on error.	Execute a test script with the break loop on error box checked and without.	test_runner.ahk
TR-15	Test Runner shall support a user readable flag indicating that loop testing is occurring.	Execute a test script that checks the $loop_testing variable while looping and again while not looping.	test_runner.ahk
TR-16	Test Runner shall support a user readable flag indicating manual operations are desired.	Execute a test script with the manual box checked and without.	test_runner.ahk

Packet Viewer

Packet Viewer provides a simple tool to view the realtime contents of any telemetry packet defined in the system in a tabular, key-value format.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
PV-1	Packet Viewer shall allow selection of a telemetry packet by target name and packet name.	Select a specific telemetry packet by target name and packet name in the drop down menus.	packet_viewer.ahk
PV-2	Packet Viewer shall display the contents of the selected telemetry packet.	Ensure all items of the selected telemetry packet are displayed and updating.	packet_viewer.ahk
PV-3	Packet Viewer shall display the description for each telemetry item.	Hover over telemetry items to view their descriptions in the status bar.	packet_viewer.ahk
PV-4	Packet Viewer shall provide a mechanism to get detailed information on a telemetry item.	Right click on a telemetry item and select "Details" from the context menu.	packet_viewer.ahk
	Packet Viewer shall provide a mechanism	Right click on a telemetry item and	

PV-5	to edit any editable fields on a telemetry item.	Right click on a telemetry item and select "Edit" from the context menu.	packet_viewer.ahk
PV-6	Packet Viewer shall provide a mechanism to view a graph of any telemetry item.	Right click on a telemetry item and select "Graph" from the context menu.	packet_viewer.ahk
PV-7	Packet Viewer shall color telemetry values based upon limits state.	View a packet with items containing limits and verify they are colored.	packet_viewer.ahk
PV-8	Packet Viewer shall support a configurable polling rate.	Select File->Options and change the polling rate.	packet_viewer.ahk
PV-9	Packet Viewer shall support a color-blind mode to allow distinguishing limits states for those who are color blind.	Select View->Color Blind Mode and verify that items with limits are also displayed with a textual indication of limits state color.	packet_viewer.ahk
PV-10	Packet Viewer shall support displaying telemetry in each of the four COSMOS value types (raw, converted, formatted, and formatted with units)	In the View menu, select each of the four value types and verify values are displayed accordingly.	packet_viewer.ahk
PV-11	Packet Viewer shall support quickly bringing up the packet for any telemetry point.	Use the search box.	packet_viewer.ahk

Telemetry Viewer

Telemetry Viewer provides a way to organize telemetry points into custom "screens" that allow for the creation of unique and organized views of telemetry data. Screens are made up of widgets or small GUI components that display telemetry in unique ways.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
TV-1	Telemetry Viewer shall display user-defined telemetry screens.	Open a telemetry screen.	tlm_viewer.ahk
TV-2	Telemetry Viewer shall display realtime data.	Verify telemetry screens show realtime data.	tlm_viewer.ahk
TV-3	Telemetry Viewer shall support saving open telemetry screens and their positions.	Open three telemetry screens and then select File->Save Configuration.	tlm_viewer.ahk
TV-4	Telemetry Viewer shall support generating telemetry screens.	Select File->Generate Screens then select a specific target to generate screens for. Restart Telemetry Viewer and open the generated screens.	tlm_viewer.ahk
TV-5	Telemetry Viewer shall support auditing screens for missing telemetry points.	Select File->Audit Screens vs. Tlm	tlm_viewer.ahk

Telemetry Grapher

Telemetry Grapher performs graphing of telemetry points in both realtime and log file playback.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
TG-1	Telemetry Grapher shall provide line graphs of telemetry points.	Add several housekeeping data objects to a plot.	tlm_grapher.ahk
TG-2	Telemetry Grapher shall provide x-y graphs of telemetry points.	Create a normal x-y plot and a single-x-y plot.	tlm_grapher.ahk
TG-3	Telemetry Grapher shall support realtime graphing of telemetry.	Press Start to start realtime graphing.	tlm_grapher.ahk

TG-4	Telemetry Grapher shall support graphing data from telemetry log files.	Select File->Open to graph data from a log file.	tlm_grapher.ahk
TG-5	Telemetry Grapher shall support multiple tabs.	Add multiple tabs.	tlm_grapher.ahk
TG-6	Telemetry Grapher shall support multiple plots per tab.	Add multiple plots.	tlm_grapher.ahk
TG-7	Telemetry Grapher shall support multiple telemetry points per plot.	Add multiple data objects to one plot.	tlm_grapher.ahk
TG-8	Telemetry Grapher shall support a variable refresh rate.	Edit the refresh rate.	tlm_grapher.ahk
TG-9	Telemetry Grapher shall support saving a variable number of data points.	Edit Points Saved.	tlm_grapher.ahk
TG-10	Telemetry Grapher shall support graphing a variable duration of time.	Edit Seconds Plotted.	tlm_grapher.ahk
TG-11	Telemetry Grapher shall support graphing a variable number of data points.	Edit Points Plotted.	tlm_grapher.ahk
TG-12	Telemetry Grapher shall support taking screenshots of graphs.	Use the menus to take screenshots.	tlm_grapher.ahk
TG-13	Telemetry Grapher shall support exporting graph data.	Use the menus to export data.	tlm_grapher.ahk
TG-14	Telemetry Grapher shall support running analysis on data points.	Create data objects with analysis processing.	tlm_grapher.ahk
TG-15	Telemetry Grapher shall provide a method of quickly adding telemetry points to the graph.	Using the Add Housekeeping Data Object search box.	tlm_grapher.ahk
TG-16	Telemetry Grapher shall support graphing with two independent Y-axis.	Create a data_object that plots on the right Y-axis.	tlm_grapher.ahk
TG-17	Telemetry Grapher shall support saving its configuration.	Save the current configuration.	tlm_grapher.ahk
TG-18	Telemetry Grapher shall support loading its configuration.	Load the previously saved configuration.	tlm_grapher.ahk

Data Viewer

Data Viewer provides for textual display of telemetry packets where other display methods are not a good fit. It is especially useful for memory dumps and for log message type data display.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
DV-1	Data Viewer shall support realtime processing of telemetry packets.	Press Start to start realtime processing.	data_viewer.ahk
DV-2	Data Viewer shall support log file playback of telemetry packets.	Select File->Open Log File to process a telemetry log file.	data_viewer.ahk
DV-3	Data Viewer shall support textual display of telemetry packets.	View the display of data.	data_viewer.ahk
DV-4	Data Viewer shall support multiple tabs for data display.	Switch between several tabs of data.	data_viewer.ahk
DV-5	Data Viewer shall support deleting tabs.	Delete a tab.	data_viewer.ahk
DV-6	Data Viewer shall support enabled and disabling packets within tabs.	Disable and Enable a packet.	data_viewer.ahk

Limits Monitor

Limits Monitor displays all telemetry points that are currently out of limits and also shows any telemetry points that have gone out of limits since Limits Monitor was started.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
LM-1	Limits Monitor shall display all telemetry points currently out of limits.	View displayed telemetry points.	limits_monior.ahk
LM-2	Limits Monitor shall support ignoring telemetry points.	Click ignore on a telemetry point.	limits_monior.ahk
LM-3	Limits Monitor shall keep a displayed log of limits violations.	View the log tab.	limits_monior.ahk
LM-4	Limits Monitor shall continue displaying a telemetry point that temporarily went out of limits.	Watch until a telemetry points returns to green.	limits_monior.ahk
LM-5	Limits Monitor shall support saving its configuration.	Save the configuration.	limits_monior.ahk
LM-6	Limits Monitor shall support loading its configuration.	Load the configuration.	limits_monior.ahk

Telemetry Extractor

Telemetry Extractor processes telemetry log files and extracts data into a CSV format for analysis in Excel or other tools.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
TE-1	Telemetry Extractor shall process one or more telemetry log files at a time.	Open and process a telemetry log file.	tlm_extractor.ahk
TE-2	Telemetry Extractor shall support adding individual telemetry points.	Add an individual telemetry point.	tlm_extractor.ahk
TE-3	Telemetry Extractor shall support adding entire telemetry packets.	Add an entire packet.	tlm_extractor.ahk
TE-4	Telemetry Extractor shall support adding entire telemetry targets.	Add all packets for a target.	tlm_extractor.ahk
TE-5	Telemetry Extractor shall support selecting the value type to extract for each telemetry point (RAW, CONVERTED, FORMATTED, WITH_UNITS)	Double click an item and change the value type.	tlm_extractor.ahk
TE-6	Telemetry Extractor shall support adding text columns.	Add a text column.	tlm_extractor.ahk
TE-7	Telemetry Extractor shall support opening the result into a text editor.	Press the Open in Text Editor button.	tlm_extractor.ahk
TE-8	Telemetry Extractor shall support opening the result in Excel on Windows.	Press the Open in Excel button.	tlm_extractor.ahk
TE-9	Telemetry Extractor shall support saving configurations.	Select File->Save Config	tlm_extractor.ahk
TE-10	Telemetry Extractor shall support loading configurations.	Select File->Load Config	tlm_extractor.ahk
TE-11	Telemetry Extractor shall support deleting items	Select an Item and press delete	tlm_extractor.ahk

Command Extractor

Command Extractor converts command packet logs into human readable text files.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
CE-1	Command Extractor shall process one or more command log files at a time.	Open a command log file.	cmd_extractor.ahk
CE-2	Command Extractor shall support opening the result into a text editor.	Press the Open in Text Editor button.	cmd_extractor.ahk
CE-3	Command Extractor shall support a mode that displays raw command data.	Select Mode->Include Raw Data and process a log file.	cmd_extractor.ahk

Table Manager

Table Manager provides a graphical user interface for editing binary files containing one or more tables of data.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
TBL-1	Table Manager shall create new table files given a table definition.	Create a new table file using File->New.	table_manager.ahk
TBL-2	Table Manager shall open existing table files.	Open an existing table using File->Open.	table_manager.ahk
TBL-3	Table Manager shall support checking files for invalid entries.	Select File->Check.	table_manager.ahk
TBL-4	Table Manager shall support displaying files as hex data.	Select File->Hex Dump	table_manager.ahk
TBL-5	Table Manager shall support creating human readable text files of table data.	Select File->Create Report.	table_manager.ahk
TBL-6	Table Manager shall support returning tables to default values.	Select Table->Default	table_manager.ahk
TBL-7	Table Manager shall support editing table files.	Open a table file. Save it. Reopen and verify edits.	table_manager.ahk
TBL-8	Table Manager shall support checking individual tables for invalid entries.	Select Table->Check	table_manager.ahk
TBL-9	Table Manager shall support display individual tables as hex data.	Select Table->Hex Dump	table_manager.ahk
TBL-10	Table Manager shall support dumping an individual table to its own binary file.	Select Table->Save Table Binary.	table_manager.ahk
TBL-11	Table Manager shall support committing modifications to a single table to another table file.	Select Table->Commit to Existing File	table_manager.ahk
TBL-12	Table Manager shall support updating default table values from within the tool.	Select Table->Update Definition	table_manager.ahk
TBL-13	Table Manager shall support one-dimensional tables.	Edit a one-dimensional table.	table_manager.ahk
TBL-14	Table Manager shall support two-dimensional tables.	Edit a two-dimensional table.	table_manager.ahk

Handbook Creator

Handbook Creator provides a simple method to convert COSMOS command and telemetry definitions into beautiful Command and Telemetry Handbooks.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE

HC-1	Handbook Creator shall output handbooks in HTML format.	Press the Create HTML Handbooks button.	handbook_creator.ahk
HC-2	Handbook Creator shall output handbooks in PDF format.	Press the Create PDF Handbooks button.	handbook_creator.ahk
HC-3	Handbook Creator shall support outputing both HTML and PDF handbooks with one click.	Press the Create HTML and PDF Handbooks button.	handbook_creator.ahk
HC-4	Handbook Creator shall support opening the HTML handbooks in a web browser.	Press the Open in Web Browser button.	handbook_creator.ahk

Launcher

Launcher provides a utility to organize all applicable applications for a project and to launch those applications.

REQT. ID	DESCRIPTION	TEST DESCRIPTION	TEST TRACE
L-1	Launcher shall display a label and icon for each application to launch.	View Launcher.	launcher.ahk
L-2	Launcher shall start the requested application when clicking on its icon.	Launch a program from Launcher.	launcher.ahk
L-3	Launcher shall support launching multiple applications with one click.	Press the COSMOS button.	launcher.ahk
L-4	Launcher shall CRC check the COSMOS core and project files on startup.	View the CRC results on startup.	launcher.ahk
L-5	Launcher shall display a legal dialog on startup.	View the Legal dialog on startup.	launcher.ahk

Contributing

Improve this page

So you've got an awesome idea to throw into COSMOS. Great! Hers is the basic process:

1. Fork the project on Github
2. Create a feature branch
3. Make your changes
4. Submit a pull request

Don't Forget the Contributor License Agreement!
Before any contributions can be incorporated we do require all contributors to sign a Contributor License Agreement here: Contributor License Agreement This protects both you and us and you retain full rights to any code you write.

Test Dependencies

To run the test suite and build the gem you'll need to install COSMOS's dependencies. COSMOS uses Bundler, so a quick run of the `bundle` command and you're all set!

```
$ bundle
```

Before you start, run the tests and make sure that they pass (to confirm your environment is configured properly):

```
$ bundle exec rake build spec
```

Workflow

Here's the most direct way to get your work merged into the project:

- Fork the project.
- Clone down your fork:

```
git clone git://github.com/<username>/COSMOS.git
```

- Create a topic branch to contain your change:

```
git checkout -b my_awesome_feature
```

- Hack away, add tests. Not necessarily in that order.
- Make sure everything still passes by running `bundle exec rake`.
- If necessary, rebase your commits into logical chunks, without errors.

- Push the branch up:

```
git push origin my_awesome_feature
```

- Create a pull request against BallAerospace/COSMOS:master and describe what your change does and the why you think it should be merged.

Let us know what could be better!
Both using and hacking on COSMOS should be fun, simple, and easy, so if for some reason you find it's a pain, please create an issue on GitHub describing your experience so we can make it better.

BACK NEXT

Contact Us

Improve this page

Business Development and Support Contracts

Mike Lammertin
mlammert@ball.com

Phil Inslee
pinslee@ball.com

Technical Leads

Ryan Melton
rmelton@ball.com

Jason Thomas
jmthomas@ball.com

Release History

Improve this page

3.5.0 / 2015-06-22

This release contains a lot of new functionality and a key new feature: The ability to create new COSMOS targets and tools as reusable gems! This will hopefully allow the open source community to create sharable configuration for a large amount of hardware and allow for community generated tools to be easily integrated.

Bug Fixes:

- #153 set_tlm should support settings strings with spaces using the normal syntax
- #155 Default to not performing DNS lookups

New Features:

- #25 Warn users if reading a packet log uses the latest instead of the version specified in the file header
- #106 Allow the server to run headless
- #109 Cmd value api
- #129 Script Runner doesn't syntax highlight module namespacing
- #133 Add sound to COSMOS alerts
- #138 Limits Monitor should show what is stale
- #142 Support gem based targets and tools
- #144 Never have nothing happen when trying to launch a tool
- #152 Provide a method to retrieve current suite/group/case in TestRunner
- #157 Launcher support command line options in combobox
- #163 Allow message_box to display buttons vertically

Maintenance:

- #131 Consolidate Find/Replace logic in the FindReplaceDialog
- #137 Improve Server message log performance
- #142 Improve Windows Installer bat file
- #146 Need support for additional non-standard serial baud rates
- #150 Improve Win32 serial driver performance

Migration Notes from COSMOS 3.4.2:

The launcher scripts and .bat files that live in the COSMOS project tools folder have been updated to be easier to maintain and to ensure that the user always sees some sort of error message if a problem occurs starting a tool. All users should copy the new files from the tools folder in the COSMOS demo folder into their projects as part of the upgrade to COSMOS 3.5.0

COSMOS now disables reverse DNS lookups by default because they can take a long time in some environments. If you still want to see hostnames when someone connects to a TCP/IP server interface/router then you will need to add ENABLE_DNS to your system.txt file.

3.4.2 / 2015-05-08

Issues:

- #123 TestRunner command line option to launch a test automatically
- #125 Fix COSMOS issues for qtbindings 4.8.6.2
- #126 COSMOS GUI Chooser updates

Migration Notes from COSMOS 3.3.x or 3.4.x:

COSMOS 3.4.2 requires qtbindings 4.8.6.2. You must also update qtbindings when installing this release. Also note that earlier versions of COSMOS will not work with qtbindings 4.8.6.2. All users are strongly recommended to update both gems.

3.4.1 / 2015-05-01

Issues:

- #121 BinaryAccessor write crashes with negative bit sizes

Migration Notes from COSMOS 3.3.x:

None

Note: COSMOS 3.4.0 has a serious regression when writing to variably sized packets. Please upgrade to 3.4.1 immediately if you are using 3.4.0.

3.4.0 / 2015-04-27

Issues:

- #23 Handbook Creator User's Guide Mode
- #72 Refactor binary_accessor
- #101 Support Ruby 2.2 and 64-bit Ruby on Windows
- #104 CmdTlmServer Loading Tmp & SVN Conflict Files
- #107 Remove truthy and falsey from specs
- #110 Optimize TlmGrapher
- #111 Protect Interface Thread Stop from AutoReconnect
- #114 Refactor Cosmos::Script module
- #118 Allow PacketViewer to hide ignored items

Migration Notes from COSMOS 3.3.x:

None

3.3.3 / 2015-03-23

Issues:

- #93 Derived items that return arrays are not formatted to strings bug
- #94 JsonDRb retry if first attempt hits a closed socket bug

- #96 Make max lines written to output a variable in ScriptRunnerFrame enhancement
- #99 Increase Block Count in DataViewer

Migration Notes from COSMOS 3.2.x:

System.telemetry.target_names and System.commands.target_names no longer contain the 'UNKNOWN' target.

3.3.1 / 2015-03-19

COSMOS first-time startup speed is now 16 times faster - hence this release is codenamed "Startup Cheetah". Enjoy!

Issues:

- #91 Add mutex around creation of System.instance
- #89 Reduce maximum block count from 10000 to 100 everywhere
- #87 MACRO doesn't support more than one item
- #85 Replace use of DL with Fiddle
- #82 Improve COSMOS startup speed
- #81 UNKNOWN target identifies all buffers before other targets have a chance
- #78 Reduce COSMOS memory usage
- #76 Fix specs to new expect syntax and remove 'should'
- #74 Server requests/sec and utilization are incorrect

Migration Notes from COSMOS 3.2.x:

System.telemetry.target_names and System.commands.target_names no longer contain the 'UNKNOWN' target.

3.2.1 / 2015-02-23

Issues:

- #61 Don't crash TestRunner if there is an error during require_utilities()
- #63 Creating interfaces with the same name does not cause an error
- #64 Launcher RUBYW substitution broken by refactor
- #65 CmdTlmServer ensure log messages start scrolled to bottom on Linux
- #66 Improve graceful shutdown on linux and prevent continuous exceptions from InterfaceThread
- #70 ask() should take a default

Migration Notes from COSMOS 3.1.x:

No significant updates to existing code should be needed. The primary reason for update to 3.2.x is fixing the slow shutdown present in all of 3.1.x.

3.2.0 / 2015-02-17

Issues:

- #34 Refactor packet_config
- #43 Add ccsds_log_reader.rb as an example of alternative log readers
- #45 Slow shutdown of CTS and TlmViewer with threads trying to connect
- #46 Add mutex protection to Cosmos::MessageLog
- #47 TlmGrapher RangeError in Overview Graph

- #49 Make about dialog scroll
- #55 Automatic require of stream_protocol fix and cleanup
- #57 Add OPTION keyword to support passing arbitrary options to interfaces/routers
- #59 Add password mode to ask and ask_string

Migration Notes from COSMOS 3.1.x:

No significant updates to existing code should be needed. The primary reason for update to 3.2.x is fixing the slow shutdown present in all of 3.1.x.

3.1.2 / 2015-02-03

Issues:

- #20 Handbook Creator should output relative paths
- #21 Improve code metrics
- #26 Dynamically created file for Mac launchers should not be included in CRC calculation
- #27 TestRunner build_test_suites destroys CustomTestSuite if underlying test procedures change
- #28 TlmGrapher - Undefined method nan? for 0:Fixnum
- #35 Race condition starting new binary log
- #36 TlmDetailsDialog non-functional
- #37 Remaining TlmGrapher regression
- #38 Allow INTERFACE_TARGET to work with target name substitutions

Migration Notes from COSMOS 3.0.x:

The definition of limits persistence has changed. Before it only applied when changing to a bad state (yellow or red). Now persistence applies for all changes including from stale to a valid state and from bad states back to green.

3.1.1 / 2015-01-28

Issues:

- #10 Simulated Targets Button only works on Windows
- #11 Mac application folders not working
- #12 Persistence should be applied even if changing from stale
- #14 Allow information on logging page to be copied
- #16 Ensure read conversion cache cannot be cleared mid-use
- #17 NaNs in telemetry graph causes scaling crash

Migration Notes from COSMOS 3.0.x:

The definition of limits persistence has changed. Before it only applied when changing to a bad state (yellow or red). Now persistence applies for all changes including from stale to a valid state and from bad states back to green.

3.0.1 / 2015-01-06

First Announced Open Source Release