



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.

[Click here for a PDF version of this webpage](#)

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.

 BACK

NEXT 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the [Jekyll](#) project documentation which is licensed under the terms of the MIT License.

Custom Search



Proudly hosted by 



Navigate the docs... ▾

Installation

Improve this page

Installing COSMOS

The following sections describe howto get COSMOS installed on various operating systems.

Windows 7+

Run the COSMOS Installation bat file:

1. Right click this link and choose "Save Target As" or "Save Link As": [INSTALL_COSMOS.bat](#)
2. Save the file to your harddrive
3. Run the bat from Windows explorer or a cmd window



NEW - Windows 7: Powershell 4.0 Required

Most websites now require TLS 1.2 for downloads. The version of powershell included by default with Windows 7 does not have support for TLS 1.2 which causes downloads to fail. Before running the COSMOS installer bat file, you must install Powershell 4 which is part of Windows Management Framework 4 which can be found here: [Windows Management Framework 4](#)



SSL Issues

The COSMOS install scripts use command line tools like curl to download the code necessary for COSMOS across https connections. Increasingly organizations are using some sort of SSL decryptor device which can cause curl and other command line tools like git to have SSL certificate problems. If installation fails with messages that involve "certificate", "SSL", "self-signed", or "secure" this is the problem. IT typically sets up browsers to work correctly but not command line applications. Note that the file extension might not be .pem, it could be .pem, crt, .ca-bundle, .cer, .p7b, .p7s, or potentially something else.

The workaround is to get a proper local certificate file from your IT department that can be used by tools like curl (for example mine is at C:\Shared\Ball.pem). Doesn't matter just somewhere with no spaces.

Then set the following environment variables to that path (ie. C:\Shared\Ball.pem)

```
SSL_CERT_FILE  
CURL_CA_BUNDLE  
REQUESTS_CA_BUNDLE
```

Here are some directions on environment variables in Windows: [Windows Environment Variables](#)
You will need to create new ones with the names above and set their value to the full path to the certificate file.

After these changes the installer should work. At Ball please contact COSMOS@ball.com for assistance.



Offline Installation

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.



Note on Internet Explorer

If you left click the link above and try to save it, IE will corrupt the bat file. Don't download using Internet Explorer.

CentOS Linux 6.5/6.6/7, Ubuntu Linux 14.04LTS, and Mac OSX Mavericks+

The following instructions work for an installation on CentOS Linux 6.5, 6.6, or 7, and Ubuntu 14.04LTS from a clean install or any version of Mac OSX after and include Mavericks. Similar steps should work on other distributions/versions, particularly Redhat.

Run the following command in a terminal running the `bash` shell:

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



Issues with http_proxy

If you are using the `http_proxy` environment variable to use a proxy server, you **MUST** also have a `no_proxy` variable that includes `127.0.0.1` for COSMOS to work. Note that `127.0.0.0/8` in a `no_proxy` variable does not work with COSMOS. It must contain exactly `127.0.0.1`

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.

If installing in an environment where SSL Certificates are not setup correctly. The following commands will let COSMOS install in an insecure fashion:

```
echo "insecure" >> ~/.curlrc
export RUBY_BUILD_CURL_OPTS="-k"
git config --global http.sslVerify false
bash <(\curl -sSL https://raw.githubusercontent.com/BallAerospace/COSMOS/master/vendor/installers/lin
```

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 its configuration files:

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

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

Alternative Install directions using Mac Ports:

```
sudo port install qt4-mac
sudo port install ruby24
sudo port select --set ruby ruby24
sudo gem install ruby-termios
sudo gem install cosmos
```

You should have a working install and can now create a sample COSMOS directory structure where you can configure and run cosmos. To create a test area do this:

```
cosmos demo test
cd test
ruby Launcher
```

General

Notes:

1. The bash shell is required on Linux and Mac
2. 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`
3. The http_proxy environment variable can cause problems. Make sure you also have a no_proxy variable for localhost, something like no_proxy="127.0.0.1,localhost".

◀ BACK

NEXT ▶

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

Upgrading and the Gemfile

Improve this page

Upgrading COSMOS to the latest version is easy. Every COSMOS project comes with what is called a Gemfile. The Gemfile is used to track the library dependencies of your COSMOS project. If your project requires a gem that is not a standard COSMOS dependency, be sure to add it to your Gemfile to track that it is a requirement of your project. In any case, to upgrade to the latest version of COSMOS all you need to do is run:

```
bundle update cosmos
```

And that should get the latest version installed (unless your Gemfile has locked COSMOS to a specific version). After upgrading, you should also look at the [COSMOS release notes](#) to see if any other migration is required. If you would like to lock COSMOS (or any other gem) to a specific version, you can also do that with your Gemfile. Here is an example Gemfile that locks COSMOS to version 3.6.0, shows the ruby-termios gem requirement on non-windows systems, and also adds a project specific requirement for the sshkit gem.

```
gem 'cosmos', '3.6.0'  
gem 'ruby-termios' if RbConfig::CONFIG['target_os'] !~ /mswin|mingw|cygwin/i  
gem 'sshkit'
```

Finally, whenever you receive a new COSMOS project, running the following command will ensure you have everything you need installed to run that configuration.

```
bundle install
```

For more information on Gemfiles and managing dependencies for a COSMOS (or any other Ruby-based) project see the bundler documentation at: [Bundler Docs](#)

◀ BACK

NEXT ▶



Navigate the docs... ▾

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
├── config
│   ├── dart (optional)
│   ├── data
│   ├── system
│   └── targets
│       ├── TARGET
│       │   ├── cmd_tlm
│       │   ├── lib
│       │   ├── procedures
│       │   ├── screens
│       │   ├── sequences
│       │   ├── tables
│       │   └── tools
│       │       ├── table_manager
│       │       └── ...
│       └── cmd_tlm_server.txt
│   └── target.txt
└── ...
└── tools
    ├── cmd_tlm_server
    └── ...
└── lib
└── outputs
    ├── dart (optional)
    ├── handbooks
    ├── logs
    ├── saved_config
    ├── sequences
    ├── tables
    └── tmp
└── procedures
└── tools
    ├── mac
    └── ...

```

An overview of what each of these does:

FILE / DIRECTORY	DESCRIPTION
Gemfile	Defines the gems and their versions used by your COSMOS configuration. If you want to use other gems in your cosmos project you should add them here and then run "bundle install" from the command line. See the Bundler documents and our Upgrading section 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.
	The Rakefile contains user modifiable scripts to perform common tasks using the ruby rake

Rakefile	application. By default it comes with a script that is used to calculate CRCs over the project's files. rake crc
config	The config folder contains all of the configuration necessary for a COSMOS project.
config/ dart	The config/dart folder contains the DART Gemfile along with any other configuration files needed by the DART application.
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 your 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/ TARGET	config/targets/TARGET 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/ TARGET/ cmd_tlm	config/targets/TARGET/cmd_tlm contains command and telemetry definition files for the target. See Command and Telemetry Configuration for more information.
config/ targets/ TARGET/ lib	config/targets/TARGET/lib contains any custom code required by the target. Often this includes a custom Interface class. See Interfaces for more information.
config/ targets/ TARGET/ procedures	config/targets/TARGET/procedures contains target specific procedures which exercise functionality of the target. These procedures should be kept simple and only use the command and telemetry definitions associated with this target. See the Scripting Guide for more information.
config/ targets/ TARGET/ screens	config/targets/TARGET/screens contains telemetry screens for the target. See Screen Configuration for more information.
config/ targets/ TARGET/ sequences	config/targets/TARGET/sequences contains command sequences for the target. These are specific files used by the COSMOS Command Sequence tool.
config/ targets/ TARGET/ tables	config/targets/TARGET/tables contains binary tables for the target. These are specific files used by the COSMOS Table Manager tool.
config/ targets/ tools	config/targets/TARGET/tools contains target specific configuration files for the COSMOS applications. Most tools support configuration but do not require it.
config/ targets/ TARGET/ cmd_tlm_server.txt	config/targets/TARGET/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/ TARGET/ target.txt	config/targets/TARGET/target.txt contains target specific configuration such as which command parameters should be ignored by Command Sender. See Target 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 contains the configuration file for the COSMOS Command and

config/ tools/ cmd_tlm_server	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. This is also the place to override default COSMOS functionality. To do this you need to mirror the COSMOS directly structure and naming. First create a 'cosmos' directory and then add sub-folders and files which mirror the COSMOS source .
outputs	The outputs folder contains all files generated by COSMOS applications.
outputs/ dart	The outputs/dart folder contains the data and logs directories which DART uses to log the incoming command and telemetry data and log DART application status.
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. If you are running in production and collecting results these configurations should be saved (configuration managed) so you can re-parse the binary log data collected.
outputs/ sequences	The outputs/tables folder contains sequence files generated by Command Sequence.
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. See the Scripting Guide for more information.
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** 



Navigate the docs... ▾

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](#)

- 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”. Note the first packet defined without any ID_ITEMS is a “catch-all” packet that matches all incoming data (even if the data lengths don’t match).
 - 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



Navigate the docs... ▾

✍ Improve this page

- System Configuration
 - [AUTO_DECLARE_TARGETS](#)
 - [DECLARE_TARGET](#)
 - [DECLARE_GEM_TARGET](#)
 - [PORT](#)
 - [LISTEN_HOST](#)
 - [CONNECT_HOST](#)
 - [PATH](#)
 - [DEFAULT_PACKET_LOG_WRITER](#)
 - [DEFAULT_PACKET_LOG_READER](#)
 - [STALENESS_SECONDS](#)
 - [ENABLE_DNS](#)
 - [DISABLE_DNS](#)
 - [ENABLE_SOUND](#)
 - [ALLOW_ACCESS](#)
 - [META_INIT](#)
 - [TIME_ZONE_UTC](#)
 - [ADD_MD5_FILE](#)
- Target Configuration
 - [REQUIRE](#)
 - [IGNORE_PARAMETER](#)
 - [IGNORE_ITEM](#)
 - [COMMANDS](#)
 - [TELEMETRY](#)
 - [AUTO_SCREEN_SUBSTITUTE](#)
- Command and Telemetry Server Configuration
 - [TITLE](#)
 - [PACKET_LOG_WRITER](#)
 - [AUTO_INTERFACE_TARGETS](#)
 - [INTERFACE_TARGET](#)
 - [INTERFACE](#)
 - [ROUTER](#)

- COLLECT_METADATA
- BACKGROUND_TASK
 - INTERFACE Modifiers
 - TARGET
 - DONT_CONNECT
 - DONT_RECONNECT
 - RECONNECT_DELAY
 - DISABLE_DISCONNECT
 - LOG
 - DONT_LOG
 - LOG_RAW
 - PROTOCOL
 - OPTION
 - ROUTER Modifiers
 - ROUTE
 - TARGET
 - DONT_CONNECT
 - DONT_RECONNECT
 - RECONNECT_DELAY
 - DISABLE_DISCONNECT
 - LOG
 - DONT_LOG
 - LOG_RAW
 - PROTOCOL
 - OPTION
 - BACKGROUND_TASK Modifiers
 - STOPPED
- Project CRC Checking

System Configuration

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.

AUTO_DECLARE_TARGETS

Automatically load all the target folders under config/targets into the system

When automatically discovering target folders the COSMOS naming convention must be followed. 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. This keyword is REQUIRED unless you individually declare your targets using the DECLARE_TARGET keyword.

DECLARE_TARGET

Declare a COSMOS target and name it

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. Valid Values: Any Target Name	True
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.	False
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.	False

Example Usage:

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

DECLARE_GEM_TARGET

Declare a COSMOS target which resides in a Ruby gem

COSMOS targets can be distributed in gem form in which case they must be named 'cosmos-' where is the descriptive name of the gem. See the COSMOS documentation at cosmosrb.com for information on creating a COSMOS gem target.

PARAMETER	DESCRIPTION	REQUIRED
Gem Name	The COSMOS gem name which must be cosmos- where is the descriptive gem name. In the above example, the target will be named 'GEM-NAME' (note the cosmos- prefix is stripped off).	True
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 gem name as described above.	False

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.	False
--------------------	--	-------

PORT

Set a Server 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	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 as it also accepts commands. CTS_CMD_ROUTER - This port provides access to a stream of all command packets that have been accepted by COSMOS. Valid Values: CTS_API , TLMVIEWER_API , CTS_PREIDENTIFIED , CTS_CMD_ROUTER	True
Port Value	Port number to use for the specified port name	True

Example Usage:

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

LISTEN_HOST

IP addresses or host names to bind API ports to

(Since 4.0.0)

PARAMETER	DESCRIPTION	REQUIRED
Host Port Name	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 as it also accepts commands. CTS_CMD_ROUTER - This port provides access to a stream of all command packets that have been accepted by COSMOS. Valid Values: CTS_API , TLMVIEWER_API , CTS_PREIDENTIFIED , CTS_CMD_ROUTER	True
Host Value	The IP address or host name to bind the Host Port Name to. By default, the CTS_API and TLMVIEWER_API are bound to localhost meaning that only tools on the same machine can connect. By default, CTS_PREIDENTIFIED and CTS_CMD_ROUTER are bound to '0.0.0.0' to allow for tools on any machines to connect. This allows for chaining two or more Servers out of the box.	True

CONNECT_HOST

IP addresses or host names to connect tools to

(Since 4.0.0)

PARAMETER	DESCRIPTION	REQUIRED
	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 as it also accepts commands. CTS_CMD_ROUTER - This port provides access to a stream of all command packets that have been accepted by COSMOS. Valid Values: CTS_API , TLMVIEWER_API , CTS_PREIDENTIFIED , CTS_CMD_ROUTER	
Host Port Name	The IP address or host name to connect tools to. By default all ports are bound to localhost so all tools connect to the local machine.	True
Host Value		True

PATH

Set a Server 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).

!! 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.

PARAMETER	DESCRIPTION	REQUIRED
Path Name	Path name to set. Valid Values: LOGS , TMP , SAVED_CONFIG , TABLES , PROCEDURES , HANDBOOK	True
Path Value	File system path to use for the specified path name	True

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
```

DEFAULT_PACKET_LOG_WRITER

Set the class used when creating binary packet log files

Overriding the default log writer can break the ability to write log files that COSMOS can interpret. Proceed with caution!



PARAMETER	DESCRIPTION	REQUIRED
Filename	Ruby file to use when instantiating a new log writer	True
parameter	Parameters which are passed to the log read upon initialization	False

Example Usage:

```
DEFAULT_PACKET_LOG_WRITER packet_log_writer.rb
```

DEFAULT_PACKET_LOG_READER

Set the class used when reading binary packet log file



Overriding the default log read can break the ability to read log files. Proceed with caution!

PARAMETER	DESCRIPTION	REQUIRED
Filename	Ruby file to use when instantiating a new log writer	True
parameter	Parameters which are passed to the log read upon initialization	False

Example Usage:

```
DEFAULT_PACKET_LOG_READER packet_log_reader.rb
```

STALENESS_SECONDS

Number of seconds before marking the packet stale

A stale packet is identified in telemetry screens by all the telemetry items in the stale packet being colored purple..

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

Example Usage:

```
STALENESS_SECONDS 30
```

ENABLE_DNS

Enable reverse DNS lookups for tools

(Since 3.5.0)

Enables reverse DNS lookups 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.

DISABLE_DNS

Disable reverse DNS lookups for tools

Disable reverse DNS lookups 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

ENABLE_SOUND

Enable audible sounds when popups occur

(Since 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.

ALLOW_ACCESS

White list machines that are allowed to connect to the Server

PARAMETER	DESCRIPTION	REQUIRED
Name or IP Address	Machine name to allow access or you can specify 'ALL' to allow all machines access	True

Example Usage:

```
ALLOW_ACCESS ALL
```

META_INIT

Specify a file to initialize the SYSTEM META packet

(Since 4.0.0)

Filename should be a text file with key value pairs where the keyword matches an item in the SYSTEM META packet. Note you do not have to specify ALL the items in the SYSTEM META packet.

PARAMETER	DESCRIPTION	REQUIRED
Filename	Filename, either fully qualified, or relative to Cosmos::USERPATH	True

TIME_ZONE_UTC

Report all times as UTC

(Since 3.10.0)

COSMOS will report all times as UTC time. If this keyword is not used, COSMOS will report all times as local times, where the local time zone is determined automatically by Ruby based upon the operating system time settings. This setting affects packet receive times, timestamped log filenames, message logs, Cmd/Tlm extractor time ranges, etc.

ADD_MD5_FILE

Add a file to the MD5 sum calculation

(Since 4.0.0)

Adds a file to the set of files used in marshal file MD5 sum calculation. Upon startup, COSMOS calculates an MD5 sum over the command/telemetry definition files for all targets. After the definitions have been processed, COSMOS saves the resulting objects as marshal files in a folder with the MD5 sum as part of the name. The next time COSMOS runs, if the MD5 checksum of the cmd/tlm definition files has not changed, COSMOS can load the marshal files instead of re-processing the definitions. If a file is specified with the ADD_MD5_FILE keyword, COSMOS will include it in the MD5 sum calculation. This means that a change in the file will cause COSMOS to re-process the cmd/tlm definitions and create a new set of marshal files.

PARAMETER	DESCRIPTION	REQUIRED
Filename	Filename, either fully qualified, or relative to Cosmos::USERPATH	True

Example Usage:

```
ADD_MD5_FILE lib/user_version.rb
```

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.

REQUIRE

Requires a Ruby file

Ruby files must be required to be available to call in other code. Files are first required from the target's lib folder. If no file is found the Ruby system path is checked which includes the base COSMOS/lib folder.

PARAMETER	DESCRIPTION	REQUIRED
Filename	Filename to require. For files in the target's lib directory simply supply the filename, e.g. "REQUIRE my_file". Files in the base COSMOS lib directory also should just list the filename. If a file is in a folder under the lib directory then you must specify the folder name, e.g. "REQUIRE folder/my_file". The filename can also be an absolute path but this is not common. Note the ".rb" extension is optional when specifying the filename.	True

Example Usage:

```
REQUIRE limits_response.rb
```

IGNORE_PARAMETER

Ignore the given command parameter

Hint to other COSMOS tools to hide or ignore this command parameter when processing the command. For example, Command Sender and Command Sequence will not display the parameter (by default) when showing the command and Script Runner code completion will not display the parameter.

PARAMETER	DESCRIPTION	REQUIRED
Parameter Name	The name of a command parameter. Note that this parameter will be ignored in ALL the commands it appears in.	True

Example Usage:

```
IGNORE_PARAMETER CCSDS_VERSION
```

IGNORE_ITEM

Ignore the given telemetry item

Hint to other COSMOS tools to hide or ignore this telemetry item when processing the telemetry. For example, Packet

Viewer will not display the item (by default) when showing the packet.

PARAMETER	DESCRIPTION	REQUIRED
Item name	The name of a telemetry item. Note that this item will be ignored in ALL the telemetry it appears in.	True

Example Usage:

```
IGNORE_ITEM CCSDS_VERSION
```

COMMANDS

Process the given command definition file

This keyword is used to explicitly add the command definition file to the list of command and telemetry files to process.

 Usage of this keyword overrides automatic command and telemetry file discovery. If this keyword is used, you must also use the TELEMETRY keyword to specify the telemetry files to process.

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of a command definition file in the target's cmd_tlm directory, e.g. "cmd.txt".	True

Example Usage:

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

TELEMETRY

Process the given telemetry definition file

This keyword is used to explicitly add the telemetry definition file to the list of command and telemetry files to process.

 Usage of this keyword overrides automatic command and telemetry file discovery. If this keyword is used, you must also use the COMMAND keyword to specify the command files to process.

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of a telemetry definition file in the target's cmd_tlm directory, e.g. "tlm.txt".	True

Example Usage:

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

AUTO_SCREEN_SUBSTITUTE

Automatically substitute the target's name in screen definitions

Targets can be renamed when they are declared which would break any screen definitions using the explicit target name. This keyword automatically replaces the target name in the screen definitions with the actual target name.

!! Replaces ALL target names in a screen definition file, so this is not suitable for screens with multiple targets.

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.

TITLE

Sets the Command and Telemetry Server window title

PARAMETER	DESCRIPTION	REQUIRED
Text	Text to put in the title of the Command and Telemetry Server window	True

PACKET_LOG_WRITER

Declare a 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.

!! You should NOT override the default 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.	True
Filename	Ruby file to use when instantiating a new log writer	True
Parameters	Optional parameters to pass to the log writer class when instantiating it.	False

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 (600s)
PACKET_LOG_WRITER COSMOS_LOG packet_log_writer.rb cosmos true 600
```

AUTO_INTERFACE_TARGETS

Automatically use each target's cmd_tlm_server.txt file to define the interface

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.

INTERFACE_TARGET

Load the specified target's cmd_tlm_server.txt configuration file

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target Valid Values: <code>Any Target Name</code>	True
Configuration File	Configuration file name which contains the interface configuration. Defaults to 'cmd_tlm_server.txt'.	False

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

Defines a connection to a physical target

Interfaces are what COSMOS uses to talk to a particular piece of hardware. Interfaces require a Ruby file which implements all the interface methods necessary to talk to the hardware. COSMOS defines many built in interfaces or you can define your own as long as it implements the interface protocol.

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. The COSMOS convention is to name interfaces after their targets with '_INT' appended to the name, e.g. INST_INT for the INST target.	True
Filename	Ruby file to use when instantiating the interface. Valid Values: <code>tcpip_client_interface.rb</code> , <code>tcpip_server_interface.rb</code> , <code>udp_interface.rb</code> , <code>serial_interface.rb</code> , <code>cmd_tlm_server_interface.rb</code> , <code>linc_interface.rb</code>	True

Additional parameters are required. Please see the [Interfaces](#) documentation for more details.

ROUTER

Create an interface which reverses cmd/tlm data

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	True

	Ruby file to use when instantiating the interface.	
Filename	Valid Values: <code>tcpip_client_interface.rb</code> , <code>tcpip_server_interface.rb</code> , <code>udp_interface.rb</code> , <code>serial_interface.rb</code> , <code>cmd_tlm_server_interface.rb</code> , <code>linc_interface.rb</code>	True

Additional parameters are required. Please see the [Interfaces](#) documentation for more details.

COLLECT_METADATA

Prompts the user for meta data when starting the Command and Telemetry Server

BACKGROUND_TASK

Create a background task in the Command and Telemetry 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.	True
Optional Arguments	Optional arguments to the background task constructor	False

Example Usage:

```
BACKGROUND_TASK example_background_task.rb
```

INTERFACE Modifiers

The following keywords must follow a INTERFACE keyword.

TARGET

Maps a target name to an interface

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Target name to map to this interface Valid Values: Any Target Name	True

DONT_CONNECT

Server will not automatically try to connect to the interface at startup

DONT_RECONNECT

Server will not try to reconnect to the interface if the connection is lost

RECONNECT_DELAY

Reconnect delay in seconds

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

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

DISABLE_DISCONNECT

Disable the Disconnect button on the Interfaces tab in the Server

Use this keyword to prevent the user from disconnecting from the interface. This is typically used in a ‘production’ environment where you would not want the user to inadvertently disconnect from a target.

LOG

Enable logging on the interface by the specified log writer

LOG is only required if you want a log writer other than the default to log commands and telemetry on this interface



Choosing a custom log writer can prevent COSMOS from reading back your log files

PARAMETER	DESCRIPTION	REQUIRED
Name	Log writer name as defined by PACKET_LOG_WRITER	True

DONT_LOG

Disable logging commands and telemetry on this interface

LOG_RAW

Log all data on the interface exactly as it is sent and received

LOG_RAW 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. You will have to manually parse these logs yourself using a hex editor or other application.

PROTOCOL

Protocols modify the interface by processing the data

(Since 4.0.0)

Protocols can be either READ, WRITE, or READ_WRITE. READ protocols act on the data received by the interface while write acts on the data before it is sent out. READ_WRITE applies the protocol to both reading and writing.

There is only one built in protocol implemented by override_protocol.rb. This protocol allows for Scripts to use the override_tlm() and normalize_tlm() methods to permanently change a telemetry value. Note, this differs from set_tlm() as set_tlm() is over-written by new incoming telemetry.

For information on creating your own custom protocol please see cosmosrb.com/docs/protocols

PARAMETER	DESCRIPTION	REQUIRED

Type	Whether to apply the protocol on incoming data, outgoing data, or both Valid Values: READ , WRITE , READ_WRITE	True
Protocol Filename or Classname	Ruby file name or class name which implements the protocol	True
Protocol specific parameters	Additional parameters used by the protocol	False

OPTION

Set a parameter on an interface

When an option is set the interface class calls the `set_option` method. Custom interfaces can override `set_option` to handle any additional options they want.

PARAMETER	DESCRIPTION	REQUIRED
	The option to set. COSMOS defines several options on the core provided interfaces. The SerialInterface defines FLOW_CONTROL which can be NONE (default) or RTSCTS and DATA_BITS which changes the data bits of the serial interface. The TcpipServerInterface defines LISTEN_ADDRESS which is the IP address to accept connections on (default 0.0.0.0) and AUTO_SYSTEM_META which will automatically send SYSTEM META when the interface connects (default false).	
Name		True
Parameters	Parameters to pass to the option	False

ROUTER Modifiers

The following keywords must follow a ROUTER keyword.

ROUTE

Map an interface to a router

Once an interface has been mapped to a router, all its received telemetry will be sent out through the router.

PARAMETER	DESCRIPTION	REQUIRED
Interface	Name of the interface	True

TARGET

Maps a target name to an interface

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Target name to map to this interface Valid Values: Any Target Name	True

DONT_CONNECT

Server will not automatically try to connect to the interface at startup

DONT_RECONNECT

Server will not try to reconnect to the interface if the connection is lost

RECONNECT_DELAY

Reconnect delay in seconds

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

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

DISABLE_DISCONNECT

Disable the Disconnect button on the Interfaces tab in the Server

Use this keyword to prevent the user from disconnecting from the interface. This is typically used in a ‘production’ environment where you would not want the user to inadvertently disconnect from a target.

LOG

Enable logging on the interface by the specified log writer

LOG is only required if you want a log writer other than the default to log commands and telemetry on this interface



Choosing a custom log writer can prevent COSMOS from reading back your log files

PARAMETER	DESCRIPTION	REQUIRED
Name	Log writer name as defined by PACKET_LOG_WRITER	True

DONT_LOG

Disable logging commands and telemetry on this interface

LOG_RAW

Log all data on the interface exactly as it is sent and received

LOG_RAW 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. You will have to manually parse these logs yourself using a hex editor or other application.

PROTOCOL

Protocols modify the interface by processing the data

(Since 4.0.0)

Protocols can be either READ, WRITE, or READ_WRITE. READ protocols act on the data received by the interface while write acts on the data before it is sent out. READ_WRITE applies the protocol to both reading and writing.

There is only one built in protocol implemented by override_protocol.rb. This protocol allows for Scripts to use the override_tlm() and normalize_tlm() methods to permanently change a telemetry value. Note, this differs from set_tlm() as set_tlm() is over-written by new incoming telemetry.

For information on creating your own custom protocol please see cosmosrb.com/docs/protocols

PARAMETER	DESCRIPTION	REQUIRED

Type	Whether to apply the protocol on incoming data, outgoing data, or both Valid Values: <code>READ</code> , <code>WRITE</code> , <code>READ_WRITE</code>	True
Protocol Filename or Classname	Ruby file name or class name which implements the protocol	True
Protocol specific parameters	Additional parameters used by the protocol	False

OPTION

Set a parameter on an interface

When an option is set the interface class calls the `set_option` method. Custom interfaces can override `set_option` to handle any additional options they want.

PARAMETER	DESCRIPTION	REQUIRED
Name	The option to set. COSMOS defines several options on the core provided interfaces. The SerialInterface defines FLOW_CONTROL which can be <code>NONE</code> (default) or <code>RTSCTS</code> and <code>DATA_BITS</code> which changes the data bits of the serial interface. The TcpipServerInterface defines <code>LISTEN_ADDRESS</code> which is the IP address to accept connections on (default <code>0.0.0.0</code>) and <code>AUTO_SYSTEM_META</code> which will automatically send SYSTEM META when the interface connects (default false).	True
Parameters	Parameters to pass to the option	False

BACKGROUND_TASK Modifiers

The following keywords must follow a `BACKGROUND_TASK` keyword.

STOPPED

Indicate the background task should not be automatically started

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

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

✍ Improve this page

- [Command Definition Files](#)
- [COMMAND](#)
- [SELECT_COMMAND](#)
 - [COMMAND Modifiers](#)
 - [PARAMETER](#)
 - [APPEND_PARAMETER](#)
 - [ID_PARAMETER](#)
 - [APPEND_ID_PARAMETER](#)
 - [ARRAY_PARAMETER](#)
 - [APPEND_ARRAY_PARAMETER](#)
 - [SELECT_PARAMETER](#)
 - [HIDDEN](#)
 - [DISABLED](#)
 - [DISABLE_MESSAGES](#)
 - [META](#)
 - [HAZARDOUS](#)
 - [PARAMETER Modifiers](#)
 - [FORMAT_STRING](#)
 - [UNITS](#)
 - [DESCRIPTION](#)
 - [META](#)
 - [REQUIRED](#)
 - [MINIMUM_VALUE](#)
 - [MAXIMUM_VALUE](#)
 - [DEFAULT_VALUE](#)
 - [STATE](#)
 - [WRITE_CONVERSION](#)
 - [POLY_WRITE_CONVERSION](#)
 - [SEG_POLY_WRITE_CONVERSION](#)
 - [GENERIC_WRITE_CONVERSION_START](#)
 - [GENERIC_WRITE_CONVERSION_END](#)
 - [OVERFLOW](#)
- [Example File](#)

Command Configuration

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. The easiest way to do this is to add an extension to an existing file name. For example, if you already have cmd.txt you can create cmd_override.txt for telemetry that depends on the definitions in tlm.txt. Also note that due to the way the [ASCII Table](#) is structured, files beginning with capital letters are processed before lower case letters.

When defining command parameters you can choose from the following data types: INT, UINT, FLOAT, DERIVED, STRING, BLOCK. These correspond to integers, unsigned integers, floating point numbers, derived values of 0 size which aren't actually physically defined in the packet, strings and binary blocks of data. The only difference between a STRING and BLOCK is when COSMOS reads the binary command log it stops reading a STRING type when it encounters a null byte (0). This shows up in the text log produced by Command Extractor. Note that this does NOT affect the data COSMOS writes as it's still legal to pass null bytes (0) in STRING parameters.

COMMAND

Defines a new command packet

PARAMETER	DESCRIPTION	REQUIRED
Target	Name of the target this command is associated with Valid Values: Any Target Name	True
Command	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.	True
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	True
Description	Description of this command which must be enclosed with quotes	False

Example Usage:

```
COMMAND SYSTEM STARTLOGGING BIG_ENDIAN "Starts logging both commands and telemetry for an interface"
```

SELECT_COMMAND

Selects an existing command packet for editing

Typically used in a separate configuration file from where the original command is defined to override or add to the existing command definition

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target this command is associated with Valid Values: Any Target Name	True
Command Name	Name of the command to select	True

Example Usage:

SELECT_COMMAND SYSTEM STARTLOGGING

COMMAND Modifiers

The following keywords must follow a COMMAND keyword.

PARAMETER

Defines a command parameter in the current command packet

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the parameter. Must be unique within the command.	True
Bit Offset	Bit offset into the command packet of the Most Significant Bit of this parameter. May be negative to indicate an offset from the end of the packet. Always use a bit offset of 0 for derived parameters.	True
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 is 0 and Bit Size is 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	True
Data Type	Data Type of this parameter Valid Values: INT, UINT, FLOAT, DERIVED, STRING, BLOCK	True

When Data Type is INT, UINT, FLOAT, DERIVED the remaining parameters are:

Minimum Value	Minimum allowed value for this parameter	True
Maximum Value	Maximum allowed value for this parameter	True
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.	True
Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

When Data Type is STRING, BLOCK the remaining parameters are:

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.	True
Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

Example Usage:

```

PARAMETER SYNC 0 32 UINT 0xDEADBEEF 0xDEADBEEF 0xDEADBEEF "Sync pattern"
PARAMETER DATA 32 32 INT MIN MAX 0 "Data value"
PARAMETER VALUE 64 32 FLOAT 0 10.5 2.5
PARAMETER LABEL 96 96 STRING "COSMOS" "The label to apply"
PARAMETER BLOCK 192 0 BLOCK '' "Block of binary data"

```

APPEND_PARAMETER

Defines a command parameter in the current command packet

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the parameter. Must be unique within the command.	True
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 is 0 and Bit Size is 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	True
Data Type	Data Type of this parameter Valid Values: INT, UINT, FLOAT, DERIVED, STRING, BLOCK	True

When Data Type is INT, UINT, FLOAT, DERIVED the remaining parameters are:

Minimum Value	Minimum allowed value for this parameter	True
Maximum Value	Maximum allowed value for this parameter	True
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.	True
Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

When Data Type is STRING, BLOCK the remaining parameters are:

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.	True
Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

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

Defines an identification command parameter in the current command packet

ID parameters are used to identify the binary block of data as a particular command. A command packet may have one or more ID_PARAMETERS and all 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.	True
Bit Offset	Bit offset into the command packet of the Most Significant Bit of this parameter. May be negative to indicate an offset from the end of the packet. Always use a bit offset of 0 for derived parameters.	True
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 is 0 and Bit Size is 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	True
Data Type	Data Type of this parameter Valid Values: INT, UINT, FLOAT, DERIVED, STRING, BLOCK	True

When Data Type is INT, UINT, FLOAT, DERIVED the remaining parameters are:

Minimum Value	Minimum allowed value for this parameter	True
Maximum Value	Maximum allowed value for this parameter	True
ID Value	Identification value for this parameter. The binary data must match this value for the buffer to be identified as this packet.	True
Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

When Data Type is STRING, BLOCK the remaining parameters are:

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.	True
Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

Example Usage:

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

APPEND_ID_PARAMETER

Defines an identification command parameter in the current command packet

ID parameters are used to identify the binary block of data as a particular command. A command packet may have one or more ID_PARAMETERS and all 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.	True

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 is 0 and Bit Size is 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	True
Data Type	Data Type of this parameter Valid Values: INT, UINT, FLOAT, DERIVED, STRING, BLOCK	True

When Data Type is INT, UINT, FLOAT, DERIVED the remaining parameters are:

Minimum Value	Minimum allowed value for this parameter	True
Maximum Value	Maximum allowed value for this parameter	True
ID Value	Identification value for this parameter. The binary data must match this value for the buffer to be identified as this packet.	True
Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

When Data Type is STRING, BLOCK the remaining parameters are:

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.	True
Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

Example Usage:

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

ARRAY_PARAMETER

Defines a command parameter in the current command packet that is an array

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the parameter. Must be unique within the command.	True
Bit Offset	Bit offset into the command packet of the Most Significant Bit of this parameter. May be negative to indicate an offset from the end of the packet. Always use a bit offset of 0 for derived parameters.	True
Item Bit Size	Bit size of each array item	True
Item Data Type	Data Type of each array item Valid Values: INT, UINT, FLOAT, STRING, BLOCK, DERIVED	True

Array Bit Size	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.	True
Description	Description which must be enclosed with quotes	False
Endianness	Indicates if the data is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

Example Usage:

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

APPEND_ARRAY_PARAMETER

Defines a command parameter in the current command packet that is an array

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the parameter. Must be unique within the command.	True
Item Bit Size	Bit size of each array item	True
Item Data Type	Data Type of each array item Valid Values: INT, UINT, FLOAT, STRING, BLOCK, DERIVED	True
Array Bit Size	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.	True
Description	Description which must be enclosed with quotes	False
Endianness	Indicates if the data is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

Example Usage:

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

SELECT_PARAMETER

Selects an existing command parameter for editing

PARAMETER	DESCRIPTION	REQUIRED
Parameter	Name of the parameter to select for modification	True

Example Usage:

```
SELECT_COMMAND SYSTEM STARTLOGGING
SELECT_PARAMETER LABEL
```

HIDDEN

Hides this command from all COSMOS tools such as Command Sender and Handbook Creator

Hidden commands do not appear in the Script Runner popup helper when writing scripts. The command still exists in the system and can be sent by scripts.

DISABLED

Disables this command from being sent

Hides the command and also disables it from being sent by scripts. Attempts to send DISABLED commands result in an error message.

DISABLE_MESSAGES

Disable the Server from printing cmd(...) messages. Commands are still logged.

META

Stores metadata for the current command

Meta data is user specific data that can be used by custom tools for various purposes. One example is to store additional information needed to generate source code header files.

PARAMETER	DESCRIPTION	REQUIRED
Meta Name	Name of the metadata to store	True
Meta Values	One or more values to be stored for this Meta Name	False

Example Usage:

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

HAZARDOUS

Designates the current command as hazardous

Sending a hazardous command causes a dialog asking for confirmation before sending the command

PARAMETER	DESCRIPTION	REQUIRED
Description	Description for why the command is hazardous which must be enclosed with quotes	False

PARAMETER Modifiers

The following keywords must follow a PARAMETER keyword.

FORMAT_STRING

Adds printf style formatting

PARAMETER	DESCRIPTION	REQUIRED
Format	How to format using printf syntax. For example, '0x%0X' will display the value in hex.	True

Example Usage:

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

UNITS

Add displayed units

PARAMETER	DESCRIPTION	REQUIRED
Full Name	Full name of the units type, e.g. Celcius	True
Abbreviated	Abbreviation for the units, e.g. C	True

Example Usage:

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

DESCRIPTION

Override the defined description

PARAMETER	DESCRIPTION	REQUIRED
Value	The new description	True

META

Stores custom user metadata

Meta data is user specific data that can be used by custom tools for various purposes. One example is to store additional information needed to generate source code header files.

PARAMETER	DESCRIPTION	REQUIRED
Meta Name	Name of the metadata to store	True
Meta Values	One or more values to be stored for this Meta Name	False

Example Usage:

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

REQUIRED

Parameter is required to be populated in scripts

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.

MINIMUM_VALUE

Override the defined minimum value

PARAMETER	DESCRIPTION	REQUIRED
Value	The new minimum value for the parameter	True

MAXIMUM_VALUE

Override the defined maximum value

PARAMETER	DESCRIPTION	REQUIRED

Value	The new maximum value for the parameter	True
-------	---	------

DEFAULT_VALUE

Override the defined default value

PARAMETER	DESCRIPTION	REQUIRED
Value	The new default value for the parameter	True

STATE

Defines a key/value pair for the current command parameter

Key value pairs allow for user friendly strings. 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 string state name	True
Value	The numerical state value	True
Hazardous	Indicates the state is hazardous. This will cause a popup to ask for user confirmation when sending this command. Valid Values: HAZARDOUS	False
Hazardous Description	String describing why this state is hazardous	False

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

Applies a conversion when writing the current command parameter

Conversions are 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 REQUIRE). 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'.	True

Parameter	Additional parameter values for the conversion which are passed to the class constructor.	False
-----------	---	-------

Example Usage:

```
WRITE_CONVERSION the_great_conversion.rb 1000
```

Defined in the_great_conversion.rb:

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

POLY_WRITE_CONVERSION

Adds a polynomial conversion factor to the current command parameter

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
C0	Coefficient	True
Cx	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.	False

Example Usage:

```
POLY_WRITE_CONVERSION 10 0.5 0.25
```

SEG_POLY_WRITE_CONVERSION

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.	True
C0	Coefficient	True

Cx

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.

False

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

Start a generic write conversion

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.



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.

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
```

GENERIC_WRITE_CONVERSION_END

Complete a generic write conversion

OVERFLOW

Set the behavior when writing a value overflows the type

By default COSMOS throws an error if you try to write a value which overflows its specified type, e.g. writing 255 to a 8 bit signed value. Setting the overflow behavior also allows for COSMOS to 'TRUNCATE' the value by eliminating any high order bits. You can also set 'SATURATE' which causes COSMOS to replace the value with the maximum or minimum allowable value for that type. Finally you can specify 'ERROR_ALLOW_HEX' which will allow for a maximum hex value to be written, e.g. you can successfully write 255 to a 8 bit signed value.

PARAMETER	DESCRIPTION	REQUIRED
Behavior	How COSMOS treats an overflow value. Only applies to signed and unsigned integer data types. Valid Values: ERROR , ERROR_ALLOW_HEX , TRUNCATE , SATURATE	True

Example Usage:

Example File

Example File: <COSMOSPATH>/config/TARGET/cmd_tlm/cmd.txt

```

COMMAND 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 CCSDESTYPE 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 TARGET NOOP BIG_ENDIAN "Do Nothing"
PARAMETER CCSDSVER 0 3 UINT 0 0 0 "CCSDS PRIMARY HEADER VERSION NUMBER"
PARAMETER CCSDESTYPE 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 TARGET SETTINGS BIG_ENDIAN "Set the Settings"
PARAMETER CCSDSVER 0 3 UINT 0 0 0 "CCSDS PRIMARY HEADER VERSION NUMBER"
PARAMETER CCSDESTYPE 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"
<% 5.times do |x| %>
  APPEND_PARAMETER SETTING<%= x %> 16 UINT 0 5 0 "Setting <%= x %>"
<% end %>
```

BACK

NEXT



Navigate the docs... ▾

✍ Improve this page

- [Telemetry Definition Files](#)
- [TELEMETRY](#)
- [SELECT_TELEMETRY](#)
- [LIMITS_GROUP](#)
- [LIMITS_GROUP_ITEM](#)
 - [TELEMETRY Modifiers](#)
 - [ITEM](#)
 - [APPEND_ITEM](#)
 - [ID_ITEM](#)
 - [APPEND_ID_ITEM](#)
 - [ARRAY_ITEM](#)
 - [APPEND_ARRAY_ITEM](#)
 - [SELECT_ITEM](#)
 - [META](#)
 - [PROCESSOR](#)
 - [ALLOW_SHORT](#)
 - [HIDDEN](#)
 - [ITEM Modifiers](#)
 - [FORMAT_STRING](#)
 - [UNITS](#)
 - [DESCRIPTION](#)
 - [META](#)
 - [STATE](#)
 - [READ_CONVERSION](#)
 - [POLY_READ_CONVERSION](#)
 - [SEG_POLY_READ_CONVERSION](#)
 - [GENERIC_READ_CONVERSION_START](#)
 - [GENERIC_READ_CONVERSION_END](#)
 - [LIMITS](#)
 - [LIMITS_RESPONSE](#)
- [Example File](#)

Telemetry Configuration

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. The easiest way to do this is to add an extension to an existing file name. For example, if you already have tlm.txt you can create tlm_override.txt for telemetry that depends on the definitions in tlm.txt. Note that due to the way the [ASCII Table](#) is structured, files beginning with capital letters are processed before lower case letters.

When defining telemetry items you can choose from the following data types: INT, UINT, FLOAT, DERIVED, STRING, BLOCK. These correspond to integers, unsigned integers, floating point numbers, derived values of 0 size which aren't actually physically defined in the packet, strings and binary blocks of data. The only difference between a STRING and BLOCK is when COSMOS reads a STRING type it stops reading when it encounters a null byte (0). This shows up when displaying the value in Packet Viewer or Tlm Viewer and in the output of Telemetry Extractor.

TELEMETRY

Defines a new telemetry packet

PARAMETER	DESCRIPTION	REQUIRED
Target	Name of the target this telemetry packet is associated with Valid Values: Any Target Name	True
Command	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.	True
Endianness	Indicates if the data in this packet is in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	True
Description	Description of this telemetry packet which must be enclosed with quotes	False

Example Usage:

```
TELEMETRY SYSTEM LIMITS_CHANGE BIG_ENDIAN "COSMOS limits change"
```

SELECT_TELEMETRY

Selects an existing telemetry packet for editing

Typically used in a separate configuration file from where the original telemetry is defined to override or add to the existing telemetry definition

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target this telemetry packet is associated with Valid Values: Any Target Name	True
Command Name	Name of the telemetry packet to select	True

Example Usage:

```
SELECT_TELEMETRY SYSTEM LIMITS_CHANGE
```

LIMITS_GROUP

Defines a group of related limits Items

Limits groups contain telemetry items 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	True

LIMITS_GROUP_ITEM

Adds the specified telemetry item to the last defined LIMITS_GROUP

Limits group 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.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target	True
	Valid Values: Any Target Name	
Packet Name	Name of the packet	True
Item Name	Name of the telemetry item to add to the group	True

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
```

TELEMETRY Modifiers

The following keywords must follow a TELEMETRY keyword.

ITEM

Defines a telemetry item in the current telemetry packet

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the telemetry item. Must be unique within the packet.	True
Bit Offset	Bit offset into the telemetry packet of the Most Significant Bit of this item. May be negative to indicate an offset from the end of the packet. Always use a bit offset of 0 for derived item.	True
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 is 0 and Bit Size is 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	True

Data Type	Data Type of this telemetry item Valid Values: INT, UINT, FLOAT, STRING, BLOCK, DERIVED	True
Description	Description for this telemetry item which must be enclosed with quotes	False
Endianness	Indicates if the item is to be interpreted in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

Example Usage:

```
ITEM PKTID 112 16 UINT "Packet ID"
ITEM DATA 0 0 DERIVED "Derived data"
```

APPEND_ITEM

Defines a telemetry item in the current telemetry packet

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the telemetry item. Must be unique within the packet.	True
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 is 0 and Bit Size is 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	True
Data Type	Data Type of this telemetry item Valid Values: INT, UINT, FLOAT, STRING, BLOCK, DERIVED	True
Description	Description for this telemetry item which must be enclosed with quotes	False
Endianness	Indicates if the item is to be interpreted in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

Example Usage:

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

ID_ITEM

Defines a telemetry item in the current telemetry packet

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the telemetry item. Must be unique within the packet.	True
Bit Offset	Bit offset into the telemetry packet of the Most Significant Bit of this item. May be negative to indicate an offset from the end of the packet. Always use a bit offset of 0 for derived item.	True
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 is 0 and Bit Size is 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	True

Data Type	Data Type of this telemetry item Valid Values: INT, UINT, FLOAT, STRING, BLOCK, DERIVED	True
ID Value	The value of this telemetry item that uniquely identifies this telemetry packet	True
Description	Description for this telemetry item which must be enclosed with quotes	False
Endianness	Indicates if the item is to be interpreted in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

Example Usage:

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

APPEND_ID_ITEM

Defines a telemetry item in the current telemetry packet

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the telemetry item. Must be unique within the packet.	True
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 is 0 and Bit Size is 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	True
Data Type	Data Type of this telemetry item Valid Values: INT, UINT, FLOAT, STRING, BLOCK, DERIVED	True
ID Value	The value of this telemetry item that uniquely identifies this telemetry packet	True
Description	Description for this telemetry item which must be enclosed with quotes	False
Endianness	Indicates if the item is to be interpreted in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

Example Usage:

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

ARRAY_ITEM

Defines a telemetry item in the current telemetry packet that is an array

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the telemetry item. Must be unique within the packet.	True
Bit Offset	Bit offset into the telemetry packet of the Most Significant Bit of this item. May be negative to indicate an offset from the end of the packet. Always use a bit offset of 0 for derived item.	True
Item Bit Size	Bit size of each array item	True

Item Data Type	Data Type of each array item Valid Values: INT, UINT, FLOAT, STRING, BLOCK, DERIVED	True
Array Bit Size	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.	True
Description	Description which must be enclosed with quotes	False
Endianness	Indicates if the data is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

Example Usage:

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

APPEND_ARRAY_ITEM

Defines a telemetry item in the current telemetry packedt that is an array

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the telemetry item. Must be unique within the packet.	True
Item Bit Size	Bit size of each array item	True
Item Data Type	Data Type of each array item Valid Values: INT, UINT, FLOAT, STRING, BLOCK, DERIVED	True
Array Bit Size	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.	True
Description	Description which must be enclosed with quotes	False
Endianness	Indicates if the data is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

Example Usage:

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

SELECT_ITEM

Selects an existing telemetry item for editing

PARAMETER	DESCRIPTION	REQUIRED
Item	Name of the item to select for modification	True

Example Usage:

```
SELECT_TELEMETRY SYSTEM LIMITS_CHANGE
SELECT_ITEM ITEM
```

META

Stores metadata for the current telemetry packet

Meta data is user specific data that can be used by custom tools for various purposes. One example is to store additional information needed to generate source code header files.

PARAMETER	DESCRIPTION	REQUIRED
Meta Name	Name of the metadata to store	True
Meta Values	One or more values to be stored for this Meta Name	False

Example Usage:

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

PROCESSOR

Defines a processor class that executes code every time a packet is received

PARAMETER	DESCRIPTION	REQUIRED
Processor Name	The name of the processor	True
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.	True
Processor Specific Options	Variable length number of options that will be passed to the class constructor.	False

Example Usage:

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

ALLOW_SHORT

Process telemetry packets which are less than their defined length

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.

HIDDEN

Hides this telemetry packet from all the COSMOS tools

This packet will not appear in Packet Viewer, Telemetry Grapher and Handbook Creator. It also hides this telemetry from appearing in the Script Runner popup helper when writing scripts. The telemetry still exists in the system and can be received and checked by scripts.

ITEM Modifiers

The following keywords must follow a ITEM keyword.

FORMAT_STRING

Adds printf style formatting

PARAMETER	DESCRIPTION	REQUIRED
-----------	-------------	----------

Format	How to format using printf syntax. For example, '0x%0X' will display the value in hex.	True
--------	--	------

Example Usage:

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

UNITS

Add displayed units

PARAMETER	DESCRIPTION	REQUIRED
Full Name	Full name of the units type, e.g. Celcius	True
Abbreviated	Abbreviation for the units, e.g. C	True

Example Usage:

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

DESCRIPTION

Override the defined description

PARAMETER	DESCRIPTION	REQUIRED
Value	The new description	True

META

Stores custom user metadata

Meta data is user specific data that can be used by custom tools for various purposes. One example is to store additional information needed to generate source code header files.

PARAMETER	DESCRIPTION	REQUIRED
Meta Name	Name of the metadata to store	True
Meta Values	One or more values to be stored for this Meta Name	False

Example Usage:

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

STATE

Defines a key/value pair for the current item

Key value pairs allow for user friendly strings. 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 telemetry item and allows for much greater clarity and less chance for user error.

PARAMETER	DESCRIPTION	REQUIRED
Key	The string state name	True

Value	The numerical state value	True
Color	The color the state should be displayed as	False
	Valid Values: GREEN, YELLOW, RED	

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

Applies a conversion to the current telemetry item

Conversions are 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 REQUIRE). 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 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'.	True
Parameter	Additional parameter values for the conversion which are passed to the class constructor.	False

Example Usage:

```
READ_CONVERSION the_great_conversion.rb 1000
```

Defined in the_great_conversion.rb:

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

POLY_READ_CONVERSION

Adds a polynomial conversion factor to the current telemetry item

The conversion factor is applied to 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	True
Cx	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.	False

Example Usage:

```
POLY_READ_CONVERSION 10 0.5 0.25
```

SEG_POLY_READ_CONVERSION

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.	True
C0	Coefficient	True
Cx	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.	False

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

Start a generic read conversion

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 'packet' which is a reference to the telemetry 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_READ_CONVERSION_END keyword specifies that all lines of ruby code for the conversion have been given.



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.

PARAMETER	DESCRIPTION	REQUIRED
Converted Type	Type of the converted value Valid Values: INT, UINT, FLOAT, STRING, BLOCK	False
Converted Bit Size	Bit size of converted value	False

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
```

GENERIC_READ_CONVERSION_END

Complete a generic read conversion

LIMITS

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.	True
Persistence	Number of consecutive times the telemetry item must be within a different limits range before changing limits state.	True
Initial State	Whether limits monitoring for this telemetry item is initially enabled or disabled Valid Values: ENABLED, DISABLED	True
Red Low Limit	If the telemetry value is less than or equal to this value a Red Low condition will be detected	True
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	True
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	True
Red High Limit	If the telemetry value is greater than or equal to this value a Red High condition will be detected	True
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 will be detected.	False
Green High 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 less than or equal to this value, but greater than the Green Low Limit, a Blue operational condition will be detected.	False

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

Defines a response class that is 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.	True
Response Specific Options	Variable length number of options that will be passed to the class constructor	False

Example Usage:

```
LIMITS_RESPONSE example_limits_response.rb 10
```

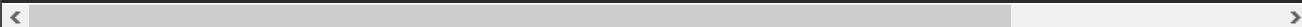
Example File

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

```

TELEMETRY 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 DERIVED "DERIVED TIME SINCE EPOCH IN SECONDS"
    GENERIC_READ_CONVERSION_START FLOAT 32
        ((packet.read('ccsdiday') * 86400.0) + (packet.read('ccsdsmmsod') / 1000.0) + (packet.read('ccsdssusoms') / 1000000.0))
    GENERIC_READ_CONVERSION_END
ITEM TIMEFORMATTED 0 0 DERIVED "DERIVED TIME SINCE EPOCH AS A FORMATTED STRING"
    GENERIC_READ_CONVERSION_START STRING 216
        time = Time.ccsds2mdy(packet.read('ccsdiday'), packet.read('ccsdsmmsod'), packet.read('ccsdssusoms'))
        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



Navigate the docs... ▾

✍ Improve this page

- Note on Naming

- **Provided Interfaces**
 - [TCPIP Client Interface](#)
 - [TCPIP Server Interface](#)
 - [UDP Interface](#)
 - [Serial Interface](#)
 - [CmdTImServer Interface](#)
 - [LINC Interface](#)
- [Streams](#)
- [Protocols](#)

Interface Configuration

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 customizable 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. Note that in most cases you can extend interfaces with [Protocols](#) rather than implementing a new interface.



Note that Interfaces and Routers are very similar and share the same configuration parameters. Routers are simply Interfaces which route an existing Interface's telemetry data out to the connected target and routes the connected target's commands back to the original Interface's target.

Interfaces have the following methods that must be implemented:

1. **connect** - Open the socket or port or somehow establish the connection to the target. Note: This method may not block indefinitely. Be sure to call super() in your implementation.
2. **connected?** - Return true or false depending on the connection state. Note: This method should return immediately.
3. **disconnect** - Close the socket or port or somehow disconnect from the target. Note: This method may not block indefinitely. Be sure to call super() in your implementation.
4. **read_interface** - Lowest level read of data on the interface. Note: This method should block until data is available or the interface disconnects. On a clean disconnect it should return nil.
5. **write_interface** - Lowest level write of data on the interface. Note: This method may not block indefinitely.

Interfaces also have the following methods that exist and have default implementations. They can be overridden if

necessary but be sure to call super() to allow the default implementation to be executed.

1. **read_interface_base** - This method should always be called from read_interface(). It updates interface specific variables that are displayed by CmdTLmServer including the bytes read count, the most recent raw data read, and it handles raw logging if enabled.
2. **write_interface_base** - This method should always be called from write_interface(). It updates interface specific variables that are displayed by CmdTLmServer including the bytes written count, the most recent raw data written, and it handles raw logging if enabled.
3. **read** - Read the next packet from the interface. COSMOS implements this method to allow the Protocol system to operate on the data and the packet before it is returned.
4. **write** - Send a packet to the interface. COSMOS implements this method to allow the Protocol system to operate on the packet and the data before it is sent.
5. **write_raw** - Send a raw binary string of data to the target. COSMOS implements this method by basically calling write_interface with the raw data.



Note on Naming

When creating your own interfaces, in most cases they will be subclasses of one of the built-in interfaces described below. It is important to know that both the filename and class name of the interface files must match with correct capitalization or you will receive "class not found" errors when trying to load your new interface. For example, an interface file called labview_interface.rb must contain the class LabviewInterface. If the class was named, LabVIEWInterface, for example, COSMOS would not be able to find the class because of the unexpected capitalization.

Provided Interfaces

Cosmos provides the following interfaces for use: TCPIP Client, TCPIP Server, UDP, Serial, Command Telemetry Server, and LINC. The interface to use is defined by the [INTERFACE](#) and [ROUTER](#) keywords.

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
Protocol Type	See Protocols.	No
Protocol Arguments	See Protocols for the arguments each stream protocol takes.	No

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
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 0xCAFEBADE
INTERFACE INTERFACE_NAME tcpip_client_interface.rb localhost 8080 8080 10.0 10.0 TERMINATED 0x0D0A 0xA
INTERFACE INTERFACE_NAME tcpip_client_interface.rb localhost 8080 8080 10.0 10.0 TEMPLATE 0xA 0xA
INTERFACE INTERFACE_NAME tcpip_client_interface.rb localhost 8080 8080 10.0 10.0 # no built-in protocol

```

See [INTERFACE](#) for a description of the INTERFACE keyword. See [Interface Modifiers](#) for a description of the keywords which can follow the INTERFACE keyword.

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
Protocol Type	See Protocols.	No
Protocol Arguments	See Protocols for the arguments each stream protocol takes.	No

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 0
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 0xCAFEBADE
INTERFACE INTERFACE_NAME tcpip_server_interface.rb 8080 8080 10.0 10.0 TERMINATED 0x0D0A 0x0D0A true
INTERFACE INTERFACE_NAME tcpip_client_interface.rb 8080 8080 10.0 10.0 TEMPLATE 0xA 0xA
INTERFACE INTERFACE_NAME tcpip_client_interface.rb 8080 8080 10.0 10.0 # no built-in protocol

```

See [INTERFACE](#) for a description of the INTERFACE keyword. See [Interface Modifiers](#) for a description of the keywords which can follow the INTERFACE keyword. Note, TcpipServerInterface processes the [OPTION](#) modifier.

UDP Interface

The UDP interface uses UDP packets to send and receive telemetry from the target.

PARAMETER	DESCRIPTION	REQUIRED	DEFAULT
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	No	nil (socket is not bound to an outgoing port)
Interface Address	If the remote machine supports multicast the interface address is used to configure the outgoing multicast address	No	nil (not used)
TTL	Time to Live. The number of intermediate routers allowed before dropping the packet.	No	128 (Windows)
Write Timeout	Number of seconds to wait before aborting the write	No	nil (block on write)
Read Timeout	Number of seconds to wait before aborting the read	No	nil (block on read)

cmd_tlm_server.txt Example:

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

See [INTERFACE](#) for a description of the INTERFACE keyword. See [Interface Modifiers](#) for a description of the keywords which can follow the INTERFACE keyword.

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
Protocol Type	See Protocols.	No
Protocol Arguments	See Protocols for the arguments each stream protocol takes.	No

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
INTERFACE INTERFACE_NAME serial_interface.rb /dev/ttyS1 /dev/ttyS1 38400 ODD 1 10.0 nil BURST 4 0xDE
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
INTERFACE INTERFACE_NAME serial_interface.rb COM4 COM4 115200 NONE 1 10.0 10.0 TERMINATED 0x0D0A 0x01
INTERFACE INTERFACE_NAME serial_interface.rb COM4 COM4 115200 NONE 1 10.0 10.0 TEMPLATE 0xA 0xA
INTERFACE INTERFACE_NAME serial_interface.rb COM4 COM4 115200 NONE 1 10.0 10.0 # no built-in protocol

```

See [INTERFACE](#) for a description of the INTERFACE keyword. See [Interface Modifiers](#) for a description of the keywords which can follow the INTERFACE keyword. Note, SerialInterface processes the OPTION modifier.

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
```

See [INTERFACE](#) for a description of the INTERFACE keyword. See [Interface Modifiers](#) for a description of the keywords which can follow the INTERFACE keyword.

LINC Interface

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

PARAMETER	DESCRIPTION	REQUIRED	DEFAULT
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	No	true
Response Timeout	Number of seconds to wait for a handshaking response	No	5 seconds
Read Timeout	Number of seconds to wait before aborting the read	No	nil (block on read)
Write Timeout	Number of seconds to wait before aborting the write	No	5 seconds
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.	No	0 bits
Length Bit Size	The size in bits of the length field	No	16 bits
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.	No	4

Fieldname GUID	Fieldname of the GUID field	No	'HDR_GUID'
Length Endianness	The endianness of the length field. Must be either 'BIG_ENDIAN' or 'LITTLE_ENDIAN'.	No	'BIG_ENDIAN'
Fieldname Cmd Length	Fieldname of the length field	No	'HDR_LENGTH'

cmd_tlm_server.txt Examples:

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

See [INTERFACE](#) for a description of the INTERFACE keyword. See [Interface Modifiers](#) for a description of the keywords which can follow the INTERFACE keyword.

Streams

Streams are low level classes that implement read, read_nonblock, write, connect, connected? and disconnect methods. The build-in Stream classes are SerialStream, TcpipSocketStream and TcpipClientStream and they are automatically used when creating a Serial Interface, TCP/IP Server Interface, or TCP/IP Client Interface.

Protocols

Protocols define the behaviour of an Interface, including differentiating packet boundaries and modifying data as necessary. COSMOS defines the following built-in protocols which can be used with the above interfaces:

NAME	DESCRIPTION
Burst	Reads as much data as possible from the interface
Fixed	Processes fixed length packets with a known ID position
Length	Processes a length field at a fixed location and then reads the remainder of the data
Terminated	Delineates packets uses termination characters at the end of each packet
Template	Processes text based command / response data such as SCPI interfaces
Preidentified	Internal COSMOS protocol used by COSMOS tools

These protocols are declared directly after the interface:

```
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 8081 10.0 nil LENGTH 0 16 0 1 BIG_ENDIAN
INTERFACE INTERFACE_NAME tcpip_client_interface.rb localhost 8080 8080 10.0 10.0 TEMPLATE 0xA 0xA
INTERFACE INTERFACE_NAME tcpip_client_interface.rb localhost 8080 8080 10.0 nil PREIDENTIFIED 0xCAFE
```

COSMOS also defines the following helper protocols:

NAME	DESCRIPTION
Override	Allows telemetry items to be fixed to given value when read

CRC

Adds CRCs to outgoing packets and verifies CRCs on incoming packets

These protocols are declared after the INTERFACE:

```
INTERFACE INTERFACE_NAME tcpip_client_interface.rb localhost 8080 8080 10.0 nil BURST 4 0xDEADBEEF
TARGET TGT
PROTOCOL READ OverrideProtocol
PROTOCOL WRITE CrcProtocol CRC # See the documentation for parameters
```

Note the first parameter after the PROTOCOL keyword is how to apply the protocol: READ, WRITE, or READ_WRITE. Read applies the protocol on incoming packets (telemetry) and write on outgoing packets (commands). The next parameter is the protocol filename or class name. All other parameters are protocol specific.

In addition, you can define your own protocols which are declared like the COSMOS helper protocols after your interface. See the [Custom Protocols](#) documentation for more information.

BACK

NEXT



Navigate the docs... ▾

✍ Improve this page

- [Packet Delineation Protocols](#)
 - [Burst Protocol](#)
 - [Fixed Protocol](#)
 - [Length Protocol](#)
 - [Terminated Protocol](#)
 - [Template Protocol](#)
 - [Preidentified Protocol](#)
- [Helper Protocols](#)
 - [Override Protocol](#)
 - [CRC Protocol](#)
- [Custom Protocols](#)
- [Method discussions](#)
 - [initialize](#)
 - [reset](#)
 - [connect_reset](#)
 - [disconnect_reset](#)
 - [read_data](#)
 - [read_packet](#)
 - [write_packet](#)
 - [write_data](#)
 - [post_write_interface](#)
- [Examples](#)

Protocol Configuration

Protocols process data on behalf of an Interface. They can modify the data being written, data being read, or both. Protocols can also mark a packet as stored instead of real-time which means COSMOS will not update the current value table with the packet data. Protocols can be layered and will be processed in order. For example, if you have a low-level encryption layer that must be first removed before processing a higher level buffer length protocol.

Protocols are typically used to define the logic to delineate packets and manipulate data as it written to and read from Interfaces. COSMOS includes Interfaces for TCP/IP Client, TCP/IP Server, Udp Client / Server, and Serial connections. For 99% of use cases these Interfaces should not require any changes as they universally handle the low-level details of reading and writing from these types of connections. All unique behavior should now be defined in Protocols as of COSMOS 4.0.0. (Note in versions of COSMOS before COSMOS 4, Interfaces supported a more limited system called Stream Protocols. Interfaces only allowed one Stream Protocol and were highly coupled. This document refers to protocols in COSMOS 4+.)

At a minimum, any byte stream based Interface will require a Protocol to delineate packets. TCP/IP and Serial are

examples of byte stream based Interfaces. A byte stream is just a simple stream of bytes and thus you need some way to know where packets begin and end within the stream.

TCP/IP is a friendly byte stream. Unless you are dealing with a very poorly written system, the first byte received on a TCP/IP connection will always be the start of a packet. Also, TCP/IP is a reliable connection in that it ensures that all data is received in the correct order, that no data is lost, and that the data is not corrupted (TCP/IP is protected by a CRC32 which is pretty good for avoiding unrecognized data corruption).

Serial is a much less friendly byte stream. With serial connections, it is very likely that when you open a serial port and start receiving data you will receive the middle of a message. (This problem is only avoided when interfacing with a system that only writes to the serial port in response to a command). For this reason, sync patterns are highly beneficial for serial interfaces. Additionally, serial interfaces may use some method to protect against unrecognized data corruption (Checksums, CRCs, etc.)

UDP is an inherently packet based connection. If you read from a UDP socket, you will always receive back an entire packet. The best UDP based Protocols take advantage of this fact. Some implementations try to make UDP act like a byte stream, but this is a misuse of the protocol because it is highly likely that you will lose data and have no way to recover.

Packet Delineation Protocols

COSMOS provides the following packet delineation protocols: Burst, Fixed, Length, Preidentified, Template, and Terminated. Each of these protocols has the primary purpose of separating out packets from a byte stream.

Burst Protocol

The Burst Protocol simply reads as much data as it can from the interface before returning the data as a COSMOS Packet (It returns a packet for each burst of data read). 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 if necessary. It can also discard bytes from the incoming data to remove the sync pattern. Finally, it can add sync patterns to data being written out of the Interface.

PARAMETER	DESCRIPTION	REQUIRED	DEFAULT
Discard Leading Bytes	The number of bytes to discard from the binary data after reading. Note that this applies to bytes starting with the sync pattern if the sync pattern is being used.	No	0 (do not discard bytes)
Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw data. This pattern represents a packet delimiter and all data found including the sync pattern will be returned	No	nil (no sync pattern)
Fill Fields	Whether to fill in the sync pattern on outgoing packets	No	false
Allow Empty Data	Whether this protocol will allow an empty string to be passed down to later Protocols (instead of returning :STOP). Can be true, false, or nil, where nil is interpreted as true unless the Protocol is the last Protocol of the chain. (As of COSMOS 4.1.1)	No	nil

Fixed Protocol

The Fixed Protocol reads a preset minimum amount of data 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 interface as necessary to create the packet which it then returns. This protocol 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 Protocol would elegantly handle this case with a minimum read size of 6 bytes. The Fixed Protocol also supports a sync pattern, discarding leading bytes, and filling the sync pattern similar to the Burst Protocol.

PARAMETER	DESCRIPTION	REQUIRED	DEFAULT
Minimum ID Size	The minimum number of bytes needed to identify a packet. All the packet definitions must declare their ID_ITEM(s) within this given number of bytes.	Yes	
Discard Leading Bytes	The number of bytes to discard from the binary data after reading. Note that this applies to bytes starting with the sync pattern if the sync pattern is being used.	No	0 (do not discard bytes)
Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw data. This pattern represents a packet delimiter and all data found including the sync pattern will be returned.	No	nil (no sync pattern)
Telemetry	Whether the data is telemetry	No	true (false means command)
Fill Fields	Whether to fill in the sync pattern on outgoing packets	No	false
Unknown Raise	Whether to raise an exception for an unknown packet	No	false
Allow Empty Data	Whether this protocol will allow an empty string to be passed down to later Protocols (instead of returning :STOP). Can be true, false, or nil, where nil is interpreted as true unless the Protocol is the last Protocol of the chain. (As of COSMOS 4.1.1)	No	nil

Length Protocol

The Length 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 Protocol can be set up to handle the length field and even the length offset CCSDS uses. The Length Protocol also supports a sync pattern, discarding leading bytes, and filling the length and sync pattern similar to the Burst Protocol.

PARAMETER	DESCRIPTION	REQUIRED	DEFAULT
Length Bit Offset	The bit offset from the start of the packet to 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. Be sure to account for the length of the Sync Pattern in this value (if present).	No	0 bits
Length Bit Size	The size in bits of the length field	No	16 bits
Length Value Offset	The offset to apply to the length field value. The actual value of the length field plus this offset should equal the exact number of bytes required to read all data for the packet (including the length field itself, sync pattern, etc). For example, if the length field indicates packet length minus one, this value should be one. Be sure to account for the length of the Sync Pattern in this value (if present).	No	0
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, e.g. if the length field count is in words.	No	1 byte
Length Endianness	The endianness of the length field. Must be either 'BIG_ENDIAN' or 'LITTLE_ENDIAN'.	No	'BIG_ENDIAN'

Discard Leading Bytes	The number of bytes to discard from the binary data after reading. Note that this applies to bytes including the sync pattern if the sync pattern is being used. Discarding is one of the very last steps so any size and offsets above need to account for all the data before discarding.	No	0 (do not discard bytes)
Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw data. This pattern represents a packet delimiter and all data found including the sync pattern will be returned.	No	nil (no sync pattern)
Max Length	The maximum allowed value in the length field	No	nil (no maximum length)
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.	No	false
Allow Empty Data	Whether this protocol will allow an empty string to be passed down to later Protocols (instead of returning :STOP). Can be true, false, or nil, where nil is interpreted as true unless the Protocol is the last Protocol of the chain. (As of COSMOS 4.1.1)	No	nil

Terminated Protocol

The Terminated Protocol delineates packets using termination characters found at the end of every packet. It continuously reads data 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 Protocol and simply thrown away.

PARAMETER	DESCRIPTION	REQUIRED	DEFAULT
Write Termination Characters	The data to write after writing a command packet. Given as a hex string such as 0xABCD.	Yes	
Read Termination Characters	The characters which delineate the end of a telemetry packet. Given as a hex string such as 0xABCD.	Yes	
Strip Read Termination	Whether to remove the read termination characters before returning the telemetry packet	No	true
Discard Leading Bytes	The number of bytes to discard from the binary data after reading. Note that this applies to bytes including the sync pattern if the sync pattern is being used.	No	0 (do not discard bytes)
Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw data. This pattern represents a packet delimiter and all data found including the sync pattern will be returned.	No	nil (no sync pattern)
Fill Fields	Whether to fill in the sync pattern on outgoing packets	No	false
Allow Empty Data	Whether this protocol will allow an empty string to be passed down to later Protocols (instead of returning :STOP). Can be true, false, or nil, where nil is interpreted as true unless the Protocol is the last Protocol of the chain. (As of COSMOS 4.1.1)	No	nil

Template Protocol

The Template Protocol works much like the Terminated Protocol except it is designed for text-based command and

response type interfaces such as SCPI (Standard Commands for Programmable Instruments). It delineates packets in the same way as the Terminated 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 delineated within HTML tag style brackets "**<EXAMPLE>**". The Template Protocol will read the named items from within the packet and 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 TEMPLATE target within the COSMOS Demo configuration for an example of usage.

Check out this [Template Protocol](#) blog post for additional tips.

PARAMETER	DESCRIPTION	REQUIRED	DEFAULT
Write Termination Characters	The data to write after writing a command packet. Given as a hex string such as 0xABCD.	Yes	
Read Termination Characters	The characters which delineate the end of a telemetry packet. Given as a hex string such as 0xABCD.	Yes	
Ignore Lines	Number of response lines to ignore (completely drop)	No	0 lines
Initial Read Delay	An initial delay after connecting after which the interface will be read till empty and data dropped. Useful for discarding connect headers and initial prompts.	No	nil (no initial read)
Response Lines	The number of lines that make up expected responses	No	1 line
Strip Read Termination	Whether to remove the read termination characters before returning the telemetry packet	No	true
Discard Leading Bytes	The number of bytes to discard from the binary data after reading. Note that this applies to bytes including the sync pattern if the sync pattern is being used.	No	0 (do not discard bytes)
Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw data. This pattern represents a packet delimiter and all data found including the sync pattern will be returned.	No	nil (no sync pattern)
Fill Fields	Whether to fill in the sync pattern on outgoing packets	No	false
Response Timeout	Number of seconds to wait for a response before timing out	No	5.0
Response Polling Period	Number of seconds to wait between polling for a response	No	0.02
Raise Exceptions	Whether to raise exceptions when errors occur like timeouts or unexpected responses	No	false
Allow Empty Data	Whether this protocol will allow an empty string to be passed down to later Protocols (instead of returning :STOP). Can be true, false, or nil, where nil is interpreted as true unless the Protocol is the last Protocol of the chain. (As of COSMOS 4.1.1)	No	nil

Preidentified Protocol

The Preidentified Protocol is used internally by the COSMOS Command and Telemetry Server only and delineates packets using a custom COSMOS header. This 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	DEFAULT
Sync Pattern	Hex string representing a byte pattern that will be searched for in the raw data. This pattern represents a packet delimiter and all data found AFTER the sync pattern will be returned. The sync pattern itself is discarded.	No	nil (no sync pattern)
Max Length	The maximum allowed value in the length field	No	nil (no maximum length)
Mode	The Version of the preidentified protocol to support (2 or 4) - 4 is the new protocol released in COSMOS 4.3 (This option is only available in COSMOS 4.3+)	No	4
Allow Empty Data	Whether this protocol will allow an empty string to be passed down to later Protocols (instead of returning :STOP). Can be true, false, or nil, where nil is interpreted as true unless the Protocol is the last Protocol of the chain. (As of COSMOS 4.1.1)	No	nil

Helper Protocols

COSMOS provides the following helper protocols: Override, and Crc. These protocols provide helper functionality to Interfaces.

Override Protocol

The Override Protocol allows telemetry items to be overridden when read. This action is permanent and any incoming data is overwritten with the new value. This is in contrast to using the set_tlm() API method which temporarily sets a telemetry value until new data arrives over the interface.

To use this protocol, you must add it to your interface wherever that is defined. Typically, this is in the target's cmd_tlm_server.txt file but it could also be in the global cmd_tlm_server.txt. For example:

```
INTERFACE INST_INT simulated_target_interface.rb sim_inst.rb
TARGET INST
PROTOCOL READ OverrideProtocol
```

By adding this to your interface you now have access to the following APIs:

```
# Permanently set the converted value of a telemetry point to a given value
override_tlm(target_name, packet_name, item_name, value)
# or
override_tlm("target_name packet_name item_name = value")

# Permanently set the raw value of a telemetry point to a given value
override_tlm_raw(target_name, packet_name, item_name, value)
# or
override_tlm_raw("target_name packet_name item_name = value")

# Clear an override of a telemetry point
normalize_tlm(target_name, packet_name, item_name)
# or
normalize_tlm("target_name packet_name item_name")
```

PARAMETER	DESCRIPTION	REQUIRED	DEFAULT
Allow Empty Data	Whether this protocol will allow an empty string to be passed down to later Protocols (instead of returning :STOP). Can be true, false, or nil, where nil is interpreted as true unless the Protocol is the last Protocol of the chain. (As of COSMOS 4.1.1)	No	nil

CRC Protocol

The CRC protocol can add CRCs to outgoing commands and verify CRCs on incoming telemetry packets.

PARAMETER	DESCRIPTION	REQUIRED	DEFAULT
Write Item Name	Item to fill with calculated CRC value for outgoing packets (nil = don't fill)	No	nil
Strip CRC	Whether to remove the CRC from incoming packets	No	false
Bad Strategy	How to handle CRC errors on incoming packets. ERROR = Just log the error, DISCONNECT = Disconnect interface	No	"ERROR"
Bit Offset	Bit offset of the CRC in the data. Can be negative to indicate distance from end of packet	No	-32
Bit Size	Bit size of the CRC - Must be 16, 32, or 64	No	32
Endianness	Endianness of the CRC (BIG_ENDIAN/LITTLE_ENDIAN)	No	"BIG_ENDIAN"
Poly	Polynomial to use when calculating the CRC expressed as an integer	No	nil (use default polynomial - 16-bit=0x1021, 32-bit=0x04C11DB7, 64-bit=0x42FOE1EBA9EA3693)
Seed	Seed value to start the calculation	No	nil (use default seed - 16-bit=0xFFFF, 32-bit=0xFFFFFFFF, 64-bit=0xFFFFFFFFFFFFFFFF)
Xor	Whether to XOR the CRC result with 0xFFFF	No	nil (use default value - 16-bit=false, 32-bit=true, 64-bit=true)
Reflect	Whether to bit reverse each byte of data before calculating the CRC	No	nil (use default value - 16-bit=false, 32-bit=true, 64-bit=true)
Allow Empty Data	Whether this protocol will allow an empty string to be passed down to later Protocols (instead of returning :STOP). Can be true, false, or nil, where nil is interpreted as true unless the Protocol is the last Protocol of the chain. (As of COSMOS 4.1.1)	No	nil

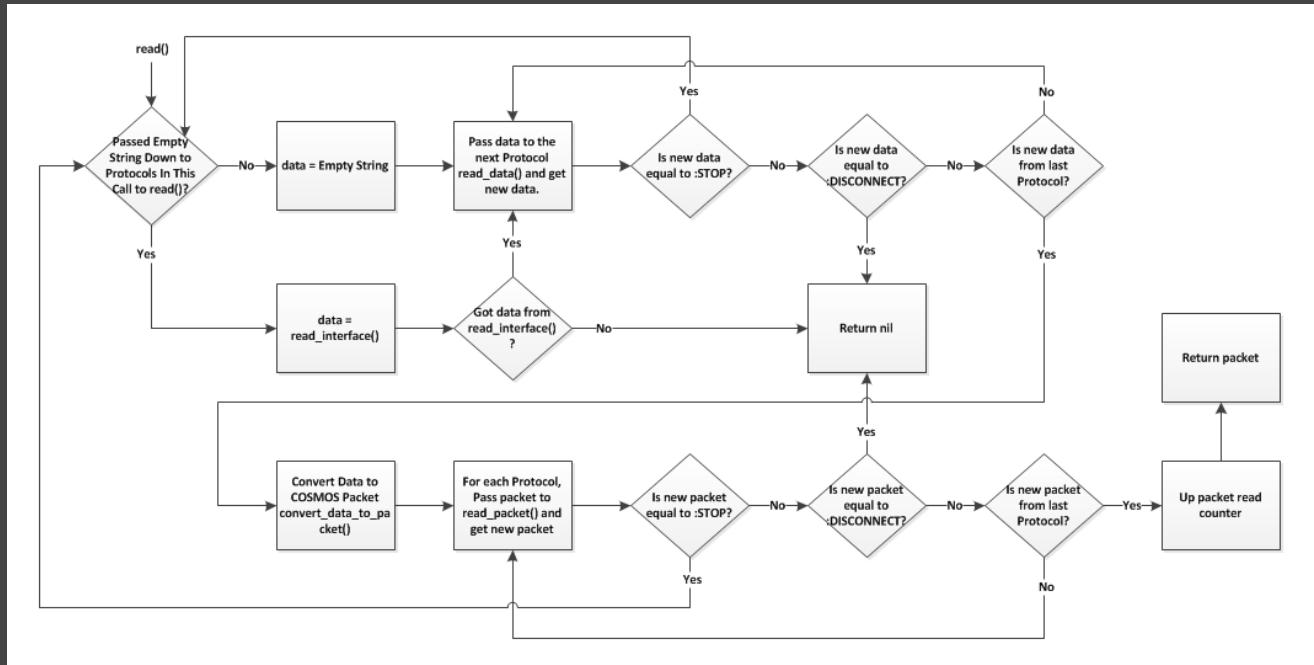
Custom Protocols

Creating a custom protocol is easy and should be the default solution for customizing COSMOS Interfaces (rather than creating a new Interface class). However, creating custom Interfaces is still useful for defaulting parameters to values that always are fixed for your target and for including the necessary Protocols. The base COSMOS Interfaces take a lot of parameters that can be confusing to your end users. Thus you may want to create a custom Interface just to hard coded these values and cut the available parameters down to something like the hostname and port to connect to.

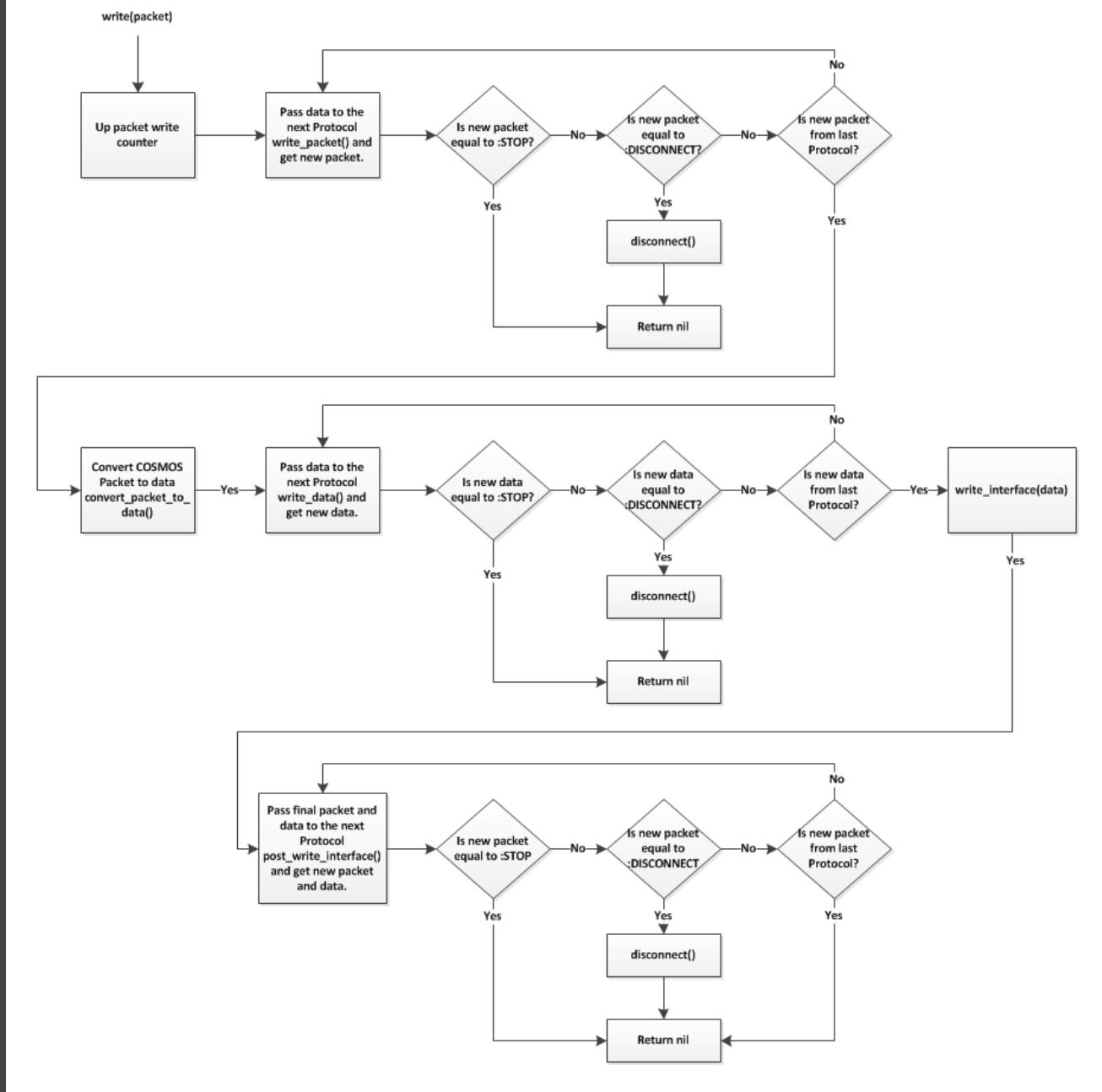
All custom Protocols should derive from the Protocol class found in the COSMOS gem at [lib/cosmos/interfaces/protocols/protocol.rb](#). This class defines the 9 methods that are relevant to writing your own protocol. The base class implementation for each method is included below as well as a discussion as to how the methods should be overridden and used in your own Protocols.

To really understand how Protocols work, you first must understand the logic within the base Interface class read and write methods.

Let's first discuss the read method.



On EVERY call to read, an empty Ruby string "" is first passed down to each of the read Protocol's `read_data()` method BEFORE new raw data is attempted to be read using the Interface's `read_interface()` method. This is a signal to Protocols that have cached up more than one packet worth of data to output those cached packets before any new data is read from the Interface. Typically no data will be cached up and one of the Protocols `read_data()` methods will return `:STOP` in response to the empty string, indicating that more data is required to generate a packet. Each Protocol's `read_data()` method can return one of three things: data that will be passed down to any additional Protocols or turned into a Packet, `:STOP` which means more data is required from the Interface for the Protocol to continue, or `:DISCONNECT` which means that something has happened that requires disconnecting the Interface (and by default trying to reconnect). Each Protocol's `read_data` method is passed the data that will eventually be turned into a packet and returns a possibly modified set of data. If the data passes through all Protocol's `read_data()` methods it is then converted into a COSMOS packet using the Interface's `convert_data_to_packet()` method. This packet is then run in a similar fashion through each Read Protocol's `read_packet()` method. This method has essentially the same return possibilities: a Packet (instead of data as in `read_data()`), `:STOP`, or `:DISCONNECT`. If the Packet makes it through all `read_packet()` methods then the Interface packet read counter is incremented and the Packet is returned to the Interface.



The Interface write() method works very similarly to read. (It should be mentioned that by default write protocols run in the reverse order of read protocols. This makes sense because when reading you're typically stripping layers of data and when writing you're typically adding on layers in reverse order.)

First, the packet write counter is incremented. Then each write Protocol is given a chance to modify the packet by its write_packet() method being called. This method can either return a potentially modified packet, :STOP, or :DISCONNECT. If a write Protocol returns :STOP no data will be written out the Interface and it is assumed that more packets are necessary before a final packet can be output. :DISCONNECT will disconnect the Interface. If the packet makes it through all the write Protocol's write_packet() methods, then it is converted to binary data using the Interface's convert_packet_to_data() method. Next the write_data() method is called for each write Protocol giving it a chance to modify the lower level data. The same return options are available except a Ruby string of data is returned instead of a COSMOS packet. If the data makes it through all write_data() methods, then it is written out on the Interface using the write_interface() method. Afterwards, each Protocol's post_write_interface() method is called with both the final modified Packet, and the actual data written out to the Interface. This method allows follow-up such as waiting for a response after writing out a message.

Method discussions

initialize

This is the constructor for your custom Protocol. It should always call super(allow_empty_data) to initialize the base Protocol class.

Base class implementation:

```
# @param allow_empty_data [true/false] Whether STOP should be returned on empty data
def initialize(allow_empty_data = false)
  @interface = nil
  @allow_empty_data = ConfigParser.handle_true_false(allow_empty_data)
  reset()
end
```

As you can see, every Protocol maintains state on at least two items. @interface holds the Interface class instance that the protocol is associated with. This is sometimes necessary to introspect details that only the Interface knows. @allow_empty_data is a flag used by the read_data(data) method that is discussed later in this document.

reset

The reset method is used to reset internal protocol state when the Interface is connected and/or disconnected. This method should be used for common resetting logic. Connect and Disconnect specific logic are handled in the next two methods.

Base class implementation:

```
def reset
end
```

As you can see, the base class reset implementation doesn't do anything.

connect_reset

The connect_reset method is used to reset internal Protocol state each time the Interface is connected.

Base class implementation:

```
def connect_reset
  reset()
end
```

The base class connect_reset implementation just calls the reset method to ensure common reset logic is run.

disconnect_reset

The disconnect_reset method is used to reset internal Protocol state each time the Interface is disconnected.

Base class implementation:

```
def disconnect_reset
  reset()
end
```

The base class disconnect_reset implementation just calls the reset method to ensure common reset logic is run.

read_data

The read_data method is used to analyze and potentially modify any raw data read by an Interface. It takes one

parameter as the current state of the data to be analyzed. It can return either a Ruby string of data, :STOP, or :DISCONNECT. If it returns a Ruby string, then it believes that data may be ready to be a full packet, and is ready for processing by any following Protocols. If :STOP is returned then the Protocol believes it needs more data to complete a full packet. If :DISCONNECT is returned then the Protocol believes the Interface should be disconnected (and typically automatically reconnected).

Base Class Implementation:

```
def read_data(data)
  if (data.length <= 0)
    if @allow_empty_data.nil?
      if @interface and @interface.read_protocols[-1] == self
        # Last read interface in chain with auto @allow_empty_data
        return :STOP
      end
    elsif !@allow_empty_data
      # Don't @allow_empty_data means STOP
      return :STOP
    end
  end
  data
end
```

The base class implementation does nothing except return the data it was given. The only exception to this is when handling an empty string. If the allow_empty_data flag is false or if it nil and the Protocol is the last in the chain, then the base implementation will return :STOP data to indicate that it is time to call the Interface read_interface() method to get more data. Blank strings are used to signal Protocols that they have an opportunity to return a cached packet.

read_packet

The read_packet method is used to analyze and potentially modify a COSMOS packet before it is returned by the Interface. It takes one parameter as the current state of the packet to be analyzed. It can return either a COSMOS packet, :STOP, or :DISCONNECT. If it returns a COSMOS packet, then it believes that the packet is valid, should be returned, and is ready for processing by any following Protocols. If :STOP is returned then the Protocol believes the packet should be silently dropped. If :DISCONNECT is returned then the Protocol believes the Interface should be disconnected (and typically automatically reconnected). This method is where a Protocol would set the stored flag on a packet if it determines that the packet is stored telemetry instead of real-time telemetry.

Base Class Implementation:

```
def read_packet(packet)
  return packet
end
```

The base class always just returns the packet given.

write_packet

The write_packet method is used to analyze and potentially modify a COSMOS packet before it is output by the Interface. It takes one parameter as the current state of the packet to be analyzed. It can return either a COSMOS packet, :STOP, or :DISCONNECT. If it returns a COSMOS packet, then it believes that the packet is valid, should be written out the Interface, and is ready for processing by any following Protocols. If :STOP is returned then the Protocol believes the packet should be silently dropped. If :DISCONNECT is returned then the Protocol believes the Interface should be disconnected (and typically automatically reconnected).

Base Class Implementation:

```
def write_packet(packet)
  return packet
end
```

The base class always just returns the packet given.

write_data

The write_data method is used to analyze and potentially modify data before it is written out by the Interface. It takes one parameter as the current state of the data to be analyzed and sent. It can return either a Ruby String of data, :STOP, or :DISCONNECT. If it returns a Ruby string of data, then it believes that the data is valid, should be written out the Interface, and is ready for processing by any following Protocols. If :STOP is returned then the Protocol believes the data should be silently dropped. If :DISCONNECT is returned then the Protocol believes the Interface should be disconnected (and typically automatically reconnected).

Base Class Implementation:

```
def write_data(data)
  return data
end
```

The base class always just returns the data given.

post_write_interface

The post_write_interface method is called after data has been written out the Interface. The typical use of this method is to provide a hook to implement command/response type interfaces where a response is always immediately expected in response to a command. It takes two parameters, the packet after all modifications by write_packet() and the data that was actually written out the Interface. It can return either the same pair of packet/data, :STOP, or :DISCONNECT. If it returns a packet/data pair then they are passed on to any other Protocols. If :STOP is returned then the Interface write() call completes and no further Protocols post_write_interface() methods are called. If :DISCONNECT is returned then the Protocol believes the Interface should be disconnected (and typically automatically reconnected). Note that only the first parameter “packet”, is checked to be :STOP, or :DISCONNECT on the return.

Base Class Implementation:

```
def post_write_interface(packet, data)
  return packet, data
end
```

The base class always just returns the packet/data given.

Examples

Please see the included COSMOS protocol code for examples of the above methods in action.

[lib/cosmos/interfaces/protocols/protocol.rb](#) [lib/cosmos/interfaces/protocols/burst_protocol.rb](#)
[lib/cosmos/interfaces/protocols/fixed_protocol.rb](#) [lib/cosmos/interfaces/protocols/length_protocol.rb](#)
[lib/cosmos/interfaces/protocols/preidentified_protocol.rb](#) [lib/cosmos/interfaces/protocols/terminated_protocol.rb](#)
[lib/cosmos/interfaces/protocols/template_protocol.rb](#) [lib/cosmos/interfaces/protocols/override_protocol.rb](#)
[lib/cosmos/interfaces/protocols/crc_protocol.rb](#)

 BACK

NEXT 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Custom Search



Proudly hosted by **GitHub**



Navigate the docs... ▾

Environment Variables

Improve this page

Environment Variables Affecting Tool Execution

ENVIRONMENT VARIABLE	DESCRIPTION
COSMOS_DIR	Windows Only. Tells the Windows .bat launcher files where to find the COSMOS Installation of Ruby. Typically set to C:\COSMOS. Expects Vendor\Ruby to exist in the directory. Is not used if Vendor\Ruby is found within two directories up of the executing .bat (typically within a COSMOS Installation)
COSMOS_USERPATH	Allows scripts that run outside of your COSMOS project configuration folder to know where the project configuration folder is. Only needed if you plan on using scripts that rely on your project specific command/telemetry definitions or other configuration outside of the project folder.
COSMOS_TEXT	Process to launch for COSMOS "Open in Text Editor" buttons. Should be exactly what needs to be typed from a command line and expect a single argument of a quoted filename to open.
RUBYLIB	Adds folders to the Ruby library search path.
RUBYOPT	Always passes specific command line options to the ruby interpreter
GEM_PATH	Provides one or more paths to where gems may reside
GEM_HOME	Provides the location where gems will be installed
VISUAL	Preferred text editor used on linux if COSMOS_TEXT is not defined and gedit is not present
EDITOR	Text editor used on linux if COSMOS_TEXT is not defined, and VISUAL is not defined, and gedit is not present
PATH	PATH controls what executables are available to COSMOS when it shells out to tools. It also affects the dynamic library search on some platforms.
SystemRoot	Windows Only. Used to know the windows installation folder.

Environment Variables Only Affecting Development And Unit Testing

ENVIRONMENT VARIABLE	DESCRIPTION
COSMOS_DEVEL	Path to a local COSMOS development area. Overrides using the gem in Gemfiles.
COSMOS_NO_SIMPLECOV	If defined runs unit tests without doing coverage.
PROFILE	If defined runs unit tests with profiling

BENCHMARK	If defined runs unit tests with benchmarking
TRAVIS	Used to skip specific unit tests when running in the Travis environment
APPVEYOR	Used to skip specific unit tests when running in the AppVeyor environment
VERSION	Used to specify the COSMOS version of the gemspec and Rakefile when building COSMOS gems

Environment Variables Set By COSMOS

ENVIRONMENT VARIABLE	DESCRIPTION
COSMOS_LOGS_DIR	Set by COSMOS so that tools outside of COSMOS can know where they can put log files. Specifically used by the SegFault catching code to capture segfault logs.
COSMOS_USERPATH	Set by Launcher to whatever Launcher determined the COSMOS_USERPATH to be. This ensures that all tools spawned by Launcher will user the same COSMOS_USERPATH
PATH	COSMOS adds the COSMOS gem's bin folder to the beginning of the PATH.

◀ BACK

NEXT ▶

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

Improve this page

- [Launcher Configuration](#)
- [TITLE](#)
- [TOOL_FONT](#)
- [LABEL_FONT](#)
- [LABEL](#)
- [DIVIDER](#)
- [NUM_COLUMNS](#)
- [TOOL](#)
- [MULTITOOL_START](#)
- [MULTITOOL_END](#)
 - [TOOL Modifiers](#)
 - [DONT_CAPTURE_IO](#)
 - [MULTITOOL_START Modifiers](#)
 - [TOOL](#)
 - [DELAY](#)
- [Example File](#)

Launcher

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.

To specify a different configuration file add ‘–config launcher.txt’ when starting the Launcher. For example, on Windows create a new Batch file at the top of your COSMOS configuration that looks like the following:

```
call tools\Launcher.bat --config launcher_config.txt
```

On Linux create a new executable file at the top of your COSMOS configuration that looks like the following:

```
ruby tools\Launcher --config launcher_config.txt
```

TITLE

Sets the title of the COSMOS Launcher

PARAMETER	DESCRIPTION	REQUIRED
Title	Launcher title. Default is "COSMOS Launcher".	True

TOOL_FONT

Sets the font used for tool buttons

!! Only set the TOOL_FONT once as the last encountered setting will apply to all button labels

PARAMETER	DESCRIPTION	REQUIRED
	The font family to use	
Font Family	Valid Values: 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	True
Font Size	The size of the font in standard points	True

LABEL_FONT

Sets the font used for labels

!! Only set the LABEL_FONT once as the last encountered setting will apply to all text labels

PARAMETER	DESCRIPTION	REQUIRED
	The font family to use	
Font Family	Valid Values: 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	True
Font Size	The size of the font in standard points	True

LABEL

Creates a label of text in the current font style

PARAMETER	DESCRIPTION	REQUIRED
Text	The text of the label	True

DIVIDER

Creates a horizontal line between tools

NUM_COLUMNS

Specifies how many launcher buttons should be created per row

PARAMETER	DESCRIPTION	REQUIRED
Columns	The number of launcher buttons per row before Launcher automatically creates a new row of buttons. Default is 4.	True

TOOL

Create a new tool launcher button

PARAMETER	DESCRIPTION	REQUIRED
Button Text	Label that is put on the button that launches the tool	True
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.	True
Icon Filename	Filename of a icon located in the data directory. Passing 'nil' or an empty string "" will result in Launcher using the default COSMOS icon.	False
Tool Parameters	Tool parameters as you would type on the command line. Specifying parameters here rather than in the 'Shell Command' parameter 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, e.g. --config filename.txt.	False
	Warning: The full configuration option name must be used rather than the short name. These parameters will override any parameters specified in the Shell Command.	

MULTITOOL_START

Creates a button which launches multiple tools

PARAMETER	DESCRIPTION	REQUIRED
Button Text	Label that is put on the button that launches the tools	True
Icon Filename	Filename of a icon located in the data directory. Passing 'nil' or an empty string "" will result in Launcher using the default COSMOS icon.	False

MULTITOOL_END

Ends the creation of a multi-tool button

TOOL Modifiers

The following keywords must follow a TOOL keyword.

DONT_CAPTURE_IO

Don't capture IO when running the command

MULTITOOL_START Modifiers

The following keywords must follow a MULTITOOL_START keyword.

TOOL

Create a new tool launcher button

PARAMETER	DESCRIPTION	REQUIRED
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.	True

DELAY

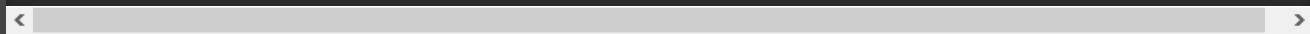
Inserts a delay between launching multiple tools

PARAMETER	DESCRIPTION	REQUIRED
Delay	Delay time in seconds	True

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"
```



◀ BACK NEXT ▶



Navigate the docs... ▾

Improve this page

- [AUTO_START](#)
- [AUTO_TARGET_COMPONENTS](#)
- [TARGET_COMPONENT](#)
- [COMPONENT](#)
 - [COMPONENT Modifiers](#)
 - [PACKET](#)

Data Viewer

This document describes Data Viewer configuration file parameters.

AUTO_START

Automatically start Data Viewer and connect to the Server

AUTO_TARGET_COMPONENTS

Automatically load all `data_viewer.txt` configuration files in each target

TARGET_COMPONENT

Load a Data Viewer configuration file in the specified target

PARAMETER	DESCRIPTION	REQUIRED
Target	Name of the target Valid Values: Any Target Name	True
Filename	The Data Viewer configuration file name. Defaults to <code>data_viewer.txt</code> .	False

COMPONENT

Declare a Data Viewer component

PARAMETER	DESCRIPTION	REQUIRED
Tab Name	Name of the component which shows up in the GUI tab	True
Filename	Name of the Ruby file which contains the component class	True

Options	Optional parameters that are sent to the component constructor	False
---------	--	-------

COMPONENT Modifiers

The following keywords must follow a COMPONENT keyword.

PACKET

Declare a packet to process by the component

PARAMETER	DESCRIPTION	REQUIRED
Target	Name of the target Valid Values: Any Target Name	True
Packet	Name of the packet	True

 BACK  NEXT 



Navigate the docs... ▾

Improve this page

- [PAGE](#)
- [TARGET_PAGES](#)
 - [PAGE Modifiers](#)
 - [NO_PDF](#)
 - [PDF_COVER](#)
 - [PDF_TOC](#)
 - [PDF_HEADER](#)
 - [PDF_FOOTER](#)
 - [PDF_TOP_MARGIN](#)
 - [PDF_BOTTOM_MARGIN](#)
 - [PDF_SIDE_MARGIN](#)
 - [TARGET](#)
 - [SECTION](#)
 - [CMD_SECTION](#)
 - [TLM_SECTION](#)

Handbook Creator

This document describes Handbook Creator configuration file parameters.

PAGE

Create a new top level webpage

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of the HTML file	True

TARGET_PAGES

Create a new webpage for each of the defined TARGETS

The page will be generated for each target. A top level “Targets” drop down will be created with links to all the generated target pages.

PARAMETER	DESCRIPTION	REQUIRED
Filename Postfix	The postfix will be applied to each generated target file. For example, a target named INST with a postfix of '_cmd_tlm.html' will result in a file named 'inst_cmd_tlm.html'.	True

PAGE Modifiers

The following keywords must follow a PAGE keyword.

NO_PDF

Indicates this page should not be generated when building the PDF

Handbook Creator can create webpages and export them to PDFs. Use this keyword when you have an index webpage or another page for which you do not want a PDF created.

PDF_COVER

Defines an HTML ERB template which is used to create the PDF cover page

The template is simply HTML tags which can reference 'title' set by this keyword from ERB context

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of the HTML ERB template file which is found in the config/tools/handbook_creator/templates directory.	True
Title Text	Title text which is accessible in the template	True

PDF_TOC

Insert the PDF table of contents

The table of contents is generated from the XSL templated found in config/tools/handbook_creator/default_toc.xsl

PDF_HEADER

Defines an HTML ERB template which is used to create the header on each PDF page

The template is simply HTML tags which can reference 'title' set by this keyword from ERB context

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of the HTML ERB template file which is found in the config/tools/handbook_creator/templates directory.	True
Title Text	Title text which is accessible in the template	True

PDF_FOOTER

Defines an HTML ERB template which is used to create the footer on each PDF page

The template is simply HTML tags which can reference 'title' set by this keyword from ERB context

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of the HTML ERB template file which is found in the config/tools/handbook_creator/templates directory.	True
Title Text	Title text which is accessible in the template	True

PDF_TOP_MARGIN

The margin from the top of the PDF page before the header

PARAMETER	DESCRIPTION	REQUIRED

Margin	The margin in pixels	True
--------	----------------------	------

PDF_BOTTOM_MARGIN

The margin below the footer to the bottom of the PDF page

PARAMETER	DESCRIPTION	REQUIRED
Margin	The margin in pixels	True

PDF_SIDE_MARGIN

The margin on both sides on the PDF page

PARAMETER	DESCRIPTION	REQUIRED
Margin	The margin in pixels	True

TARGET

Specify a target which this page is generated for

PARAMETER	DESCRIPTION	REQUIRED
Target	The target name Valid Values: Any Target Name	True

SECTION

Define a page SECTION

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of the HTML ERB template file which is found in the config/tools/handbook_creator/templates directory.	True
Output	What output formats should contain this section Valid Values: ALL, HTML, PDF	True
Title Text	Title text which is accessible in the template	False

CMD_SECTION

Define a page SECTION which is generated for all command packets

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of the HTML ERB template file which is found in the config/tools/handbook_creator/templates directory.	True
Output	What output formats should contain this section Valid Values: ALL, HTML, PDF	True
Title Text	Title text which is accessible in the template	False

TLM_SECTION

Define a page SECTION which is generated for all telemetry packets

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of the HTML ERB template file which is found in the config/tools/handbook_creator/templates directory.	True
Output	What output formats should contain this section Valid Values: ALL, HTML, PDF	True
Title Text	Title text which is accessible in the template	False

 BACK  NEXT

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

Improve this page

- [IGNORE_ITEM](#)
- [IGNORE_PACKET](#)
- [IGNORE_STALE](#)
- [COLOR_BLIND](#)
- [IGNORE_OPERATIONAL_LIMITS](#)

Limits Monitor

This document describes Limits Monitor configuration file parameters.

IGNORE_ITEM

Ignore a particular telemetry item

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the telemetry target Valid Values: Any Target Name	True
Packet Name	Name of the telemetry packet	True
Item Name	Name of the telemetry item	True

IGNORE_PACKET

Ignore all telemetry items for a specified packet

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the telemetry target Valid Values: Any Target Name	True
Packet Name	Name of the telemetry packet	True

IGNORE_STALE

Ignore the staleness of a specified packet

(Since 4.0.0)

Note that only the packet staleness is ignored. If the packet is received and an item is out-of-limits, that will still count against the overall system limits state unless the item or packet is separately ignored with the IGNORE_ITEM or IGNORE_PACKET keyword.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the telemetry target Valid Values: Any Target Name	True
Packet Name	Name of the telemetry packet	True

COLOR_BLIND

Use a color-blind accessibility mode

Color blind mode causes the value for out-of-limits items to have a code appended to indicate the limits status. The code is r = red low, y = yellow low, g = green low, B = blue, G = green or green high, Y = yellow high, R = red high.

IGNORE_OPERATIONAL_LIMITS

Ignore GREEN_HIGH or GREEN_LOW limits states

Items with operational limits will only show up in limits monitor if they go yellow or red

 BACK  NEXT 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

✍ Improve this page

- [LINE_DELAY](#)
- [MONITOR_LIMITS](#)
- [PAUSE_ON_RED](#)
- [Example File](#)

Script Runner

This document describes Script Runner configuration file parameters.

LINE_DELAY

Sets the amount of time in seconds before the next line of a script will be executed

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.	True

MONITOR_LIMITS

Log limits events to the Script Runner log file while a script is running

Limits events are always logged by the Command and Telemetry Server but are not put in the Script Runner log without this keyword.

PAUSE_ON_RED

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
```

 BACK

NEXT 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Custom Search



Proudly hosted by 



Navigate the docs... ▾

Improve this page

- [TABLEFILE](#)
- [TABLE](#)
 - [TABLE Modifiers](#)
 - [PARAMETER](#)
 - [APPEND_PARAMETER](#)
 - [PARAMETER Modifiers](#)
 - [FORMAT_STRING](#)
 - [UNITS](#)
 - [DESCRIPTION](#)
 - [META](#)
 - [REQUIRED](#)
 - [MINIMUM_VALUE](#)
 - [MAXIMUM_VALUE](#)
 - [DEFAULT_VALUE](#)
 - [STATE](#)
 - [WRITE_CONVERSION](#)
 - [POLY_WRITE_CONVERSION](#)
 - [SEG_POLY_WRITE_CONVERSION](#)
 - [GENERIC_WRITE_CONVERSION_START](#)
 - [GENERIC_WRITE_CONVERSION_END](#)
 - [OVERFLOW](#)
 - [HIDDEN](#)
 - [UNEDITABLE](#)
- [Example File](#)

Table Manager

This document describes Table Manager configuration file parameters.

TABLEFILE

Specify 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.	True

TABLE

Start a new table definition

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the table in quotes. The name will appear on the GUI tab.	True
Endianness	Indicates if the data in this table is in Big Endian or Little Endian format Valid Values: BIG_ENDIAN , LITTLE_ENDIAN	True
Display	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 Valid Values: ONE_DIMENSIONAL , TWO_DIMENSIONAL	False

When Display is ONE_DIMENSIONAL the remaining parameters are:

Description	Description of the table in quotes. The description is used in mouseover popups and status line information.	False
-------------	--	-------

When Display is TWO_DIMENSIONAL the remaining parameters are:

Rows	The number of rows in the table	False
Description	Description of the table in quotes. The description is used in mouseover popups and status line information.	False

TABLE Modifiers

The following keywords must follow a TABLE keyword.

PARAMETER

Defines a parameter in the current table

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the parameter. Must be unique within the table.	True
Bit Offset	Bit offset into the table of the Most Significant Bit of this parameter. May be negative to indicate an offset from the end of the table. Always use a bit offset of 0 for derived parameters.	True
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 is 0 and Bit Size is 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	True
Data Type	Data Type of this parameter Valid Values: INT , UINT , FLOAT , DERIVED , STRING , BLOCK	True

When Data Type is INT, UINT, FLOAT, DERIVED the remaining parameters are:

Minimum Value	Minimum allowed value for this parameter	True

Maximum Value	Maximum allowed value for this parameter	True
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.	True
Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

When Data Type is STRING, BLOCK the remaining parameters are:

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.	True
Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

APPEND_PARAMETER

Defines a parameter in the current table

PARAMETER	DESCRIPTION	REQUIRED
Name	Name of the parameter. Must be unique within the table.	True
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 is 0 and Bit Size is 0 then this is a derived parameter and the Data Type must be set to 'DERIVED'.	True
Data Type	Data Type of this parameter Valid Values: INT, UINT, FLOAT, DERIVED, STRING, BLOCK	True

When Data Type is INT, UINT, FLOAT, DERIVED the remaining parameters are:

Minimum Value	Minimum allowed value for this parameter	True
Maximum Value	Maximum allowed value for this parameter	True
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.	True
Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN, LITTLE_ENDIAN	False

When Data Type is STRING, BLOCK the remaining parameters are:

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.	True
---------------	---	------

Description	Description for this parameter which must be enclosed with quotes	False
Endianness	Indicates if the data in this command is to be sent in Big Endian or Little Endian format Valid Values: BIG_ENDIAN , LITTLE_ENDIAN	False

PARAMETER Modifiers

The following keywords must follow a PARAMETER keyword.

FORMAT_STRING

Adds printf style formatting

PARAMETER	DESCRIPTION	REQUIRED
Format	How to format using printf syntax. For example, '0x%0X' will display the value in hex.	True

Example Usage:

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

UNITS

Add displayed units

PARAMETER	DESCRIPTION	REQUIRED
Full Name	Full name of the units type, e.g. Celcius	True
Abbreviated	Abbreviation for the units, e.g. C	True

Example Usage:

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

DESCRIPTION

Override the defined description

PARAMETER	DESCRIPTION	REQUIRED
Value	The new description	True

META

Stores custom user metadata

Meta data is user specific data that can be used by custom tools for various purposes. One example is to store additional information needed to generate source code header files.

PARAMETER	DESCRIPTION	REQUIRED
Meta Name	Name of the metadata to store	True
Meta Values	One or more values to be stored for this Meta Name	False

Example Usage:

META TEST "This parameter is for test purposes only"

REQUIRED

Parameter is required to be populated in scripts

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.

MINIMUM_VALUE

Override the defined minimum value

PARAMETER	DESCRIPTION	REQUIRED
Value	The new minimum value for the parameter	True

MAXIMUM_VALUE

Override the defined maximum value

PARAMETER	DESCRIPTION	REQUIRED
Value	The new maximum value for the parameter	True

DEFAULT_VALUE

Override the defined default value

PARAMETER	DESCRIPTION	REQUIRED
Value	The new default value for the parameter	True

STATE

Defines a key/value pair for the current command parameter

Key value pairs allow for user friendly strings. 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 string state name	True
Value	The numerical state value	True
Hazardous	Indicates the state is hazardous. This will cause a popup to ask for user confirmation when sending this command. Valid Values: HAZARDOUS	False
Hazardous Description	String describing why this state is hazardous	False

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

Applies a conversion when writing the current command parameter

Conversions are 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 REQUIRE). 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'.	True
Parameter	Additional parameter values for the conversion which are passed to the class constructor.	False

Example Usage:

```
WRITE_CONVERSION the_great_conversion.rb 1000
```

Defined in the _great_conversion.rb:

```

require 'cosmos/conversions/conversion'
module Cosmos
  class TheGreatConversion < Conversion
    def initialize(multiplier)
      super()
      @multiplier = multiplier.to_f
    end
    def call(value, packet, buffer)
      return value * multiplier
    end
  end
end

```

POLY_WRITE_CONVERSION

Adds a polynomial conversion factor to the current command parameter

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
C0	Coefficient	True

Cx	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.	False
----	--	-------

Example Usage:

```
POLY_WRITE_CONVERSION 10 0.5 0.25
```

SEG_POLY_WRITE_CONVERSION

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.	True
C0	Coefficient	True
Cx	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.	False

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

Start a generic write conversion

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.



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.

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
```

GENERIC_WRITE_CONVERSION_END

Complete a generic write conversion

OVERFLOW

Set the behavior when writing a value overflows the type

By default COSMOS throws an error if you try to write a value which overflows its specified type, e.g. writing 255 to a 8 bit signed value. Setting the overflow behavior also allows for COSMOS to 'TRUNCATE' the value by eliminating any high order bits. You can also set 'SATURATE' which causes COSMOS to replace the value with the maximum or minimum allowable value for that type. Finally you can specify 'ERROR_ALLOW_HEX' which will allow for a maximum hex value to be written, e.g. you can successfully write 255 to a 8 bit signed value.

PARAMETER	DESCRIPTION	REQUIRED
Behavior	How COSMOS treats an overflow value. Only applies to signed and unsigned integer data types. Valid Values: <code>ERROR</code> , <code>ERROR_ALLOW_HEX</code> , <code>TRUNCATE</code> , <code>SATURATE</code>	True

Example Usage:

```
OVERFLOW TRUNCATE
```

HIDDEN

Indicates that the parameter should not be shown to the user in the Table Manager GUI

Hidden parameters still exist and will be saved to the resulting binary. This is useful for padding and other essential but non-user editable fields.

UNEDITABLE

Indicates that the parameter should be shown to the user but not editable.

Uneditable parameters are useful for control fields which the user may be interested in but should not be able to edit.

Example File

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

```

# Define tables by: TABLE, name, description, endian
#   name and description are strings contained in double quotes
#   endian is either BIG_ENDIAN or LITTLE_ENDIAN
TABLE "Master Table Map" "The one table to rule them all" ONE_DIMENSIONAL BIG_ENDIAN 1
    # Each element in a UNIQUE table is defined as follows:
    #   name, description, type, bit size, display type, min, max, default
    #   type must be INT or UNIT
    #   display type must be DEC, HEX, STATE
    #       add -U to make DEC or HEX uneditable, i.e. DEC-U or HEX-U
    #   if min or max are too large for the type they will be set to the types max
PARAMETER "Param1" "The first parameter" INT 32 STATE 0 1 0
    STATE DEFAULT 0
    STATE USER 1
PARAMETER "Param2" "The second parameter" INT 32 STATE 0 1 1
    STATE DEFAULT 0
    STATE USER 1
PARAMETER "Param3" "The third parameter" STRING 80 STRING ""
PARAMETER "Param4" "The fourth parameter" UINT 8 HEX MIN MAX MAX
PARAMETER "PAD" "Unused padding" INT 576 HEX-U 0 0 0

TABLE "Trailer" "Data appended to a table file" ONE_DIMENSIONAL BIG_ENDIAN 2
PARAMETER "File ID" "Uneditable file id" UINT 16 DEC-U 0 65535 4
PARAMETER "Version ID" "User defined version id" UINT 16 DEC MIN_UINT16 MAX_UINT16 1
PARAMETER "CRC32" "Auto-generated CRC" UINT 32 HEX-U MIN MAX 0

```

 BACK  NEXT

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

✍ Improve this page

- [LINE_DELAY](#)
- [MONITOR_LIMITS](#)
- [PAUSE_ON_RED](#)
- [LOAD.Utility](#)
- [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_METADATA](#)
 - [Example File](#)

Test Runner

This document describes Test Runner configuration file parameters.

LINE_DELAY

Sets the amount of time in seconds before the next line of a script will be executed

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.	True

MONITOR_LIMITS

Log limits events to the Script Runner log file while a script is running

Limits events are always logged by the Command and Telemetry Server but are not put in the Script Runner log without this keyword.

PAUSE_ON_RED

Pause a running script if a red limit occurs

LOAD.Utility

Specify a test procedure to run

Procedures 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. This keyword is used to load the script and execute it line by line showing the current execution point. Sometimes you want to load utility code that is NOT executed line by line but is executed in the background. For example, a CRC routine or other long running calculation. The Ruby keyword 'load' can be used for this purpose, e.g. 'load utility.rb'. This code will be loaded but not shown when executing.

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of the test file in quotes	True

REQUIRE.Utility

DEPRECATED in favor of LOAD.Utility

RESULTS_WRITER

Specify a different Ruby file to interpret and print the Test Runner results

The specified Ruby file must define a class which implements the Cosmos::ResultsWriter API

PARAMETER	DESCRIPTION	REQUIRED
Filename	Name of the Ruby file which implements a result writer	True
Class Parameters	Parameters to pass to the constructor of the results writer	False

ALLOW_DEBUG

Whether to allow the user to enable the debug line where the user can enter arbitrary statements

PAUSE_ON_ERROR

Set or clear 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
Enable	Whether to pause when the script encounters an error Valid Values: TRUE, FALSE	True

CONTINUE_TEST_CASE_AFTER_ERROR

Set or clear 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
Enable	Whether to continue the test case when the script encounters an error Valid Values: TRUE, FALSE	True

ABORT_TESTING_AFTER_ERROR

Set or clear 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
Enable	Whether to continue to the next test case when the script encounters an error Valid Values: TRUE, FALSE	True

MANUAL

Set the \$manual global variable for all executing scripts

The \$manual 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. This capability is completely dependent on user specific code which checks the \$manual variable.

PARAMETER	DESCRIPTION	REQUIRED
Enable	Whether to set the \$manual global to true Valid Values: TRUE, FALSE	True

LOOP_TESTING

Set or clear 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
Enable	Whether to loop the selected test level Valid Values: TRUE, FALSE	True

BREAK_LOOP_AFTER_ERROR

Set or clear 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
Enable	Whether to break the loop after encountering an error Valid Values: TRUE, FALSE	True

IGNORE_TEST

Ignore 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	True

IGNORE_TEST_SUITE

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 tests	True

CREATE_DATA_PACKAGE

Creates a data package of every file created during the test

AUTO_CYCLE_LOGS

Automatically start a new server message log and cmd/tlm logs at the beginning and end of each test. Typically used in combination with CREATE_DATA_PACKAGE.

COLLECT_METADATA

Prompt for Meta Data before starting tests

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
```

 BACK  NEXT 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

✍ Improve this page

- [DELIMITER](#)
- [FILL_DOWN](#)
- [DOWNSAMPLE_SECONDS](#)
- [MATLAB_HEADER](#)
- [UNIQUE_ONLY](#)
- [UNIQUE_IGNORE](#)
- [SHARE_COLUMNS](#)
- [SHARE_COLUMN](#)
- [FULL_COLUMN_NAMES](#)
- [DONT_OUTPUT_Filenames](#)
- [TEXT](#)
- [ITEM](#)
- [Example File](#)

Telemetry Extractor

This document describes Telemetry Extractor configuration file parameters.

DELIMITER

Specify an alternative column delimiter over the default tab character

PARAMETER	DESCRIPTION	REQUIRED
Delimiter	Character or string to use as a delimiter. For example ‘,’.	True

FILL_DOWN

Insert a value into every row of the output

Telemetry Grapher produces an output file with columns for each item name. If you are graphing items from different packets, only the columns with items from that packet are populated with values. Fill down causes a particular item column to be populated with the previous value so every column is ‘filled’. This makes it easier to graph multiple values across packets in Excel.

DOWNSAMPLE_SECONDS

Downsample data to only output a value every X seconds of time

PARAMETER	DESCRIPTION	REQUIRED
Seconds		True

MATLAB_HEADER

Prepend the Matlab comment symbol of ‘%’ to the header lines in the output file

UNIQUE_ONLY

Only output a row if one of the extracted values has changed

This keyword is useful to extract telemetry items over a large time period by only outputting those values where items have changed

UNIQUE_IGNORE

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. Use this keyword if you have a similar telemetry item that you want to display in the output but not be used to determine uniqueness.

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

SHARE_COLUMNS

Export telemetry items with the same name into the same column

Normally items from different packets are put in their own column in the output. If this keyword is used, items with the same name in different packets will now share one column in the output. This applies to all telemetry items with identical names. If you want to share columns for only specific items, use the SHARE_COLUMN keyword instead.

SHARE_COLUMN

Export telemetry items with the same name into the same column

(Since 3.9.2)

Normally items from different packets are put in their own column in the output. This keyword is used to specify specific items that are from different packets but should share one column in the output. This applies to all telemetry items with identical names. If you want to share columns for all duplicate item names in different packets, use the SHARE_COLUMNS keyword instead.

PARAMETER	DESCRIPTION	REQUIRED
Item Name	Name of the telemetry item	True
Item Type	Type of the telemetry item Valid Values: CONVERTED, RAW, FORMATTED, WITH_UNITS	False

FULL_COLUMN_NAMES

Use TARGET PACKET ITEM as the column header for each column

Normally just the item name is used as the column header but this adds the target and packet name. Note that the TARGET and PACKET columns are not generated when this mode is active since the items are fully qualified.

DONT_OUTPUT_FILERAMES

Don't output the list of input filenames at the top of each output file

TEXT

Insert arbitrary text in the Telemetry Extractor output

Text also allows you to dynamically create Excel formulas using a special syntax

PARAMETER	DESCRIPTION	REQUIRED
Header	The column header text	True
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 typically the TARGET and PACKET and telemetry items start in column 'C' in Excel.	True

ITEM

Specify a telemetry item to extract

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the telemetry target Valid Values: Any Target Name	True
Packet Name	Name of the telemetry packet	True
Item Name	Name of the telemetry item	True
Item Type	Type of the telemetry item Valid Values: CONVERTED, RAW, FORMATTED, WITH_UNITS	False

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_FILERAMES
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
```

BACK

NEXT

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

✍ Improve this page

- [AUTO_TARGETS](#)
- [AUTO_TARGET](#)
- [NEW_COLUMN](#)
- [TARGET](#)
- [ADD_SHOW_ON_STARTUP](#)
- [GROUP](#)
 - [TARGET Modifiers](#)
 - [SCREEN](#)
 - [SCREEN Modifiers](#)
 - [SHOW_ON_STARTUP](#)
 - [GROUP Modifiers](#)
 - [GROUP_SCREEN](#)

Telemetry Viewer

This document describes Telemetry Viewer configuration file parameters.

For documentation on how to build Telemetry Screens and how to configure the screen widgets please see the [Telemetry Screens](#).

AUTO_TARGETS

Add all screens defined in the screens directory of each target folder

Automatically added 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.

AUTO_TARGET

Add all screens defined in the screens directory of the specified target folder

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

PARAMETER	DESCRIPTION	REQUIRED
-----------	-------------	----------

Target	Name of the target Valid Values: Any Target Name	True
--------	--	------

NEW_COLUMN

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

Used in conjunction with the SCREEN keyword to define individual screens within a target's screen directory.

PARAMETER	DESCRIPTION	REQUIRED
Target	Name of the target Valid Values: Any Target Name	True

ADD_SHOW_ON_STARTUP

Adds show on startup to any screen that has already been defined

Screens that are discovered by AUTO_TARGETS or AUTO_TARGET aren't explicitly defined. Thus this keyword is used to add show on startup.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Target Name of the screen Valid Values: Any Target Name	True
Screen Name	Name of the screen. This is equal to the screen's filename without the .txt extension.	True
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.	False
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.	False

GROUP

Create a new drop down group of screens in the GUI

PARAMETER	DESCRIPTION	REQUIRED
Group Name	The text to display in front of the drop down list of screens	True

TARGET Modifiers

The following keywords must follow a TARGET keyword.

SCREEN

Adds the specified screen from the specified target

PARAMETER	DESCRIPTION	REQUIRED
File Name	Name of the file containing the telemetry screen definition. The filename will be upcased and used in the GUI drop down selection.	True
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.	False
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.	False

SCREEN Modifiers

The following keywords must follow a SCREEN keyword.

SHOW_ON_STARTUP

Causes the previously defined SCREEN to be automatically displayed when Telemetry Viewer starts

GROUP Modifiers

The following keywords must follow a GROUP keyword.

GROUP_SCREEN

Define a screen in the given group drop down

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Target Name of the screen Valid Values: Any Target Name	True
Screen Name	Name of the screen. This is equal to the screen's filename without the .txt extension.	True
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.	False
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.	False

 BACK  NEXT 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

✍ Improve this page

- [Definitions](#)
- [Telemetry Viewer Configuration](#)
- [Telemetry Screen Definition Files](#)
- [Keywords:](#)
 - [SCREEN](#)
 - [END](#)
 - [STAY_ON_TOP](#)
 - [GLOBAL_SETTING](#)
 - [GLOBAL_SUBSETTING](#)
 - [SETTING](#)
 - [SUBSETTING](#)
 - [NAMED_WIDGET](#)
 - [WIDGETNAME](#)
- [Example File](#)
- [Telemetry Viewer Settings Files](#)
- [Keywords:](#)
 - [AUTO_TARGETS](#)
 - [AUTO_TARGET](#)
 - [NEW_COLUMN](#)
 - [TARGET](#)
 - [SCREEN](#)
 - [GROUP](#)
 - [GROUP_SCREEN](#)
 - [SHOW_ON_STARTUP](#)
 - [ADD_SHOW_ON_STARTUP](#)
- [Example File](#)
- [Widget Descriptions](#)
- [Layout Widgets](#)
 - [VERTICAL](#)
 - [VERTICALBOX](#)
 - [HORIZONTAL](#)
 - [HORIZONTALBOX](#)
 - [MATRIXBYCOLUMNS](#)
 - [SCROLLWINDOW](#)
 - [TABBOOK](#)

- TABITEM
- Decoration Widgets
 - LABEL
 - HORIZONTALLINE
 - SECTIONHEADER
 - TITLE
 - SPACER
 - STRETCH
- Telemetry widgets
 - ARRAY
 - BLOCK
 - FORMATFONTVALUE
 - FORMATVALUE
 - LABELFORMATVALUE
 - LABELPROGRESSBAR
 - LABELTRENDLIMITSBAR
 - LABELVALUE
 - LABELVALUEDESC
 - LABELVALUELIMITSBAR
 - LABELVALUELIMITSCOLUMN
 - LABELVALUERANGEBAR
 - LABELVALUERANGECOLUMN
 - LIMITSBAR
 - LIMITSCOLUMN
 - LIMITSCOLOR
 - VALUELIMITSBAR
 - VALUELIMITSCOLUMN
 - VALUERANGEBAR
 - VALUERANGECOLUMN
 - LINEGRAPH
 - PROGRESSBAR
 - RANGEBAR
 - RANGECOLUMN
 - TEXTBOX
 - TIMEGRAPH
 - TRENDBAR
 - TRENDLIMITSBAR
 - VALUE
- Interactive Widgets
 - BUTTON
 - CHECKBUTTON
 - COMBOBOX
 - RADIOBUTTON
 - TEXTFIELD
 - SCREENSHOTBUTTON
- Canvas Widgets
 - CANVAS

- [CANVASLABEL](#)
- [CANVASLABELVALUE](#)
- [CANVASIMAGE](#)
- [CANVASIMAGEVALUE](#)
- [CANVASLINE](#)
- [CANVASLINEVALUE](#)
- [CANVASDOT](#)
- [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](#)

Telemetry Screens

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
Fixed	Force the window to be fixed size and not user resizable	No

Example Usage:

```
SCREEN AUTO AUTO 1.0 FIXED
```

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.

STAY_ON_TOP

The STAY_ON_TOP keyword is used to force the screen to the front of the display stack. This forces the window to stay above ALL other windows including other applications not associated with COSMOS.

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 labelval
```

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: <:userpath>/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 .'
        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.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target directory to look for screens.	Yes

Example Usage:

```
AUTO_TARGET COSMOS
```

NEW_COLUMN

The NEW_COLUMN keyword 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

GROUP

The GROUP keyword is used to create a new drop down group in the Tlm Viewer application.

PARAMETER	DESCRIPTION	REQUIRED
Group Name	Label to display in front of the group drop down.	Yes

Example Usage:

```
GROUP "Special Ops"
```

GROUP_SCREEN

The GROUP_SCREEN keyword is used to add a screen to a previously defined group. It must follow the GROUP keyword.

PARAMETER	DESCRIPTION	REQUIRED
Target Name	Name of the target where the screen is defined	Yes
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:

```
GROUP "Special Ops"  
GROUP_SCREEN SYSTEM status.txt
```

SHOW_ON_STARTUP

The SHOW_ON_STARTUP keyword causes the previously defined SCREEN to be automatically displayed when Telemetry Viewer starts. It must be preceded by the SCREEN or GROUP_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: <:userpath>/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"
GROUP "My group"
    GROUP_SCREEN SYSTEM "status.txt"
    GROUP_SCREEN INST "hs.txt"
    GROUP_SCREEN INST2 "hs.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 = 3)	No
Vertical Packing	Pack all widgets vertically (default = true)	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 = 3)	No
Vertical Packing	Pack all widgets vertically (default = true)	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 = 0)	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
Horizontal Spacing	Spacing between horizontal items (default = 0)	No
Vertical Spacing	Spacing between vertical items (default = 0)	No

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 display 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"

```

SPACER

The SPACER widget inserts a spacer into a layout. This can be used to separate or align other widgets. For more information about how the widget size policy works please see the [QSizePolicy::Policy](#).

PARAMETER	DESCRIPTION	REQUIRED
Width	The width of the spacer in pixels.	Yes
Height	The height of the spacer in pixels.	Yes
Horizontal Policy	The horizontal size policy of the spacer. Can be FIXED, MINIMUM, MAXIMUM, PREFERRED, EXPANDING, MINUMEXPANDING, or IGNORED. Defaults to MINIMUM.	No
Vertical Policy	The vertical size policy of the spacer. Can be FIXED, MINIMUM, MAXIMUM, PREFERRED, EXPANDING, MINUMEXPANDING, or IGNORED. Defaults to MINIMUM.	No

Example Usage:

```

VERTICAL 3 FALSE
LABEL "Spacer below"
SPACER 0 100 MINIMUM EXPANDING
LABEL "Spacer above"
END

```

STRETCH

The STRETCH widget inserts stretch into a layout. Stretch expands to the end of the layout to help align other widgets in the layout.

PARAMETER	DESCRIPTION	REQUIRED
Stretch Factor	Multiple stretch items can expand at different rates. By default stretch is added with value 1 but stretch is allocated according to the stretch factor.	Yes

Example Usage:

```

VERTICAL 3 FALSE
LABEL "Stretch below"
STRETCH
LABEL "Stretch above"
END

```

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
Font Weight	The font weight. See QFont::Weight for more information. (default = Qt::Font::Normal)	No
Font Italics	Whether to display the font in italics. (default = false)	No

Example Usage:

```
FORMATFONTVALUE INST LATEST TIMESEC %012u CONVERTED 12 arial 15 Qt::Font::Bold true
```

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
```

LABELFORMATVALUE

The LABELFORMATVALUE widget displays a label with a value box 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:

```
LABELFORMATVALUE 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 TREND BAR 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
Alignment	How to align the label and value items. Options are 'split', 'right', 'left', 'center' (default = split)	No

Example Usage:

```
LABELVALUE INST LATEST TIMESEC CONVERTED 18 center
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
```

LABELVALUELIMITSCOLUMN

The LABELVALUELIMITSCOLUMN 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 LIMITSCOLUMN 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:

```
LABELVALUELIMITSCOLUMN INST HEALTH_STATUS TEMP1 CONVERTED 18
LABELVALUELIMITSCOLUMN INST HEALTH_STATUS TEMP1
```

LABELVALUERANGEBAR

The LABELVALUERANGEBAR 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 RANGEBAR 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
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 = WITH_UNITS)	No
Number of Characters	The number of characters wide to make the value box (default = 12)	No
Width	Width of the range bar (default = 160)	No
Height	Height of the range bar (default = 25)	No

Example Usage:

```
LABELVALUERANGEBAR INST HEALTH_STATUS TEMP1 0 50 CONVERTED 18 200 50
LABELVALUERANGEBAR INST HEALTH_STATUS TEMP1 0 50
```

LABELVALUERANGECOLUMN

The LABELVALUERANGECOLUMN 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 RANGECOLUMN 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
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 = WITH_UNITS)	No
Number of Characters	The number of characters wide to make the value box (default = 8)	No
Width	Width of the range bar (default = 30)	No
Height	Height of the range bar (default = 100)	No

Example Usage:

```
LABELVALUERANGECOLUMN INST HEALTH_STATUS TEMP1 0 50 CONVERTED 18 50 200
LABELVALUERANGECOLUMN INST HEALTH_STATUS TEMP1 0 50
```

LIMITSBAR

The LIMITSBAR widget displays a graphical representation of where an items value falls withing its limits ranges horizontally.

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
```

LIMITSCOLUMN

The LIMITSCOLUMN widget displays a graphical representation of where an items value falls withing its limits ranges vertically.

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 = 30)	No
Height	The height of the range bar (default = 100)	No

Example Usage:

```
LIMITSCOLUMN INST HEALTH_STATUS TEMP1 CONVERTED 50 200
LIMITSCOLUMN INST HEALTH_STATUS TEMP1
```

LIMITSCOLOR

The LIMITSCOLOR widget displays a stoplight-like circle depicting the limits color of an item

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
Radius	Radius of the circle (default = 10 pixels)	No
Full Item Name	Show the full item name (default = false)	No

Example Usage:

```
LIMITSCOLOR INST HEALTH_STATUS TEMP1 CONVERTED 20 TRUE
LIMITSCOLOR INST HEALTH_STATUS TEMP1
```

VALUELIMITSBAR

The VALUELIMITSBAR widget displays a graphical representation of where an items value falls withing its limits ranges horizontally and its value in a VALUE 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:

```
VALUELIMITSBAR INST HEALTH_STATUS TEMP1 CONVERTED 18
VALUELIMITSBAR INST HEALTH_STATUS TEMP1
```

VALUELIMITCOLUMN

The VALUELIMITCOLUMN widget displays a graphical representation of where an items value falls withing its limits ranges vertically and its value in a VALUE 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 = 8)	No

Example Usage:

```
VALUELIMITCOLUMN INST HEALTH_STATUS TEMP1 CONVERTED 18  
VALUELIMITCOLUMN INST HEALTH_STATUS TEMP1
```

VALUERANGEBAR

The VALUERANGEBAR widget displays a graphical representation of where an items value falls withing a range horizontally and its value in a VALUE 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
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 = WITH_UNITS)	No
Number of Characters	The number of characters wide to make the value box (default = 12)	No
Width	Width of the range bar (default = 160)	No
Height	Height of the range bar (default = 25)	No

Example Usage:

```
VALUERANGEBAR INST HEALTH_STATUS TEMP1 0 100 CONVERTED 18 200 50  
VALUERANGEBAR INST HEALTH_STATUS TEMP1 -1000 1000
```

VALUERANGECOLUMN

The VALUERANGECOLUMN widget displays a graphical representation of where an items value falls withing a range vertically and its value in a VALUE 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
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 = WITH_UNITS)	No
Number of Characters	The number of characters wide to make the value box (default = 8)	No
Width	Width of the range bar (default = 30)	No
Height	Height of the range bar (default = 100)	No

Example Usage:

```
VALUERANGECOLUMN INST HEALTH_STATUS TEMP1 0 100 CONVERTED 18 50 200
VALUERANGECOLUMN INST HEALTH_STATUS TEMP1 -1000 1000
```

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 horizontally.

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
```

RANGECOLUMN

The RANGECOLUMN widget displays a graphical representation of where an items value falls withing a range vertically.

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 = 30)	No
Height	Height of the range bar (default = 100)	No

Example Usage:

```
RANGECOLUMN INST HEALTH_STATUS TEMP1 0 100 CONVERTED 50 200
RANGECOLUMN 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
Value Type	The type of the value to display: RAW, CONVERTED, FORMATTED, or WITH_UNITS (default = CONVERTED)	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

Point Size	Size of the point in pixels (default = 5)	No
Time Item Name	The telemetry item to use as the time on the X axis (default = PACKET_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 (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 to execute a command:

```
BUTTON 'Start Collect' 'cmd("INST COLLECT with TYPE NORMAL, DURATION 5")'
```

Example Usage to open Script Runner with a given script:

```
BUTTON "Run Script" 'system("ruby #{Cosmos::USERPATH}/tools/ScriptRunner #{Cosmos::USERPATH}/procedu
```

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
Checked	Whether the checkbox should initially be checked. Valid values are 'CHECKED' or 'UNCHECKED'. (default = 'UNCHECKED')	No

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
```

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 1	Text to display in the selection drop down	Yes
Option Text n	Text to display in the selection drop down	No

Example Usage:

```
BUTTON 'Start Collect' 'cmd("INST COLLECT with TYPE #{get_named_widget("COLLECT_TYPE").text}, DURATION
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

Checked	Whether the radio button should initially be checked. Valid values are 'CHECKED' or 'UNCHECKED'. (default = 'UNCHECKED')	No
---------	--	----

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")}).start()'
```

SCREENSHOTBUTTON

The SCREENSHOTBUTTON widget displays a button that when clicked takes a screenshot.

PARAMETER	DESCRIPTION	REQUIRED
Button Text	Text to display on the button. (default = "Screenshot")	No
Directory	Directory to save the screenshot. (default = System.paths['LOGS'])	No

Example Usage:

```
SCREENSHOTBUTTON "Screenshot" "C:/images/screenshots"
```

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 (default = true)	No
Frame Width	Width in pixels of the frame (default = 3)	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 <:userpath>/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 telemetry point is considered on (default = 'green')	No
Color Off	Color of the line when the telemetry 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
```

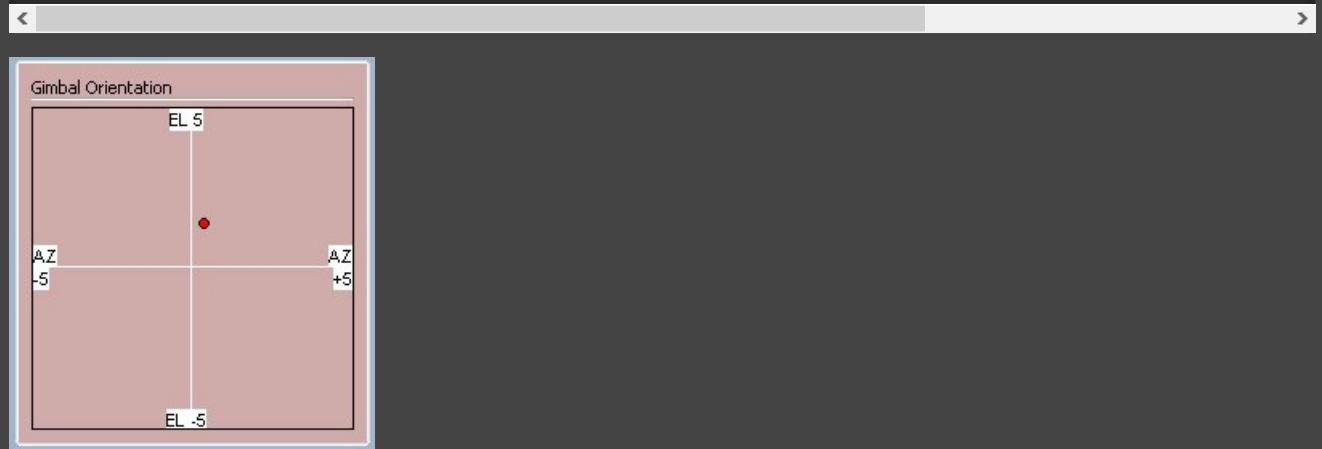
CANVASDOT

The CANVASDOT widget draws a dot onto the canvas, and it can be programmed to change its position with ruby code.

PARAMETER	DESCRIPTION	REQUIRED
X or Str to Eval	X position of the dot, or it can be a string of ruby code	Yes
Y or Str to Eval	Y position of the dot, or it can be a string of ruby code	Yes
Color	Color of the dot (default = black)	No
Width	Width of the dot in pixels (default = 3)	No

Example Usage:

```
CANVAS 201 201
CANVASLINE 0 0 200 0
CANVASLINE 200 0 200 200
CANVASLINE 200 200 0 200
CANVASLINE 0 200 0 0
CANVASLINE 99 1 99 199 white
CANVASLINE 1 99 199 99 white
CANVASDOT 'tlm_variable("GIMBAL AXIS_STATUS_X POSITION", :RAW) + 100' 'tlm_variable("GIMBAL AXIS_S'
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		
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 width 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 widget's 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 their 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

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

Improve this page

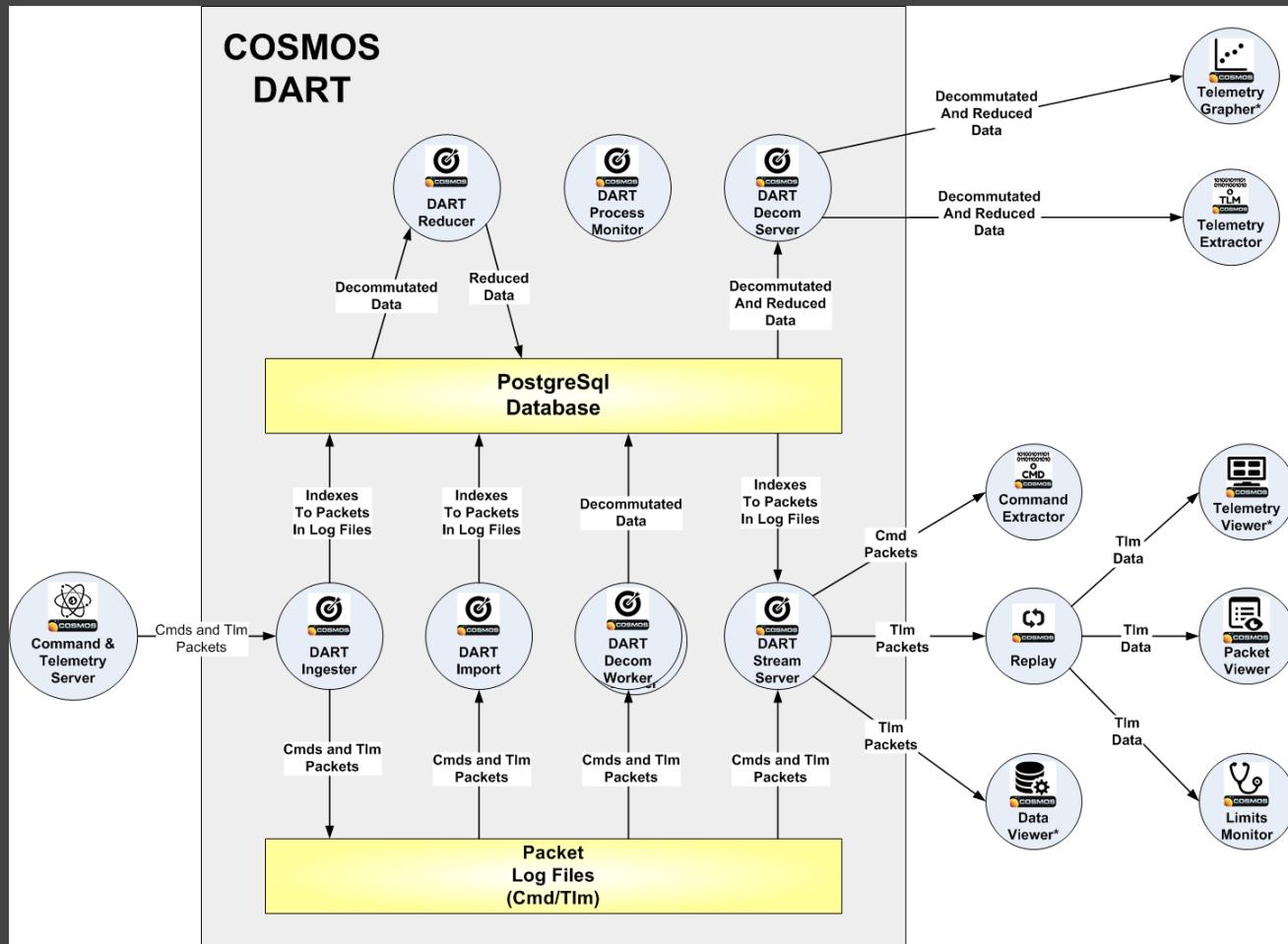
- [Overall Architecture and Context Diagram](#)
- [Using the dart_import utility](#)

DART Overview

DART (Data Archival Retrieval Trending) is the COSMOS tool to support long term storage and trending of command and telemetry data. It was built on top of the open source PostgreSQL database to ensure rapid retrieval of data over long periods. Data is reduced at time periods of every minute, hour, and day to allow for increased performance of long queries. With this reduction, to return a reduced year's worth of data the database simply has to return 365 telemetry items. The original data is always preserved for indepth analysis over specific time periods.

Overall Architecture and Context Diagram

The following diagram shows how DART integrates into a COSMOS System:



Key aspects of this architecture:

- DART Contains 7 different applications
 - DART Process Monitor - Cleans the database and starts the next five applications / monitors aliveness
 - DART Ingester - Connects to the COSMOS CmdTlmServer and receives realtime cmd/tlm packets
 - DART Workers - Decommutates packets and save the decommutated data into the PostgreSQL database
 - DART Reducer - Reduces decommutated data into minute/hour/day reduced data sets
 - DART Stream Server - Serves a stream of raw packets from DART on request
 - DART Decom Server - Provides a JSON formatted array of decommutated or reduced data for an item on request
 - DART Import - Command line tool to import previously logged data into DART
- The COSMOS Command and Telemetry Server nominally streams data to DART in realtime. This keeps the data in DART as fresh as possible
- Alternatively, or after an outage, data can be ingested into DART using the `dart_import` utility
- TlmGrapher and TlmExtractor can pull decommutated or reduced data from the DART Decom Server
- Replay, DataViewer, and CmdExtractor can pull a stream of raw packets from the DART Stream Server
- Other telemetry tools can receive data from DART through Replay
- Raw packets are stored in DART in normal COSMOS binary packet log files. The PostgreSQL database keeps indexes into these files that point to each packet for quick random access.
- Decommutated and Reduced data sets are stored directly in the PostgreSQL database
- Only integer and floating point data types are reduced into averages, minimum, maximum, and standard deviation at minute/hour/day granularities.

Using the `dart_import` utility

Dart Import is used to import COSMOS command and telemetry packet log files into the DART system. This is useful for importing older data that was never imported for whatever reason, or if DART was offline when the data was collected. Important: These log files are imported in place and become part of the DART database. The files must be placed into the DART data folder (defaults to `outputs/dart/data`) before they can be imported. Note that the data requires a SYSTEM META packet, and therefore generally only supports data from COSMOS 4.1.1+ unless the data is massaged first.

Usage:

```
dart_import <filename> [--force]
```

The `--force` flag can be used to force data to be reimported into the database even if the tool determines that all of the data is already likely in place. It still performs an algorithm to prevent duplicate data from being inserted into the database, but will check every single packet to make sure it is already in the database.

◀ BACK

NEXT ▶

Proudly hosted by



Navigate the docs... ▾

Improve this page

- Other possible dependencies
- Changing the DART Data File Directory
- Shutting Down DART

DART Installation

1. DART requires at least COSMOS version 4.3.0 although you should always install the latest to get important bug fixes and feature updates. Install the latest COSMOS by running:

```
bundle update cosmos
```

2. Install the PostgreSQL database. DART has been tested with both version 9.x and 10.x so we recommend the latest version unless you already have a 9.x instance. When installing PostgreSQL you must create an overall superuser with a username and password. Once you have PostgreSQL installed with the superuser created, follow the following steps to configure for DART. (Note that on linux you will need to install postgresql and the postgresql-server-dev (or equivalent) packages)
3. If you are upgrading an existing COSMOS project perform the following steps.
 1. Somewhere else on your computer create a new COSMOS demo project:

```
cosmos demo demo
```

2. Copy config/dart/Gemfile from the demo project to your COSMOS project
3. Copy config/targets/DART from the demo project to your COSMOS project
4. Add these lines to your config/system/system.txt file:

```
DECLARE_TARGET DART
LISTEN_HOST DART_STREAM 0.0.0.0 # 127.0.0.1 is more secure if you don't need external conn
LISTEN_HOST DART_DECOM 0.0.0.0 # 127.0.0.1 is more secure if you don't need external conne
CONNECT_HOST DART_STREAM 127.0.0.1
CONNECT_HOST DART_DECOM 127.0.0.1
PORT DART_STREAM 8777
PORT DART_DECOM 8779
PATH DART_DATA ./outputs/dart/data
PATH DART_LOGS ./outputs/dart/logs
```

5. Add the following line to config/tools/launcher.txt:

```
TOOL "DART" "LAUNCH_TERMINAL Dart" "dart.png"
```

4. Ensure the following line is in your project's Gemfile:

```
# Uncomment this line to add DART dependencies to your main Gemfile
instance_eval File.read(File.join(__dir__, 'config/dart/Gemfile'))
```

5. Run the following command from your COSMOS project configuration directory (not C:/COSMOS which contains Demo & Vendor as that is the installation directory) to install all the ruby dependencies for DART.

```
bundle install
```

6. In postgres, create a 'dart' user and password (with CREATEDB permissions), and both a 'dart' and 'darttest' database. The databases should be created with template0, LC_COLLATE 'C', LC_CTYPE 'C', and ENCODING 'SQL_ASCII'

1. On windows use pgadmin

2. Otherwise use the psql command line:

```
psql postgres
CREATE ROLE dart WITH LOGIN PASSWORD 'dart';
ALTER ROLE dart CREATEDB;
CREATE DATABASE dart TEMPLATE template0 LC_COLLATE 'C' LC_CTYPE 'C' ENCODING 'SQL_ASCII';
CREATE DATABASE darttest TEMPLATE template0 LC_COLLATE 'C' LC_CTYPE 'C' ENCODING 'SQL_ASCII'
\q
```

7. Create an environment variable on your system named DART_USERNAME and set it to 'dart'. Note you can change this username if necessary as long as the postgres username created above has the same name.
8. Create an environment variable on your system named DART_PASSWORD and set it to the password created earlier.
9. Create an environment variable on your system named DART_DB and set it to 'dart'. Note: You can use whatever name you want
10. Create an environment variable on your system named DART_TEST_DB and set it to 'darttest'. Note: You can use whatever name you want
11. Open a new shell (to get the newly created environment variables) and run the following command from your COSMOS project configuration directory (not C:/COSMOS which contains Demo & Vendor as that is the installation directory)

```
bundle exec rake db:schema:load
bundle exec rake db:seed
```

Other possible dependencies

Depending on your platform other packages may be required. In particular the latest version of Ubuntu 18.04 requires libpq-dev and nodejs. Additionally you may need to edit pg_hba.conf and change from using the peer authentication method to md5 and restart postgres.

At this point the DART database is configured and ready to import COSMOS telemetry. When you start the COSMOS Demo you should see the new DART button in the Utilities section. If you're upgrading from an older version of COSMOS simply add this line to your config/tools/launcher/launcher.txt file:

```
TOOL "DART" "LAUNCH_TERMINAL Dart" "dart.png"
```

Once you click the DART button you should see a new terminal open with the following:

```
2018/04/03 16:00:30.269 INFO: Starting database cleanup...
2018/04/03 16:00:30.269 INFO: Cleaning up SystemConfig...
2018/04/03 16:00:30.373 INFO: Marshal load success: C:/git/COSMOS/demo/outputs/tmp/marshal_fee3ac209
2018/04/03 16:00:30.377 INFO: Cleaning up PacketLog...
2018/04/03 16:00:30.385 INFO: Cleaning up PacketConfig...
2018/04/03 16:00:30.392 INFO: Cleaning up PacketLogEntry...
2018/04/03 16:00:30.396 INFO: Cleaning up Decommutation tables (tX_Y)...
2018/04/03 16:00:30.396 INFO: Cleaning up Reductions...
2018/04/03 16:00:30.397 INFO: Database cleanup complete!
2018/04/03 16:00:30.397 INFO: Dart starting each process...
2018/04/03 16:00:30.397 INFO: Starting: ruby C:/git/cosmos/lib/cosmos/dart/processes/dart_ingester.r
2018/04/03 16:00:30.411 INFO: Starting: ruby C:/git/cosmos/lib/cosmos/dart/processes/dart_reducer.rb
2018/04/03 16:00:30.424 INFO: Starting: ruby C:/git/cosmos/lib/cosmos/dart/processes/dart_stream_ser
2018/04/03 16:00:30.438 INFO: Starting: ruby C:/git/cosmos/lib/cosmos/dart/processes/dart_decom_serv
2018/04/03 16:00:30.453 INFO: Starting: ruby C:/git/cosmos/lib/cosmos/dart/processes/dart_worker.rb
2018/04/03 16:00:30.478 INFO: Dart Monitoring processes...
```

This indicates that DART is now monitoring the Server waiting for packets. Note that sometimes it can take a while for output to appear in the window. If you now start the COSMOS Command and Telemetry Server, DART will start receiving and processing telemetry data.

By default, DART writes log files in the COSMOS outputs/dart/logs directory. This directory can be changed by setting DART_LOGS in the config/system/system.txt file. Log files are created for each of the DART processes and are written to continuously as DART processes data and responds to user requests. By default, DART creates binary log files in the outputs/dart/data directory. This directory can be changed by setting DART_DATA in the config/system/system.txt file.

Changing the DART Data File Directory

DART indexes the files in outputs/dart/data. If these files are moved DART can no longer access them when responding to a Stream Server request (from Replay or Data Viewer). DART will not permanently delete them unless you start DART with the '--force-cleanup' option. Using that option will permanently delete the files and their associated decommissioned data in the database.

Shutting Down DART

Always shutdown dart with Ctrl-C or a soft kill (kill -2). A hard kill will usually require database cleanup on the next startup of DART.

 BACK

NEXT 

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Custom Search



Proudly hosted by **GitHub**



Navigate the docs... ▾

Improve this page

- [DART Decommutation Server](#)
- [DART Stream Server](#)

DART API

DART (Data Archival Retrieval Trending) provides two client APIs to access the data. Decommutated data can be accessed by sending JSON requests to the DART Decommutation Server. A raw packet stream can also be accessed by sending JSON requests to the DART Stream Server.

DART Decommutation Server

The DART Decommutation API implements a relaxed version of the [JSON-RPC 2.0 Specification](#). Requests with an "id" of NULL are not supported. Numbers can contain special non-string literal's such as NaN, and +/-inf. Request params must be specified by-position, by-name is not supported. Section 6 of the spec, Batch Operations, is not supported.

DART creates a HTTP Server at port 8777 by default to respond to request for decommutated data. Note that this can be changed by overriding the DART_DECOM port in the system.txt file. The server expects to receive JSON RPC formatted request to the "query" method with a since hash parameter with the following fields.

FIELD NAME	DESCRIPTION	EXAMPLE	REQUIRED (DEFAULT)
start_time_sec	Unix start time in seconds (UTC)	1514764800 (Jan 1, 2018 00:00:00)	Yes
start_time_usec	Microseconds part of the start time	0	Yes
end_time_sec	Unix end time in seconds (UTC)	1514809815 (Jan 1, 2018 12:30:15)	Yes
end_time_usec	Microseconds part of the end time	0	Yes
item	Array specifying the target, packet, and item name	['INST','HEALTH_STATUS','TEMP1']	Yes
reduction	How to reduce the data	"NONE", "MINUTE", "HOUR", "DAY"	Yes
value_type	The type of data to return	"RAW", "RAW_MAX", "RAW_MIN", "RAW_AVG", "RAW_STDDEV", "CONVERTED", "CONVERTED_MAX", "CONVERTED_MIN", "CONVERTED_AVG", "CONVERTED_STDDEV"	Yes
cmd_tlm	Whether the request is for command or telemetry data	'CMD' or 'TLM'	No ('TLM')

limit	Maximum number of data items to return. Must be less than or equal to 10000.	100	No (10000)
offset	Offset into the data stream. Since the maximum number of values allowed is 10000, you can set the offset to 10000, then 20000, etc to get additional values.	10000	No (0)
meta_filters	Array of logical meta data filter expressions. Supports logical assertions on all the defined SYSTEM META items in your COSMOS definition. Logical operators include '==' (or '==', both mean equals), '!= (not equal)', '>', '<', '>=' and '<='.	["OPERATOR_NAME" == "Jason"]	No ([])
meta_ids	Array of the meta ID(s) to use when filtering the data. The meta IDs are an internal DART ID and thus this is only used if you have obtained the database meta IDs from a previous DART Decommunication Server request.	1	No ([])

After sending the request, the JSON RPC response will contain an Array of Arrays containing the item value, item seconds, item microseconds, samples (always 1 for NONE reduction, varies for other reduction values), and meta_id. Note this meta_id is the ID which can be used in subsequent requests in the meta_ids field.

Example Usage:

```
--> {"jsonrpc": "2.0", "method": "query", "params": [{"start_time_sec": 1514764800, "start_time_usec": 0}, {"end_time_sec": 1514809815, "end_time_usec": 0}], "id": 1}
<- {"jsonrpc": "2.0", "result": [[[{"meta_id": 10, "item": "SUSPENDED", "seconds": 1514764800, "microseconds": 0, "samples": 1}, {"meta_id": 10, "item": "SUSPENDED", "seconds": 1514764801, "microseconds": 340, "samples": 1}], {"meta_id": 11, "item": "HEALTH_STATUS", "seconds": 1514764800, "microseconds": 0, "samples": 1}, {"meta_id": 11, "item": "HEALTH_STATUS", "seconds": 1514764801, "microseconds": 340, "samples": 1}], "id": 1}
```

DART Stream Server

DART creates a TCP/IP Server at port 8779 by default to respond to requests for a stream of raw COSMOS packet data. Note that this can be changed by overriding the DART_STREAM port in the system.txt file. The server expects to receive JSON formatted requests with the following fields.

FIELD NAME	DESCRIPTION	EXAMPLE	REQUIRED (DEFAULT)
start_time_sec	Unix start time in seconds (UTC)	1514764800 (Jan 1, 2018 00:00:00)	Yes
start_time_usec	Microseconds part of the start time	0	Yes
end_time_sec	Unix end time in seconds (UTC)	1514809815 (Jan 1, 2018 12:30:15)	Yes
end_time_usec	Microseconds part of the end time	0	Yes
packets	Array of arrays specifying the target and packet name	[["INST", "HEALTH_STATUS"], ["INST", "ADCS"]]	No (All Packets)
cmd_tlm	Whether the request is for command or telemetry data	'CMD' or 'TLM'	No ('TLM')

meta_filters	Array of logical meta data filter expressions. Supports logical assertions on all the defined SYSTEM META items in your COSMOS definition. Logical operators include '=' (or '==', both mean equals), '!= (not equal)', '>', '<', '>=' and '<='.	["OPERATOR_NAME" == "Jason"]	No ([])
meta_ids	Array of the meta ID(s) to use when filtering the data. The meta IDs are an internal DART ID and thus this is only used if you have obtained the database meta IDs from a previous DART Decommunication Server request.	1	No ([])

After sending the request, the client should read from the same socket which will return COSMOS [Packets](#) using the COSMOS PREIDENTIFIED stream format until the request has streamed all the requested packets.

 **BACK** **NEXT** 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the [Jekyll](#) project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

✍ Improve this page

- [Concepts](#)
 - [Ruby Programming Language](#)
 - [Telemetry Types](#)
- [Writing Test Procedures](#)
 - [Using Subroutines](#)
- [Example Test Procedures](#)
 - [Subroutines](#)
 - [Ruby Control Structures](#)
 - [Iterating over similarly named telemetry points](#)
 - [Prompting for User Input](#)
 - [Skipping a test case in TestRunner](#)
- [Running Test Procedures](#)
- [Execution Environment](#)
 - [Using Script Runner](#)
 - [From the Command Line](#)
- [Test Procedure API](#)
- [Retrieving User Input](#)
 - [ask](#)
 - [ask_string](#)
 - [message_box](#)
 - [vertical_message_box](#)
 - [combo_box](#)
 - [save_file_dialog](#)
 - [open_file_dialog](#)
 - [open_files_dialog](#)
 - [open_directory_dialog](#)
- [Providing information to the user](#)
 - [prompt](#)
 - [status_bar](#)
 - [play_wav_file](#)
- [Commands](#)
 - [cmd](#)
 - [cmd_no_range_check](#)
 - [cmd_no_hazardous_check](#)
 - [cmd_no_checks](#)
 - [cmd_raw](#)

- [cmd_raw_no_range_check](#)
- [cmd_raw_no_hazardous_check](#)
- [cmd_raw_no_checks](#)
- [send_raw](#)
- [send_raw_file](#)
- [get_cmd_list](#)
- [get_cmd_param_list](#)
- [get_cmd_hazardous](#)
- [get_cmd_value](#)
- [get_cmd_time](#)
- [get_cmd_cnt \(since 3.9.2\)](#)
- [get_all_cmd_info \(since 4.1.0\)](#)

- **[Handling Telemetry](#)**

- [check](#)
- [check_raw](#)
- [check_formatted](#)
- [check_with_units](#)
- [check_tolerance](#)
- [check_tolerance_raw](#)
- [check_expression](#)
- [check_exception \(since 4.1.0\)](#)
- [tlm](#)
- [tlm_raw](#)
- [tlm_formatted](#)
- [tlm_with_units](#)
- [tlm_variable](#)
- [get_tlm_packet](#)
- [get_tlm_values](#)
- [get_tlm_list](#)
- [get_tlm_item_list](#)
- [get_tlm_details](#)
- [get_tlm_cnt \(since 3.9.2\)](#)
- [get_all_tlm_info \(since 4.1.0\)](#)
- [set_tlm](#)
- [set_tlm_raw](#)
- [inject_tlm](#)
- [override_tlm](#)
- [override_tlm_raw](#)
- [normalize_tlm](#)

- **[Packet Data Subscriptions](#)**

- [subscribe_packet_data](#)
- [unsubscribe_packet_data](#)
- [get_packet](#)
- [get_packet_data](#)

- **[Delays](#)**

- [wait](#)
- [wait_raw](#)

- [wait_tolerance](#)
- [wait_tolerance_raw](#)
- [wait_expression](#)
- [wait_packet](#)
- [wait_check](#)
- [wait_check_raw](#)
- [wait_check_tolerance](#)
- [wait_check_tolerance_raw](#)
- [wait_check_expression](#)
- [wait_check_packet](#)

- **Limits**

- [limits_enabled?](#)
- [enable_limits](#)
- [disable_limits](#)
- [enable_limits_group](#)
- [disable_limits_group](#)
- [get_limits_groups](#)
- [set_limits_set](#)
- [get_limits_set](#)
- [get_limits_sets](#)
- [get_limits](#)
- [set_limits](#)
- [get_out_of_limits](#)
- [get_overall_limits_state](#)
- [get_stale](#)

- **Limits Events**

- [subscribe_limits_events](#)
- [unsubscribe_limits_events](#)
- [get_limits_event](#)

- **Targets**

- [get_target_list](#)
- [get_target_info \(since 3.9.2\)](#)
- [get_all_target_info \(since 4.1.0\)](#)
- [get_target_ignored_parameters \(since 4.1.0\)](#)
- [get_target_ignored_items \(since 4.1.0\)](#)

- **Interfaces**

- [connect_interface](#)
- [disconnect_interface](#)
- [interface_state](#)
- [map_target_to_interface](#)
- [get_interface_names](#)
- [get_interface_targets \(since 3.9.2\)](#)
- [get_interface_info \(since 3.9.2\)](#)
- [get_all_interface_info \(since 4.1.0\)](#)

- **Routers**

- [connect_router](#)
- [disconnect_router](#)

- [router_state](#)
- [get_router_names](#)
- [get_router_info \(since 3.9.2\)](#)
- [get_all_router_info \(since 4.1.0\)](#)
- **Logging**
 - [get_cmd_log_filename](#)
 - [get_tlm_log_filename](#)
 - [start_logging](#)
 - [start_cmd_log](#)
 - [start_tlm_log](#)
 - [stop_logging](#)
 - [stop_cmd_log](#)
 - [stop_tlm_log](#)
 - [get_server_message_log_filename](#)
 - [start_new_server_message_log](#)
 - [start_raw_logging_interface](#)
 - [stop_raw_logging_interface](#)
 - [start_raw_logging_router](#)
 - [stop_raw_logging_router](#)
 - [get_packet_loggers \(since 4.1.0\)](#)
 - [get_packet_logger_info \(since 3.9.2\)](#)
 - [get_all_packet_logger_info \(since 4.1.0\)](#)
 - [get_output_logs_filenames \(since 4.1.0\)](#)
- **Command and Telemetry Server**
 - [get_server_status \(since 4.1.0\)](#)
 - [cmd_tlm_reload \(since 4.1.0\)](#)
 - [cmd_tlm_clear_counters \(since 4.1.0\)](#)
 - [subscribe_server_messages \(since 4.1.0\)](#)
 - [unsubscribe_server_messages \(since 4.1.0\)](#)
 - [get_server_message \(since 4.1.0\)](#)
 - [get_background_tasks \(since 4.1.0\)](#)
 - [start_background_task \(since 4.1.0\)](#)
 - [stop_background_task \(since 4.1.0\)](#)
- **Replay**
 - [set_replay_mode \(since 4.1.0\)](#)
 - [get_replay_mode \(since 4.1.0\)](#)
 - [replay_select_file \(since 4.1.0\)](#)
 - [replay_status \(since 4.1.0\)](#)
 - [replay_set_playback_delay \(since 4.1.0\)](#)
 - [replay_play \(since 4.1.0\)](#)
 - [replay_reverse_play \(since 4.1.0\)](#)
 - [replay_stop \(since 4.1.0\)](#)
 - [replay_step_forward \(since 4.1.0\)](#)
 - [replay_step_back \(since 4.1.0\)](#)
 - [replay_move_start \(since 4.1.0\)](#)
 - [replay_move_end \(since 4.1.0\)](#)
 - [replay_move_index \(since 4.1.0\)](#)

- [Executing Other Procedures](#)
 - [start](#)
 - [load_utility](#)
- [Opening, Closing & Creating Telemetry Screens](#)
 - [display](#)
 - [clear](#)
 - [clear_all \(since 3.9.2\)](#)
 - [get_screen_list \(since 4.1.0\)](#)
 - [get_screen_definition \(since 4.1.0\)](#)
 - [local_screen \(since 4.3.0\)](#)
 - [close_local_screens \(since 4.3.0\)](#)
- [Script Runner Specific Functionality](#)
 - [set_line_delay](#)
 - [get_line_delay](#)
 - [get_scriptrunner_message_log_filename](#)
 - [start_new_scriptrunner_message_log](#)
 - [disable_instrumentation](#)
 - [set_stdout_max_lines](#)
- [Debugging](#)
 - [insert_return](#)
 - [step_mode](#)
 - [run_mode](#)
 - [show_backtrace](#)
 - [shutdown_cmd_tlm](#)
 - [set_cmd_tlm_disconnect](#)
 - [get_cmd_tlm_disconnect](#)

Scripting Guide

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:

```

#####
# 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
load_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 'load_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. 'load_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 loaded.

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 telemetry 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

```
# 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

load_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

```

Skipping a test case in TestRunner

```

def test_feature_x
  continue = ask("Test feature x?")

  if continue == 'y'
    # Test goes here
  else
    raise SkipTestCase, "Skipping feature x test"
  end
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.
blank_or_default	Whether or not to allow empty responses (optional - defaults to false). If a non-boolean value is passed it is used as a default value.
password	Whether to treat the entry as a password which is displayed with dots and not logged.

Example:

```

value = ask("Enter an integer")
value = ask("Enter a value or nothing", true)
value = ask("Enter a value", 10)
password = ask("Enter your password", false, true)

```

ask_string

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

Syntax:

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

PARAMETER	DESCRIPTION
question	Question to prompt the user with.
blank_or_default	Whether or not to allow empty responses (optional - defaults to false). If a non-boolean value is passed it is used as a default value.
password	Whether to treat the entry as a password which is displayed with dots and not logged.

Example:

```

string = ask_string("Enter a String")
string = ask_string("Enter a value or nothing", true)
string = ask_string("Enter a value", "test")
password = ask_string("Enter your password", false, true)

```

message_box

vertical_message_box

combo_box

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>", ...)
message_box("<message>", "<button text 1>", ..., false) # Since COSMOS 3.8.3
vertical_message_box("<message>", "<button text 1>", ...) # Since COSMOS 3.5.0
vertical_message_box("<message>", "<button text 1>", ..., false) # Since COSMOS 3.8.3
combo_box("<message>", "<selection text 1>", ...) # Since COSMOS 3.5.0
combo_box("<message>", "<selection text 1>", ..., false) # Since COSMOS 3.8.3

```

PARAMETER	DESCRIPTION
message	Message to prompt the user with.

button/selection text	Text for a button or selection
false	Whether to display the “Cancel” button (since 3.8.3)

Example:

```

value = message_box("Select the sensor number", 'One', 'Two')
value = vertical_message_box("Select the sensor number", 'One', 'Two')
value = combo_box("Select the sensor number", 'One', 'Two')

case value
when 'One'
  puts 'Sensor One'
when 'Two'
  puts 'Sensor Two'
end

```

save_file_dialog

open_file_dialog

open_files_dialog

open_directory_dialog

The save_file_dialog, open_file_dialog, open_files_dialog, and open_directory_dialog methods create a file dialog box so the user can select a file/directory. The selected file/directory is returned.

Syntax:

```

save_file_dialog("<directory>", "<message>", "<filter>")
open_file_dialog("<directory>", "<message>", "<filter>")
open_files_dialog("<directory>", "<message>", "<filter>")
open_directory_dialog("<directory>", "<message>")

```

PARAMETER	DESCRIPTION
Directory	The directory to start browsing in. Optional parameter, defaults to Cosmos::USERPATH.
Message	The message to display in the dialog box. Optional parameter, defaults to “Save File”, “Open File”, “Open File(s)”, or “Open Directory”.
Filter	Filter to select allowed file type. Optional parameter, defaults to “*”

Example:

```

selected_file = open_file_dialog()
file_data = ""
File.open(selected_file, 'rb') {|file| file_data = file.read()}

# Filter will initially show only .txt files, but can be changed to show all files...
selected_file = open_file_dialog(Cosmos::USERPATH, "Open File", "Text (*.txt);;All (*.*)")

```

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

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  
cmd_no_range_check("<Target Name>", "<Command Name>", "Param #1 Name" => <Param #1 Value>, "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_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 Value>")
cmd_no_hazardous_check("<Target Name>", "<Command Name>", "Param #1 Name" => <Param #1 Value>, "Param #2 Value" => <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_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 intentionally have invalid parameters.

Syntax:

```
cmd_no_checks("<Target Name> <Command Name> with <Param #1 Name> <Param #1 Value>, <Param #2 Name> <Param #2 Value>")
cmd_no_checks("<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_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 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_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> <Param #2 Value>")
cmd_raw_no_range_check("<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_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> <Param #2 Value>,...")
cmd_raw_no_hazardous_check("<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_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 intentionally have invalid parameters.

Syntax:

```
cmd_raw_no_checks("<Target Name> <Command Name> with <Param #1 Name> <Param #1 Value>, <Param #2 Name> <Param #2 Value>")  
cmd_raw_no_checks("<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_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 arrays where 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, ... ]]
```

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

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

Syntax:

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

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

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

Syntax:

```
get_cmd_time("<Target Name - optional>", "<Command Name - optional>") # Since COSMOS 3.5.0
```

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

get_cmd_cnt (since 3.9.2)

The get_cmd_cnt method returns the number of times a specified command has been sent.

Syntax:

```
get_cmd_cnt("<Target Name>", "<Command Name>")
```

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

Example:

```
cmd_cnt = get_cmd_cnt("INST", "COLLECT") # Number of times the INST COLLECT command has been sent
```

get_all_cmd_info (since 4.1.0)

The get_all_cmd_info method returns the number of times each command has been sent. The return value is an array of arrays where each subarray contains the target name, command name, and packet count for a command.

Syntax/ Example:

```
cmd_info = get_all_cmd_info()
cmd_info.each do |target_name, cmd_name, pkt_count|
  puts "Target: #{target_name}, Command: #{cmd_name}, Packet count: #{pkt_count}"
end
```

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 comparison 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 comparison 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	\pm 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	\pm 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")
```

check_exception (since 4.1.0)

The check_exception method executes a method and expects an exception to be raised. If the method does not raise an exception, a CheckError is raised.

Syntax:

```
check_exception("<Method Name>, "<Method Params - optional>")
```

PARAMETER	DESCRIPTION
Method Name	The COSMOS scripting method to execute, e.g. 'cmd', 'cmd_raw', etc.
Method Params	Parameters for the method

Example:

```
check_exception("cmd", "INST", "COLLECT", "TYPE=>"NORMAL")
```

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",
```

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_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_details("INST", "HEALTH_STATUS", "COLLECTS")
```

get_tlm_cnt (since 3.9.2)

The `get_tlm_cnt` method returns the number of times a specified telemetry packet has been received.

Syntax:

```
get_tlm_cnt("<Target Name>", "<Packet Name>")
```

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

Example:

```
tlm_cnt = get_tlm_cnt("INST", "HEALTH_STATUS") # Number of times the INST HEALTH_STATUS telemetry pa
```

get_all_tlm_info (since 4.1.0)

The `get_all_tlm_info` method returns the number of times each telemetry packet has been received. The return value is an array of arrays where each subarray contains the target name, telemetry packet name, and packet count for a telemetry packet.

Syntax / Example:

```

tlm_info = get_all_tlm_info()
tlm_info.each do |target_name, pkt_name, pkt_count|
  puts "Target: #{target_name}, Packet: #{pkt_name}, Packet count: #{pkt_count}"
end

```

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
```

inject_tlm

The inject_tlm method injects a packet into the system as if it was received from an interface.

Syntax:

```
inject_tlm("<target_name>", "<packet_name>", <item_hash>, <value_type>, <send_routers>, <send_packet_log_writers>)
```

PARAMETER	DESCRIPTION
Target	Target name
Packet	Packet name
Item Hash	Hash of item name/value for each item. If an item is not specified in the hash, the current value table value will be used. Optional parameter, defaults to nil.
Value Type	Type of values in the item hash (:RAW or :CONVERTED). Optional parameter, defaults to :CONVERTED.
Send Routers	Whether or not to send to routers for the target's interface. Optional parameter, defaults to true.
Send Packet Log Writers	Whether or not to send to the packet log writers for the target's interface. Optional parameter, defaults to true.
Create New Logs	Whether or not to create new log files before writing this packet to logs. Optional parameter, defaults to false.

Example:

```
inject_tlm("INST", "PARAMS", { 'VALUE1'=>5.0, 'VALUE2'=>7.0})
```

override_tlm

The override_tlm method sets the converted value for a telemetry point in the Command and Telemetry Server. This value will be maintained even if a new packet is received on the interface unless the override is canceled with the normalize_tlm method. Note that the interface definition must explicitly add this capability by declaring PROTOCOL READ OverrideProtocol (refer to the documentation for the [override protocol](#)).

Syntax:

```
override_tlm("<Target> <Packet> <Item> = <Value>")
```

PARAMETER	DESCRIPTION
-----------	-------------

Target	Target name
Packet	Packet name
Item	Item name
Value	Value to set

Example:

```
override_tlm("INST HEALTH_STATUS TEMP1 = 5")
```

override_tlm_raw

The override_tlm_raw method sets the raw value for a telemetry point in the Command and Telemetry Server. This value will be maintained even if a new packet is received on the interface unless the override is canceled with the normalize_tlm method. Note that the interface definition must explicitly add this capability by declaring PROTOCOL READ OverrideProtocol (refer to the documentation for the [override protocol](#)).

Syntax:

```
override_tlm_raw("<Target> <Packet> <Item> = <Value>")
```

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

Example:

```
override_tlm_raw("INST HEALTH_STATUS TEMP1 = 5")
```

normalize_tlm

The normalize_tlm method clears the override of a telemetry point in the Command and Telemetry Server. Note that the interface definition must explicitly add this capability by declaring PROTOCOL READ OverrideProtocol (refer to the documentation for the [override protocol](#)).

Syntax:

```
normalize_tlm("<Target> <Packet> <Item>")
```

PARAMETER	DESCRIPTION
Target	Target name
Packet	Packet name

Item

Item name

Example:

```
normalize_tlm("INST HEALTH_STATUS TEMP1")
```

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.

!! Overflow Queues Are Deleted

By default the packet queue is 1000 packets deep. If you don't call get_packet fast enough to keep up with the population of this queue and it overflows, COSMOS will clean up the resources and delete the queue. At this point when you call get_packet you will get a "RuntimeError : Packet data queue with id X not found." Note you can pass a larger queue size to the subscribe_packet_data method.

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 a 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.

!! Overflow Queues Are Deleted

By default the packet queue is 1000 packets deep. If you don't call get_packet_data fast enough to keep up with the population of this queue and it overflows, COSMOS will clean up the resources and delete the queue. At this point when you call get_packet_data you will get a "RuntimeError : Packet data queue with id X not found." Note you can pass a larger queue size to the subscribe_packet_data method.

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 (minimum 10ms) 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>, <
```

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	\pm 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>)
```

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	\pm 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",
```

```
< >
```

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>)
```

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	\pm Tolerance on the expected value.
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.

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>)
```

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	\pm Tolerance on the expected value.
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.

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') > 2")
```

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 / 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/ Example:

```
limits_set = get_limits_set()
```

get_limits_sets

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

Syntax/ 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)>)
```
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 t

```

Example:

```

```ruby
limits_set, persistence_setting, enabled, red_low, yellow_low, yellow_high, red_high, green_low, green_high = get_limits("satellite", "imu", "roll")
```

```

## 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 Hi

```

| 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 limits 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 / 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', 'PACKET_NAME', 'ITEM_NAME'], ...] |

Example:

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

## get\_stale

The get\_stale method returns a list of stale packets. The return value is an array of arrays where each subarray contains the target name and packet name for a stale packet.

Syntax:

```
get_stale(<with_limits_only> (optional), <target> (optional))
```

| PARAMETER        | DESCRIPTION                                                                                                                                            |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| With Limits Only | If true, return only the packets that have limits items and thus affect the overall limits state of the system. Optional parameter, defaults to false. |
| Target           | If specified, return only the packets associated with the given target. Optional parameter, defaults to nil.                                           |

Example:

```
stale_packets = get_stale()
stale_packets.each do |target, packet|
 puts "Stale packet: #{target} #{packet}"
end
```

## 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.



### Overflow Queues Are Deleted

By default the limits queue is 1000 events deep. If you don't call `get_limits_event` fast enough to keep up with the population of this queue and it overflows, COSMOS will clean up the resources and delete the queue. At this point when you call `get_limits_event` you will get a "RuntimeError : Packet data queue with id X not found." Note you can pass a larger queue size to the `subscribe_limits_events` method.

Syntax:

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

| PARAMETER | DESCRIPTION                                                                                          |
|-----------|------------------------------------------------------------------------------------------------------|
| id        | Unique id given to the tool by <code>subscribe_limits_events</code> .                                |
| 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_STA
puts event.inspect # [:LIMITS_SET, :NEW_LIMITS_SET)
puts event.inspect # [:LIMITS_SETTINGS, "TARGET_NAME", "PACKET_NAME", "ITEM_NAME", :LIMITS_SET, persi
```

## 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 / Example:

```
targets = get_target_list()
```

### get\_target\_info (since 3.9.2)

The `get_target_info` method returns information about a target. The information includes the number of commands sent to the target and the number of telemetry packets received from the target.

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

| PARAMETER   | DESCRIPTION         |
|-------------|---------------------|
| Target Name | Name of the target. |

Example:

```
cmd_cnt, tlm_cnt = get_target_info("INST")
```

### get\_all\_target\_info (since 4.1.0)

The get\_all\_target\_info method returns information about all targets. The return value is an array of arrays where each subarray contains the target name, interface name, command count, and telemetry count for a target.

Syntax/ Example:

```
target_info = get_all_target_info()
target_info.each do |target_name, interface_name, cmd_count, tlm_count|
 puts "Target: #{target_name}, Interface: #{interface_name}, Cmd count: #{cmd_count}, Tlm count: #{tlm_count}"
end
```

## get\_target\_ignored\_parameters (since 4.1.0)

The get\_target\_ignored\_parameters method returns a list of ignored command parameters for the specified target. Ignored command parameters are those specified by the IGNORE\_PARAMETER keyword in the target configuration.

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

| PARAMETER   | DESCRIPTION         |
|-------------|---------------------|
| Target Name | Name of the target. |

Example:

```
ignored_params = get_target_ignored_parameters("INST")
```

## get\_target\_ignored\_items (since 4.1.0)

The get\_target\_ignored\_items method returns a list of ignored telemetry items for the specified target. Ignored telemetry items are those specified by the IGNORE\_ITEM keyword in the target configuration.

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

| PARAMETER   | DESCRIPTION         |
|-------------|---------------------|
| Target Name | Name of the target. |

Example:

```
ignored_items = get_target_ignored_items("INST")
```

# 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/ Example:

```
interface_names = get_interface_names()
```

## get\_interface\_targets (since 3.9.2)

The get\_interface\_targets method returns the list of targets which are mapped to the given interface.

Syntax:

```
get_interface_targets("<Interface Name>")
```

| PARAMETER      | DESCRIPTION            |
|----------------|------------------------|
| Interface Name | Name of the interface. |

Example:

```
targets = get_interface_targets("INST_INT")
```

## get\_interface\_info (since 3.9.2)

The get\_interface\_info method returns information about an interface. The information includes the connection state, number of connected clients, transmit queue size, receive queue size, bytes transmitted, bytes received, command count, and telemetry count.

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

| PARAMETER      | DESCRIPTION            |
|----------------|------------------------|
| Interface Name | Name of the interface. |

Example:

```
state, clients, tx_q_size, rx_q_size, bytes_tx, bytes_rx, cmd_cnt, tlm_cnt = get_interface_info("INS")
```

## get\_all\_interface\_info (since 4.1.0)

The get\_all\_interface\_info method returns information about all interfaces. The return value is an array of arrays where each subarray contains the interface name, connection state, number of connected clients, transmit queue size, receive queue size, bytes transmitted, bytes received, command count, and telemetry count.

Syntax/ Example:

```
interface_info = get_all_interface_info()
interface_info.each do |interface_name, connection_state, num_clients, tx_q_size, rx_q_size, tx_bytes, rx_bytes, cmd_count, tlm_count|
 puts "Interface: #{interface_name}, Connection state: #{connection_state}, Num connected clients: #{num_clients}"
 puts "Transmit queue size: #{tx_q_size}, Receive queue size: #{rx_q_size}, Bytes transmitted: #{tx_bytes}"
 puts "Cmd count: #{cmd_count}, Tlm count: #{tlm_count}"
end
```

# 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 router will 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 / Example:

```
router_names = get_router_names()
```

### **get\_router\_info (since 3.9.2)**

The get\_router\_info method returns information about a router. The information includes the connection state, number of connected clients, transmit queue size, receive queue size, bytes transmitted, bytes received, packets received, and packets sent.

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

| PARAMETER   | DESCRIPTION         |
|-------------|---------------------|
| Router Name | Name of the router. |

Example:

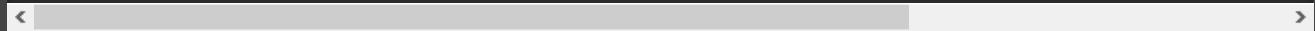
```
state, clients, tx_q_size, rx_q_size, bytes_tx, bytes_rx, pkts_rcvd, pkts_sent = get_router_info("IN:
```

### **get\_all\_router\_info (since 4.1.0)**

The get\_all\_router\_info method returns information about all routers. The return value is an array of arrays where each subarray contains the router name, connection state, number of connected clients, transmit queue size, receive queue size, bytes transmitted, bytes received, packets received, and packets sent.

Syntax/ Example:

```
router_info = get_all_router_info()
router_info.each do |router_name, connection_state, num_clients, tx_q_size, rx_q_size, tx_bytes, rx_bytes, pkts_rcvd, pkts_sent|
 puts "Router: #{router_name}, Connection state: #{connection_state}, Num connected clients: #{num_clients}"
 puts "Transmit queue size: #{tx_q_size}, Receive queue size: #{rx_q_size}, Bytes transmitted: #{tx_bytes}"
 puts "Packets received: #{pkts_rcvd}, Packets sent: #{pkts_sent}"
end
```



## 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. Labels must consist of only letters and numbers (no underscores, hyphens, etc).                                                                     |

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 / Example:

```
filename = get_server_message_log_filename()
```

### **start\_new\_server\_message\_log**

Starts a new COSMOS Command and Telemetry Server message log.

Syntax / 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")
```

## **get\_packet\_loggers (since 4.1.0)**

The get\_packet\_loggers method returns a list of the packet loggers in the system.

Syntax/ Example:

```
packet_loggers = get_packet_loggers()
```

## **get\_packet\_logger\_info (since 3.9.2)**

The get\_packet\_logger\_info method returns information about a packet logger. The information includes the interfaces associated with the logger, command log enable flag, command queue size, command filename, command file size, telemetry log enable flag, telemetry queue size, telemetry filename, and telemetry file size.

Syntax: `get_packet_logger_info("<Packet logger name>")`

| PARAMETER          | DESCRIPTION                                       |
|--------------------|---------------------------------------------------|
| Packet Logger name | Name of the packet logger to get information for. |

Example:

```
interfaces, cmd_logging, cmd_q_size, cmd_filename, cmd_file_size, tlm_logging, tlm_q_size, tlm_filename
```

## **get\_all\_packet\_logger\_info (since 4.1.0)**

The get\_all\_packet\_logger\_info method returns information about all packet loggers. The return value is an array of arrays where each subarray contains the name, associated interfaces, command log enable flag, command queue size, command filename, command file size, telemetry log enable flag, telemetry queue size, telemetry filename, and telemetry file size for a packet logger.

Syntax/ Example:

```
packet_logger_info = get_all_packet_logger_info()
packet_logger_info.each do |packet_logger_name, interfaces, cmd_logging, cmd_q_size, cmd_filename, cmd_file_size, tlm_logging, tlm_q_size, tlm_filename|
 puts "Packet logger: #{packet_logger_name}"
 puts "Associated interfaces: #{interfaces}"
 puts "Cmd logging - enable: #{cmd_logging}, queue size: #{cmd_q_size}, filename: #{cmd_filename}, file size: #{cmd_file_size}"
 puts "Tlm logging - enable: #{tlm_logging}, queue size: #{tlm_q_size}, filename: #{tlm_filename}, file size: #{tlm_file_size}"
end
```

## **get\_output\_logs\_filenames (since 4.1.0)**

The get\_output\_logs\_filenames method returns a list of files in the output logs directory.

Syntax:

```
get_output_logs_filenames("<filter>")
```

| PARAMETER | DESCRIPTION                                                                                                                      |
|-----------|----------------------------------------------------------------------------------------------------------------------------------|
| Filter    | String that can be used to filter the files returned. Defaults to '*tlm.bin', which will return only binary telemetry log files. |

Example:

```
tlm_logs = get_output_logs_filenames()
cmd_logs = get_output_logs_filenames('*_cmd.bin')
server_msg_logs = get_output_logs_filenames('*_server_messages.txt')
```

## Command and Telemetry Server

These methods allow the user to interact with Command and Telemetry Server.

### **get\_server\_status (since 4.1.0)**

The get\_server\_status method returns status information for the Command and Telemetry Server. The information includes the active limits set, API port number, JSON DRB number of clients, JSON DRB average request count, JSON DRB average request time, and number of server threads.

Syntax/ Example:

```
limits_set, api_port, json_drb_num_clients, json_drb_req_count, json_drb_avg_req_time, num_threads =
```

### **cmd\_tlm\_reload (since 4.1.0)**

The cmd\_tlm\_reload method reloads the default configuration in the Command and Telemetry Server.

Syntax/ Example:

```
cmd_tlm_reload()
```

### **cmd\_tlm\_clear\_counters (since 4.1.0)**

The cmd\_tlm\_clear\_counters method resets the counters in the Command and Telemetry Server back to zero.

Syntax/ Example:

```
cmd_tlm_clear_counters()
```

### **subscribe\_server\_messages (since 4.1.0)**

The subscribe\_server\_messages method allows the user to listen for server messages. A unique id is returned to the tool which is used to retrieve the messages. The messages are placed into a queue where they can then be processed one at a time.

Syntax:

```
subscribe_server_messages(queue_size)
```

| PARAMETER  | DESCRIPTION                                                                          |
|------------|--------------------------------------------------------------------------------------|
| queue_size | Number of messages to let queue up before dropping the connection. Defaults to 1000. |

Example:

```
id = subscribe_server_messages(2000)
```

### **unsubscribe\_server\_messages (since 4.1.0)**

The unsubscribe\_server\_messages method allows the user to stop listening for server messages. This should be called to reduce the server's load if the subscription is no longer needed.

Syntax:

```
unsubscribe_server_messages(id)
```

| PARAMETER | DESCRIPTION                                               |
|-----------|-----------------------------------------------------------|
| id        | Unique id given to the tool by subscribe_server_messages. |

Example:

```
unsubscribe_server_messages(id)
```

### get\_server\_message (since 4.1.0)

Receives a subscribed server message. If this method is called non-blocking = true, this method will raise an error if the queue is empty. The return value is an array where the first element is the message and the second element is the color associated with the message (BLACK, RED, YELLOW, GREEN).



#### Overflow Queues Are Deleted

By default the message queue is 1000 messages deep. If you don't call get\_server\_message fast enough to keep up with the population of this queue and it overflows, COSMOS will clean up the resources and delete the queue. At this point when you call get\_server\_message you will get a "RuntimeError : Packet data queue with id X not found." Note you can pass a larger queue size to the subscribe\_server\_messages method.

Syntax:

```
get_server_message(id, non_block (optional))
```

| PARAMETER | DESCRIPTION                                                                                                                           |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------|
| id        | Unique id given to the tool by subscribe_server_messages.                                                                             |
| non_block | Boolean to indicate if the method should block until a message is received or not. Defaults to false, blocks reading data from queue. |

Example:

```
msg, color = get_server_message(id)
```

### get\_background\_tasks (since 4.1.0)

The get\_background\_tasks method returns information about all background tasks. The return value is an array of arrays where each subarray contains the name, state, and status string for a background task.

Syntax / Example:

```
background_tasks = get_background_tasks()
background_tasks.each do |background_task_name, state, status_string|
 puts "Background task: #{background_task_name}, state: #{state}, status: #{status_string}"
end
```

### start\_background\_task (since 4.1.0)

The start\_background\_task method starts a background task.

Syntax:

```
start_background_task("<Background Task Name>")
```

| PARAMETER            | DESCRIPTION                  |
|----------------------|------------------------------|
| Background Task Name | Name of the background task. |

Example:

```
start_background_task("Example Background Task")
```

### **stop\_background\_task (since 4.1.0)**

The stop\_background\_task method stops a background task.

Syntax:

```
stop_background_task("<Background Task Name>")
```

| PARAMETER            | DESCRIPTION                  |
|----------------------|------------------------------|
| Background Task Name | Name of the background task. |

Example:

```
stop_background_task("Example Background Task")
```

## **Replay**

These methods allow the user to control the COSMOS Replay tool.

### **set\_replay\_mode (since 4.1.0)**

The set\_replay\_mode method configures the JSON DRB connection to connect to either the CTS or the Replay tool. The JSON DRB connection must be configured to connect to the Replay tool with this method before any of the other methods in the Replay API can be used.

Syntax:

```
set_replay_mode(<replay_mode>)
```

| PARAMETER   | DESCRIPTION                                                                    |
|-------------|--------------------------------------------------------------------------------|
| Replay Mode | Set to true to connect to the Replay tool; set to false to connect to the CTS. |

Example:

```
set_replay_mode(true)
```

### **get\_replay\_mode (since 4.1.0)**

The get\_replay\_mode method returns true if the JSON DRB connection is configured to connect to the Replay tool or

false if the JSON DRB connection is configured to connect to the CTS.

Syntax/ Example:

```
replay_mode = get_replay_mode()
```

### **replay\_select\_file (since 4.1.0)**

The replay\_select\_file method selects a file to play back in the COSMOS Replay tool.

Syntax:

```
replay_select_file("<filename>", <packet_log_reader> (optional))
```

| PARAMETER         | DESCRIPTION                                                                             |
|-------------------|-----------------------------------------------------------------------------------------|
| Filename          | A log file to load into the Replay tool.                                                |
| Packet Log Reader | The log reader to use to parse the log file. Optional parameter, defaults to "DEFAULT". |

Example:

```
replay_select_file("2018_01_04_13_19_49_tlm.bin")
```

### **replay\_status (since 4.1.0)**

The replay\_status method returns status for the Replay tool. The returned status includes the PLAYING/STOPPED status, playback delay, playback filename, file start time, file current (playback) time, file end time, file index, and file max index.

Syntax/ Example:

```
status, delay, filename, file_start, file_current, file_end, file_index, file_max_index = replay_st
```

### **replay\_set\_playback\_delay (since 4.1.0)**

The replay\_set\_playback\_delay method sets the playback delay for the Replay tool.

Syntax:

```
replay_set_playback_delay(<delay>)
```

| PARAMETER | DESCRIPTION                                                                                                                                                                                  |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Delay     | The delay between packets when the Replay tool is playing back. Set to nil for REALTIME, or specify a floating point value in seconds from 0.0 (no delay) to 1.0 (1 second between packets). |

Example:

```
replay_set_playback_delay(nil)
```

### **replay\_play (since 4.1.0)**

The replay\_play method starts playback in the Replay tool.

Syntax/ Example:

```
replay_play()
```

## replay\_reverse\_play (since 4.1.0)

The replay\_reverse\_play method starts reverse playback in the Replay tool.

Syntax/ Example:

```
replay_move_end()
replay_reverse_play()
```

## replay\_stop (since 4.1.0)

The replay\_stop method stops playback in the Replay tool.

Syntax/ Example:

```
replay_stop()
```

## replay\_step\_forward (since 4.1.0)

The replay\_step\_forward method steps the Replay tool forward by one packet.

Syntax/ Example:

```
replay_step_forward()
```

## replay\_step\_back (since 4.1.0)

The replay\_step\_back method steps the Replay tool backwards by one packet.

Syntax/ Example:

```
replay_step_back()
```

## replay\_move\_start (since 4.1.0)

The replay\_move\_start method sets the Replay tool playback pointer to the start of the file.

Syntax/ Example:

```
replay_move_start()
```

## replay\_move\_end (since 4.1.0)

The replay\_move\_end method sets the Replay tool playback pointer to the end of the file.

Syntax/ Example:

```
replay_move_end()
```

## replay\_move\_index (since 4.1.0)

The replay\_move\_index method sets the Replay tool playback pointer to a specified index. The maximum index can be found with the replay\_status method.

Syntax:

```
replay_move_index(<index>)
```

| PARAMETER | DESCRIPTION                                                    |
|-----------|----------------------------------------------------------------|
| Index     | The packet index within the file to move the playback pointer. |

Example:

```
replay_move_index(10)
```

## 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<br>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 be highlighted. If you want to import subroutines but do not want their lines to be highlighted in ScriptRunner or TestRunner, use the standard Ruby 'load' or 'require' statement.

Syntax:

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

| PARAMETER           | DESCRIPTION                                                                                                                                                                                |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Utility<br>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, Closing & Creating Telemetry Screens

These methods allow the user to open, close or create unique 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")
```

### clear\_all (since 3.9.2)

The clear\_all method closes all open screens or all screens of a particular target.

Syntax:

```
clear_all("<Target Name>")
```

| PARAMETER   | DESCRIPTION                                                                                   |
|-------------|-----------------------------------------------------------------------------------------------|
| Target Name | Close all screens associated with the target. If no target is passed, all screens are closed. |

Example:

```
clear_all("INST") # Clear all INST screens
clear_all() # Clear all screens
```

## get\_screen\_list (since 4.1.0)

The get\_screen\_list returns a list of available telemetry screens.

Syntax:

```
get_screen_list("<config_filename>", <force_refresh>)
```

| PARAMETER       | DESCRIPTION                                                                                                                 |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------|
| Config filename | A telemetry viewer config file to parse. If nil, the default config file will be used. Optional parameter, defaults to nil. |
| Force refresh   | If true the config file will be re-parsed. Optional parameter, defaults to false.                                           |

Example:

```
screen_list = get_screen_list()
```

## get\_screen\_definition (since 4.1.0)

The get\_screen\_definition returns the text file contents of a telemetry screen definition.

Syntax:

```
get_screen_definition("<screen_full_name>", "<config_filename>", <force_refresh>)
```

| PARAMETER        | DESCRIPTION                                                                                                                 |
|------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Screen full name | Telemetry screen name.                                                                                                      |
| Config filename  | A telemetry viewer config file to parse. If nil, the default config file will be used. Optional parameter, defaults to nil. |
| Force refresh    | If true the config file will be re-parsed. Optional parameter, defaults to false.                                           |

Example:

```
screen_definition = get_screen_definition("INST HS")
```

## local\_screen (since 4.3.0)

The local\_screen allows you to create a temporary screen directly from a script. This also has the ability to use local variables from within your script in your screen.

Syntax:

```
local_screen("<title>", "<screen definition>", <x position>, <y position>)
```

| PARAMETER         | DESCRIPTION                                                                                                                      |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------|
| Title             | Screen title                                                                                                                     |
| Screen Definition | You can pass the entire screen definition as a Ruby String or define it inline in a block. Optional parameter, defaults to nil.  |
| X Position        | X Position in pixels to display the screen. Note the top left corner of the display is 0,0. Optional parameter, defaults to nil. |
| Y Position        | Y Position in pixels to display the screen. Note the top left corner of the display is 0,0. Optional parameter, defaults to nil. |

Example:

```

temp = 0 # This variable is accessed in the screen
screen_def = '
SCREEN AUTO AUTO 0.1 FIXED
VERTICAL
TITLE "Local Variable"
VERTICALBOX
LABELVALUE LOCAL LOCAL temp # Note LOCAL LOCAL
END
END

'

Here we pass in the screen definition as a string
screen = local_screen("My Screen", screen_def, 100, 100)
disable_instrumentation do
 5000000.times do
 temp += 1 # Increment temp to update the screen
 end
end
screen.close # Close this local screen

temp = 0
The screen definition is nil so we define the screen in the block
local_screen("My Screen", nil, 500, 500) do
 ' # Note the quote
 SCREEN AUTO AUTO 0.1 FIXED
 VERTICAL
 TITLE "Local Variable"
 VERTICALBOX
 LABELVALUE LOCAL LOCAL temp # LOCAL LOCAL
 END
 END
 ' # Close quote
end
disable_instrumentation do
 5000000.times do
 temp += 1 # Increment temp to update the screen
 end
end
close_local_screens() # Close all open local screens

```

## close\_local\_screens (since 4.3.0)

The close\_local\_screens closes all temporary screens which were opened using local\_screen.

Syntax/ Example:

```
close_local_screens()
```

## 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.<br>Should be $\geq 0.0$ . |

Example:

```
set_line_delay(0.0)
```

## **get\_line\_delay**

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

Syntax / Example:

```
curr_line_delay = get_line_delay()
```

## **get\_scriptrunner\_message\_log\_filename**

Returns the filename of the ScriptRunner message log.

Syntax / 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 / Example:

```
filename = start_new_scriptrunner_message_log()
```

## **disable\_instrumentation**

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 separate file and using either require/load to read the file for the same effect while still allowing errors to be caught by your script.

Syntax / Example:

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

## set\_stdout\_max\_lines

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:

```
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/ Example:

```
step_mode()
```

## run\_mode

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

Syntax/ 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/ Example:

```
show_backtrace # Shows the backtrace for the latest error
show_backtrace(true) # Enables showing backtrace for every error
show_backtrace(false) # Disables showing backtrace for every error
```

## 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/ 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 False. 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 / Example:

```
mode = get_cmd_tlm_disconnect()
```

 BACK    NEXT

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

✍ Improve this page

- [Introduction](#)
- [Scripting Philosophy](#)
  - [A Super Basic Script Example](#)
  - [KISS \(Keep It Simple Stupid\)](#)
  - [Keep things DRY \(Don't Repeat Yourself\)](#)
  - [Use Comments Appropriately](#)
  - [ScriptRunner vs TestRunner](#)
  - [Looping vs Unrolled Loops](#)
- [Script Organization](#)
  - [Organize Scripts into Test Case Functions](#)
  - [Using Classes vs Unscoped Functions](#)
  - [Organizing Your Scripts into Separate Files](#)
  - [Instrumented vs Uninstrumented Lines \(require vs load\)](#)
    - [When Running Uninstrumented Code](#)
- [Debugging and Auditing](#)
  - [Built-In Debugging Capabilities](#)
  - [Breakpoints](#)
  - [Using Disconnect Mode](#)
  - [Auditing your Scripts](#)
    - [Ruby Syntax Check](#)
    - [Mnemonic Check](#)
    - [Generate Cmd/Tlm Audit](#)
- [Common Scenarios](#)
  - [User Input Best Practices](#)
  - [Conditionally Require Manual User Input Steps](#)
  - [Outputting Extra Information to a TestRunner Test Report](#)
  - [Getting the Most Recent Value of a Telemetry Point from Multiple Packets](#)
  - [Checking Every Single Sample of a Telemetry Point](#)
  - [Using Variables in Mnemonics](#)
  - [Using Custom wait\\_check\\_expression](#)
  - [COSMOS Scripting Differences from Regular Ruby Scripting](#)
    - [Do not use single line if statements](#)
    - [Methods should use explicit return statements](#)
- [When Things Go Wrong](#)
  - [Common Reasons Checks Fail](#)
  - [How to Recover from Anomalies](#)
- [Advanced Topics](#)

- [Advanced Script Configuration with CSV or Excel](#)
- [Script specific screens](#)
- [When to use Modules](#)
- [Further Reading](#)

# Scripting Best Practices

## Introduction

This guide aims to provide the best practices for using the scripting capabilities provided by COSMOS. Scripts are used to automate a series of activities for operations or testing. The goal of this document is to ensure scripts are written that are simple, easy to understand, maintainable, and correct. Guidance on some of the key details of using COSMOS's ScriptRunner and TestRunner tools is also provided.

## Scripting Philosophy

### A Super Basic Script Example

Most COSMOS scripts can be broken down into the simple pattern of sending a command to a system/subsystem and then verifying that the command worked as expected. This pattern is most commonly implemented with cmd() followed by wait\_check(), like the following:

```
cmd("INST COLLECT with TYPE NORMAL, TEMP 10.0")
wait_check("INST HEALTH_STATUS TYPE == 'NORMAL'", 5)
```

or similarly with a counter that is sampled before the command:

```
count = tlm("INST HEALTH_STATUS COLLECTS")
cmd("INST COLLECT with TYPE NORMAL, TEMP 10.0")
wait_check("INST HEALTH_STATUS COLLECTS >= #{count + 1}", 5)
```

90% of the COSMOS scripts you write should be the simple patterns shown above except that you may need to check more than one item after each command to make sure the command worked as expected.

### KISS (Keep It Simple Stupid)

Ruby is a very powerful language with many ways to accomplish the same thing. Given that, always choose the method that is easiest to understand for yourself and others. While it is possible to create complex one liners or obtuse regular expressions, you'll thank yourself later by expanding complex one liners and breaking up and documenting regular expressions.

### Keep things DRY (Don't Repeat Yourself)

A widespread problem in scripts written for any command and control system is large blocks of code that are repeated multiple times. In extreme cases, this has led to 100,000+ line scripts that are impossible to maintain and review.

There are two common ways repetition presents itself: exact blocks of code to perform a common action such as powering on a subsystem, and blocks of code that only differ in the name of the mnemonic being checked or the values checked against. Both are solved by removing the repetition using functions (also referred to as 'methods' in Ruby).

For example, a script that powers on a subsystem and ensures correct telemetry would become:

```
def power_on_subsystem
 # 100 lines of cmd(), wait_check(), etc
End
```

Ideally, the above function would be stored in another file where it could be used by other scripts. If it is truly only useful in the one script, then it could be at the top of the file. The updated script would then look like:

```
power_on_subsystem()
150 lines operating the subsystem (e.g.)
cmd(...)
wait_check(...)
#...
power_off_subsystem()
Unrelated activities
power_on_subsystem()
etc.
```

Blocks of code where only the only variation is the mnemonics or values checked can be replaced by functions with arguments like this:

```
def test_minimum_temp(enable_cmd_name, enable_tlm, temp_tlm, expected_temp)
 cmd("TARGET #{enable_cmd_name} with ENABLE TRUE")
 wait_check("TARGET #{enable_tlm} == 'TRUE;", 5)
 wait_check("TARGET #{temp_tlm} >= #{expected_temp}", 50)
end
```

## Use Comments Appropriately

Use comments when what you are doing is unclear or there is a higher-level purpose to a set of lines. Try to avoid putting numbers or other details in a comment as they can become out of sync with the underlying code. Ruby comments start with a # pound symbol and can be anywhere on a line.

```
This line sends an abort command - BAD COMMENT, UNNECESSARY
cmd("INST ABORT")
Rotate the gimbal to look at the calibration target - GOOD COMMENT
cmd("INST ROTATE with ANGLE 180.0") # Rotate 180 degrees - BAD COMMENT
```

## ScriptRunner vs TestRunner

COSMOS provides two unique ways to run scripts (also known as procedures). ScriptRunner provides both a script execution environment and a script editor. The script editor includes code completion for both COSMOS methods and command/telemetry item names. It is also a great environment to develop and test scripts. ScriptRunner provides a framework for users that are familiar with a traditional scripting model with longer style procedures, and for users that want to be able to edit their scripts in place.

TestRunner provides a more formal, but also more powerful, environment for running scripts. The name TestRunner comes from incorporating several of the concepts from software unit testing frameworks and applying them to system level test and operations. TestRunner, by design, has users break their scripts down into test suites, test groups, and

test cases. Test Suites are the highest-level concept and would typically cover a large test procedure such as a thermal vacuum test, or a large operations scenario such as performing on orbit checkout. Test Groups capture a related set of test cases such as all the test cases regarding a specific mechanism. A Test Group might be a collection of test cases all related to a subsystem, or a specific series of tests such as an RF checkout. Test Cases capture individual activities that can either pass or fail. TestRunner allows for running an entire test suite, one or more test groups, or one or more test cases easily. It also automatically produces test reports, can easily gather meta data such as operator name, and produce associated data packages.

The correct tool for the job is up to individual users, and many programs will use both tools to complete their goals. For example, while formal tests are typically organized and run out of TestRunner, ScriptRunner is still used to run quick one-off scripts specific to a single target. Additionally, ScriptRunner makes a great editor for writing TestRunner scripts. It is recommended that users try both tools before deciding which will be best for their program.

## Looping vs Unrolled Loops

Loops are powerful constructs that allow you to perform the same operations multiple times without having to rewrite the same code over and over (See the DRY Concept). However, they can make restarting a COSMOS script at the point of a failure difficult or impossible. If there is a low probability of something failing, then loops are an excellent choice. If a script is running a loop over a list of telemetry points, it may be a better choice to “unroll” the loop by making the loop body into a function, and then calling that function directly for each iteration of a loop that would have occurred.

For example:

```
10.times do |temperature_number|
 check_temperature(temperature_number + 1)
end
```

If the above script was stopped after temperature number 3, there would be no way to restart the loop at temperature number 4. A better solution for small loop counts is to unroll the loop.

```
check_temperature(1)
check_temperature(2)
check_temperature(3)
check_temperature(4)
check_temperature(5)
check_temperature(6)
check_temperature(7)
check_temperature(8)
check_temperature(9)
check_temperature(10)
```

In the unrolled version above, the COSMOS “Start script at selected line” feature can be used to resume the script at any point.

## Script Organization

### Organize Scripts into Test Case Functions

Put each test case into a distinct function. This is the natural way to use TestRunner, but also works well in ScriptRunner. Putting your test cases into functions makes organization easy and gives a great high-level overview of what the overall script does (assuming you name the functions well). There are no bonus points for vague, short function names. Make your function names long and clear.

```

def test_1_heater_zone_control
 puts "Verifies requirements 304, 306, and 310"
 # Test code here
end

```

## Using Classes vs Unscoped Functions

Classes in object-oriented programming allow you to organize a set of related methods and some associated state. The most important aspect is that the methods work on some shared state. For example, if you have code that moves a gimbal around, and need to keep track of the number of moves, or steps, performed across methods, then that is a wonderful place to use a class. If you just need a helper method to do something that happens multiple times in a script without copy and pasting, it probably does not need to be in a class.

```

class Gimbal
 attr_accessor :gimbal_steps
 def move(steps_to_move)
 # Move the gimbal
 @gimbal_steps += steps_to_move
 end
 def home_gimbal
 # Home the gimbal
 @gimbal_steps += steps_moved
 end
end

def perform_common_math(x, y)
 # Do the math and return result
end

gimbal = Gimbal.new
gimbal.home
gimbal.move(100)
gimbal.move(200)
puts "Moved gimbal #{gimbal.steps}"
perform_common_math(gimbal.steps, other_value)

```

## Organizing Your Scripts into Separate Files

As your scripts become large with many methods, it makes sense to break them up into multiple files. Here is a recommended organization for your project's scripts/procedures.

| Folder                                | Description                                                                    |
|---------------------------------------|--------------------------------------------------------------------------------|
| config/targets/TARGET_NAME/lib        | Place script files containing reusable target specific methods here            |
| config/targets/TARGET_NAME/procedures | Place simple procedures that are centered around one specific target here      |
| lib                                   | Place script files containing reusable methods that span multiple targets here |
| procedures                            | Place high-level procedures that span targets here                             |

In your main procedure you will usually bring in the other files with instrumentation using load\_utility.

```
load_utility('my_other_script')
```

## Instrumented vs Uninstrumented Lines (require vs load)

COSMOS scripts are normally “instrumented”. This means that each line has some extra code added behind the scenes that primarily highlights the current executing line and catches exceptions if things fail such as a wait\_check. If your script needs to use code in other files, there are a few ways to bring in that code. Some techniques bring in instrumented code and others bring in uninstrumented code. There are reasons to use both.

load\_utility (and the deprecated require\_utility), bring in instrumented code from other files. When COSMOS runs the code in the other file, ScriptRunner/TestRunner will dive into the other file and show each line highlighted as it executes. This should be the default way to bring in other files, as it allows continuing if something fails, and provides better visibility to operators.

However, sometimes you don’t want to display code executing from other files. Externally developed ruby libraries generally do not like to be instrumented, and code that contains large loops or that just takes a long time to execute when highlighting lines, will be much faster if included in a method that does not instrument lines. Ruby provides two ways to bring in uninstrumented code. The first is the “load” keyword. Load will bring in the code from another file and will bring in any changes to the file if it is updated on the next call to load. “require” is like load but is optimized to only bring in the code from another file once. Therefore, if you use require and then change the file it requires, you must restart ScriptRunner/TestRunner to re-require the file and bring in the changes. In general, load is recommended over require for COSMOS scripting. One gotcha with load is that it requires the full filename including extension, while the require keyword does not.

Finally, COSMOS scripting has a special syntax for disabling instrumentation in the middle of an instrumented script, with the disable\_instrumentation function. This allows you to disable instrumentation for large loops and other activities that are too slow when running instrumented.

```
disable_instrumentation do
 # Make sure nothing in here will raise exceptions!
 5000000.times do
 temp += 1
 end
end
```



### When Running Uninstrumented Code

Make sure that the code will not raise any exceptions or have any check failures. If an exception is raised from uninstrumented code, then your entire script will stop.

## Debugging and Auditing

### Built-In Debugging Capabilities

Both ScriptRunner and TestRunner have built in debugging capabilities that can be useful in determining why your script is behaving in a certain way. Of primary importance is the ability to inspect and set script variables.

To use the debugging functionality, first select the “Toggle Debug” option from the Script Menu. This will add a small Debug: prompt to the bottom of the tool. Any code entered in this prompt will be executed when Enter is pressed. To

inspect variables in a running script, pause the script and then use “puts” to print out the value of the variable in the debug prompt.

```
puts variable_name
```

Variables can also be set simply by using equals.

```
variable_name = 5
```

If necessary, you can also inject commands from the debug prompt using the normal commanding methods. These commands will be logged to the ScriptRunner message log, which may be advantageous over using a different COSMOS tool like CmdSender (where the command would only be logged in the CmdTlmServer message log).

```
cmd("INST COLLECT with TYPE NORMAL")
```

Note that the debug prompt keeps the command history and you can scroll through the history by using the up and down arrows.

## Breakpoints

While in Debug mode (Script -> Toggle Debug), you can right-click at any point in a script in ScriptRunner and select “Add Breakpoint”. This places a breakpoint on the selected line and the script will automatically pause when it hits the breakpoint. Once stopped at the breakpoint, you can evaluate the state of the system using telemetry screens or the built-in debugging capabilities.

## Using Disconnect Mode

Disconnect mode is a feature of ScriptRunner and TestRunner that allows testing scripts in an environment without real hardware in the loop. Disconnect mode is started by selecting Script -> Toggle Disconnect. Once selected, the user is prompted to select which targets to disconnect. By default, all targets are disconnected, which allows for testing scripts without any real hardware. Optionally, only a subset of targets can be selected which can be useful for trying out scripts in partially integrated environments.

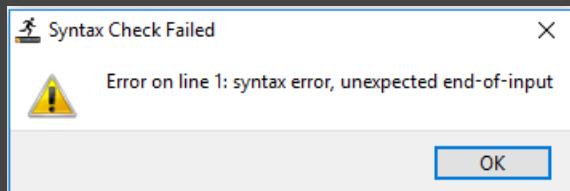
While in disconnect mode, commands to the disconnected targets always succeed. Additionally, all checks of disconnected targets’ telemetry are immediately successful. This allows for a quick run-through of procedures for logic errors and other script specific errors without having to worry about the behavior and proper functioning of hardware.

## Auditing your Scripts

ScriptRunner includes several tools to help audit your scripts both before and after execution.

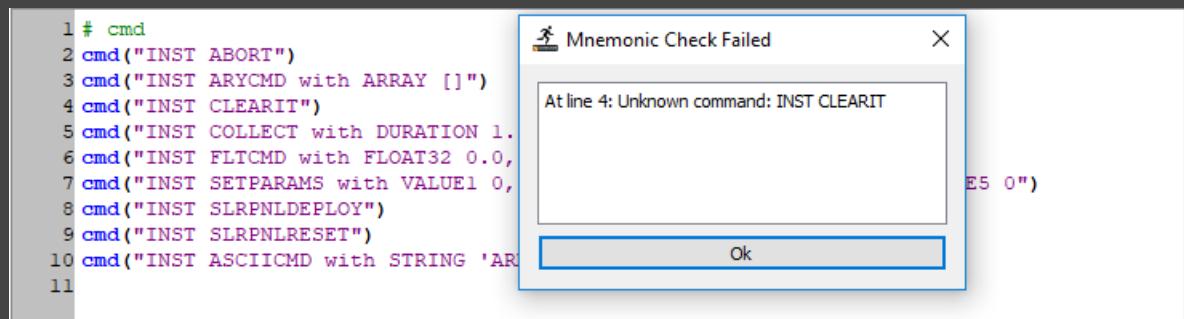
### Ruby Syntax Check

The Ruby Syntax Check tool is found under the Script Menu. This tool uses the ruby executable with the -c flag to run a syntax check on your script. If any syntax errors are found the exact message presented by the Ruby interpreter is shown to the user. These can be cryptic, but the most common faults are not closing a quoted string, forgetting an “end” keyword, or using a block but forgetting the proceeding “do” keyword.



## Mnemonic Check

The mnemonic check uses an algorithm to scan through your script for command and telemetry mnemonics and make sure they are known to COSMOS.



## Generate Cmd/Tlm Audit

After you are done running your scripts, you can execute a Cmd/Tlm audit that will scan the ScriptRunner message log files and look for all cmd() lines and CHECK lines to know how many times each command was sent and how many times each telemetry point was checked. This audit reads one or more message log files and generates a CSV file with a count for each command and telemetry point.

## Common Scenarios

### User Input Best Practices

COSMOS provides several different methods to gather manual user input in scripts. When using user input methods that allow for arbitrary values (like ask() and ask\_string()), it is very important to validate the value given in your script before moving on. When asking for text input, it is extra important to handle different casing possibilities and to ensure that invalid input will either re-prompt the user or take a safe path.

```
answer = ask_string("Do you want to continue (y/n)?")
if answer != 'y' and answer != 'Y'
 raise "User entered: #{answer}"
end

temp = 0.0
while temp < 10.0 or temp > 50.0
 temp = ask("Enter the desired temperature between 10.0 and 50.0")
end
```

When possible, always use one of the other user input methods that has a constrained list of choices for your users (message\_box, vertical\_message\_box, combo\_box).

Note that all these user input methods provide the user the option to “Cancel”. When cancel is clicked, the script is paused but remains at the user input line. When hitting “Go” to the continue, the user will be re-prompted to enter the value.

### Conditionally Require Manual User Input Steps

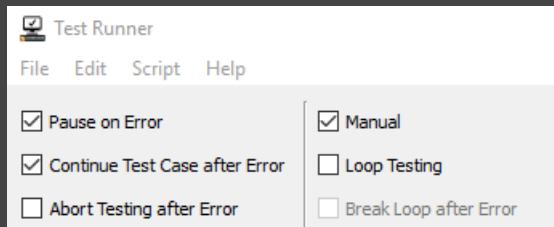
When possible, a useful design pattern is to write your scripts such that they can run without prompting for any user input. This allows the scripts to be more easily tested and provides a documented default value for any user input choices or values. To implement this pattern, all manual steps such as ask(), prompt(), and infinite wait() statements need to be wrapped with an if statement that checks the value of the \$manual variable. If \$manual is set, then the manual steps should be executed. If not, then a default value should be used.

```

Set the $manual variable - Only needed in ScriptRunner
answer = ask("Prompt for manual entry (Y/n)?")
if answer == 'n' or answer == 'N'
 $manual = false
else
 $manual = true
end
if $manual
 temp = ask("Please enter the temperature")
else
 temp = 20.0
end
if !$manual
 puts "Skipping infinite wait in auto mode"
else
 wait
end

```

In TestRunner, there is a checkbox at the top of the tool called “Manual” that affects this \$manual variable directly. In ScriptRunner you can also use the \$manual pattern, but will need to build in an ask() at the top of your script to ask if the user wants to run manual steps, and then set the \$manual variable appropriately.



## Outputting Extra Information to a TestRunner Test Report

COSMOS TestRunner automatically generates a test report that shows the PASS/FAILED/SKIPPED state for each test case. You can also inject arbitrary text into this report with the current test case using Cosmos::Test.puts “Your Text”. Alternatively, you can simply use puts to place text into the ScriptRunner message log (in both TestRunner/ScriptRunner).

```

class MyTest < Cosmos::Test
def test_1
 # The following text will be placed in the test report
 Cosmos::Test.puts "Verifies requirements 304, 306, 310"
 # Do the test here
 # This puts line will show up in the sr_messages log file
 puts "test_1 complete"
end

```

## Getting the Most Recent Value of a Telemetry Point from Multiple Packets

Some systems include high rate data points with the same name in every packet. COSMOS supports getting the most recent value of a telemetry point that is in multiple packets using a special packet name of LATEST. Let’s pretend that our target INST has two packets, PACKET1 and PACKET2. Both packets have a telemetry point called TEMP.

```
Get the value of TEMP from the most recently received PACKET1
value = tlm("INST PACKET1 TEMP")

Get the value of TEMP from the most recently received PACKET2
value = tlm("INST PACKET2 TEMP")

Get the value of TEMP from the most recently received PACKET1 or PACKET2
value = tlm("INST LATEST TEMP")
```

## Checking Every Single Sample of a Telemetry Point

When writing COSMOS scripts, checking the most recent value of a telemetry point normally gets the job done. The `tlm()`, `tlm_raw()`, etc methods all retrieve the most recent value of a telemetry point. Sometimes you need to perform analysis on every single sample of a telemetry point. This can be done using COSMOS's packet subscription system. The packet subscription system lets you choose one or more packets and receive them all from a queue. You can then pick out the specific telemetry points you care about from each packet.

```
id = subscribe_packet_data([["INST", "HEALTH_STATUS"]])
total = 0
100.times do
 packet = get_packet(id)
 value = packet.read("TEMP1")
 total += value
end
average = total / 100.0
unsubscribe_packet_data(id)
```

## Using Variables in Mnemonics

Because command and telemetry mnemonics are just strings in COSMOS scripts, you can make use of variables in some contexts to make reusable code. For example, a function can take a target name as an input to support multiple instances of a target. You could also pass in the value for a set of numbered telemetry points.

```
def example(target_name, temp_number)
 cmd("#{target_name} COLLECT with TYPE NORMAL")
 wait_check("#{target_name} TEMP#{temp_number} > 50.0")
end
```

This can also be useful when looping through a numbered set of telemetry points but be considerate of the downsides of looping as discussed in the [Looping vs Unrolled Loops](#) section.

## Using Custom `wait_check_expression`

COSMOS's `wait_check_expression` (and `check_expression`) allow you to perform more complicated checks and still stop the script with a CHECK error message if something goes wrong. For example, you can check variables against each other or check a telemetry point against a range. The exact string of text passed to `wait_check_expression` is repeatedly evalued in Ruby until it passes, or a timeout occurs. It is important to not use Ruby string interpolation `#{}`  within the actual expression or the values inside of the Ruby interpolation syntax `#{}`  will only be evaluated once when it is converted into a string. PROTIP: Using `#{}`  inside a comment inside the expression can give more insight if the expression fails, but be careful as it will show the first evaluation of the values when the check passes which can be confusing if they go from failing to passing after waiting a few seconds.

```

one = 1
two = 2

wait_check_expression("one == two", 1)
ERROR: CHECK: one == two is FALSE after waiting 1.017035 seconds

With PROTIP to see the values at the first evaluation of the expression
wait_check_expression("one == two # #{one} == #{two}", 1)
ERROR: CHECK: one == two # 1 == 2 is FALSE after waiting 1.015817 seconds

Checking an integer range
wait_check_expression("one > 0 and one < 10 # init value one = #{one}", 1)

```

## COSMOS Scripting Differences from Regular Ruby Scripting

### **Do not use single line if statements**

COSMOS scripting instruments each line to catch exceptions if things go wrong. With single line if statements the exception handling doesn't know which part of the statement failed and cannot properly continue. If an exception is raised in a single line if statement, then the entire script will stop and not be able to continue. Do not use single line if statements in COSMOS scripts. (However, they are fine to use in interfaces and other Ruby code, just not COSMOS scripts).

Don't do this:

```
run_method() if tlm("INST HEALTH_STATUS TEMP1") > 10.0
```

Do this instead:

```

It is best not to execute any code that could fail in an if statement, ie
tlm() could fail if the CmdTlmServer was not running or a mnemonic
was misspelled
temp1 = tlm("INST HEALTH_STATUS TEMP1")
if temp1 > 10.0
 run_method()
end

```

### **Methods should use explicit return statements**

In COSMOS 4.2 and earlier, COSMOS script instrumentation alters the implicit return value of each line. Therefore, when returning from instrumented methods, an explicit "return" keyword should always be used (Normal Ruby will return the result of the last line of a method). Note: This issue has been resolved in COSMOS 4.3, but all earlier versions must use explicit returns.

Don't do this:

```

def add_one(x)
 x + 1
end

```

Do this instead:

```
def add_one(x)
 return x + 1
end
```

## When Things Go Wrong

### Common Reasons Checks Fail

There are three common reasons that checks fail in COSMOS scripts:

1. The delay given was too short

The `wait_check()` function takes a timeout that indicates how long to wait for the referenced telemetry point to pass the check. The timeout needs to be large enough for the system under test to finish its action and for updated telemetry to be received. Note that the script will continue as soon as the check completes successfully. Thus, the only penalty for a longer timeout is the additional wait time in a failure condition.

2. The range or value checked against was incorrect or too stringent

Often the actual telemetry value is ok, but the expected value checked against was too tight. Loosen the ranges on checks when it makes sense. Ensure your script is using the `wait_check_tolerance()` routine when checking floating point numbers and verify you're using an appropriate tolerance value.

3. The check really failed

Of course, sometimes there are real failures. See the next section for how to handle them and recover.

### How to Recover from Anomalies

Once something has failed, and your script has stopped with a pink highlighted line, how can you recover? Fortunately, COSMOS provides several mechanisms that can be used to recover after something in your script fails.

1. Retry

After a failure, the ScriptRunner/TestRunner “Pause” button changes to “Retry”. Clicking on the Retry button will re-execute the line that failed. For failures due to timing issues, this will often resolve the issue and allow the script to continue. Make note of the failure and be sure to update your script prior to the next run.

2. Execute Selected Lines While Paused

Sometimes re-executing a command or a few other lines of a script can correct problems. This can happen when commanding is over an unreliable transport layer such as UDP or a noisy serial line. For these scenarios, users can highlight the lines of the script they want to run again, right-click, and select “Execute Selected Lines While Paused”. This will run the selected lines again with the full script context (all required variables will still be in scope), and then return. Afterwards you can retry the line that failed or just proceed with “Go”.

3. Use the Debug Prompt

By selecting Script -> Toggle Debug, you can perform arbitrary actions that may be needed to correct the situation without stopping the running script. You can also inspect variables to help determine why something failed.

4. Log Message to Script Log

Not necessarily a correction for a failure, but you can log notes or QA approval that occurred after a failure using

## Script -> Log Message to Script Log.

If you do need to stop your script and restart, COSMOS also provides several methods to prevent restarting the script from the beginning.

### 1. Execute From Cursor

By clicking into a script, and right clicking to select “Execute From Cursor”, users can restart a script at an arbitrary point. This works well if no required variable definitions exist earlier in the script. A workaround can be to start using “Execute from Cursor”, immediately pause (by hitting pause or by previously setting a breakpoint), and then “Run Selected Lines While Paused” to bring in the necessary variable declarations.

### 2. Execute Selected Lines

If only a small section of a script needs to be run, then “Execute Selected Lines” can be used to execute only a small portion of the script.

## Advanced Topics

### Advanced Script Configuration with CSV or Excel

Using a spreadsheet to store the values for use by a script can be a great option if you have a CM-controlled script but need to be able to tweak some values for a test or if you need to use different values for different serial numbers.

The Ruby CSV class be used to easily read data from CSV files (recommended for cross platform projects).

```
require 'csv'
values = CSV.read('test.csv')
puts values[0][0]
```

If you are only using Windows, COSMOS also contains a library for reading Excel files.

```
require 'cosmos/win32/excel'
ss = ExcelSpreadsheet.new('C:/git/COSMOS/demo/test.xlsx')
puts ss[0][0][0]
```

### Script specific screens

Starting with COSMOS 4.3, script writers can include temporary screens in their COSMOS scripts that show just the specific values relative to the script. They can even display local variables as shown below. This can be a fantastic way to display just the telemetry that is specifically relevant to what you are operating or testing. Screen definitions take the same format as normal COSMOS screens with the addition of using target name LOCAL and packet name LOCAL to gain access to script local variables. See the [local\\_screen](#) documentation in the [Scripting Guide](#).

### When to use Modules

Modules in Ruby have two purposes: namespacing and mixins. Namespacing allows having classes and methods with the same name, but with different meanings. For example, if they are namespaced, COSMOS can have a Packet class and another Ruby library can have a Packet class. This isn’t typically useful for COSMOS scripting though.

Mixins allow adding common methods to classes without using inheritance. Mixins can be useful in TestRunner to add common functionality to some Test classes but not others, or to break up Test classes into multiple files.

```
module MyModule
 def module_method
 end
end

class MyTest < Cosmos::Test
 include MyModule
 def test_1
 module_method()
 end
end
```

## Further Reading

Please see the [Scripting Guide](#) for the full list of available scripting methods provided by COSMOS.

◀ BACK

NEXT ▶

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the [Jekyll](#) project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

## Gem Based Targets and Tools

Improve this page

COSMOS supports sharing and reusing targets and tools by bundling the target and tool configuration and code into Ruby gems. This document provides the information necessary to use and create gem based targets and tools.

### Using Gem Based Targets

Step one is to install the gem based target into your COSMOS project by adding a line like the following to your project Gemfile:

```
Gemfile
gem 'cosmos-xxxxxx' # Name of your gem here. All should start with cosmos-
```

After making the Gemfile modification above, you can install the gem by running `bundle install` in your COSMOS project folder. Note if the gem is not hosted at rubygems.org and you just have the file locally, you will most likely need to manually `gem install cosmos-xxxxxx.gem` before running `bundle install`.

Step two is to let the COSMOS system know about the gem based target in config/system/system.txt:

```
system.txt

Declare all installed gem based targets in your Gemfile
AUTO_DECLARE_TARGETS # This will discover any gem based targets automatically

Individually specify targets (and possibly rename)
DECLARE_GEM_TARGET cosmos-xxxxxx
```

Step three is to configure the interface to the target in config/tools/cmd\_tlm\_server/cmd\_tlm\_server.txt. This is done exactly the same way as any other target.

Step four is to configure telemetry screens in config/tools/tlm\_viewer/tlm\_viewer.txt. This is again done exactly the same way as any other target.

That's it! You should now be able to connect and interact with your gem based target.

### Using Gem Based Tools

First, install the gem based tool into your COSMOS project in the same way as installing a target gem above. Then you will need to tell the COSMOS Launcher about the tool in launcher.txt like so:

```
launcher.txt

Provide buttons for all gem based tools
AUTO_GEM_TOOLS

Or a just specific one
TOOL "XXXXXX" "LAUNCH_GEM Xxxxxxx" "xxxxxxxx.png"
```

That's it! Click the new button in launcher to launch the tool.

## Creating Gem Based Targets

Creating gem based targets is easy. All that is required is creating a cosmos-xxxxxx.gemspec (replace xxxxxx with your target name) file in the target folder you wish to make into a gem. The folder structure should be just like what is in a normal target folder (ie.):

```
├── cosmos-xxxxxx.gemspec
├── cmd_tlm_server.txt
├── target.txt
└── cmd_tlm
 └── # Target command and telemetry definition files here
└── lib
 └── # Any necessary target code here
└── screens
 └── # Target telemetry screen files
```

The gem name must start with “cosmos-”, and should then be followed by the actual target name. For example: cosmos-apcpdu.gemspec could be the gem name for a target called APCPDU.

Example config/targets/XXXXXX/cosmos-xxxxxx.gemspec file:

```

encoding: ascii-8bit

require 'rbconfig'

Create the overall gemspec
spec = Gem::Specification.new do |s|
 s.name = 'cosmos-xxxxxx' # UPDATE WITH YOUR GEM NAME (must start with cosmos-)
 s.summary = 'Ball Aerospace COSMOS target' # UPDATE
 s.description = <<-EOF
 Example gem based target # UPDATE
 EOF
 s.authors = ['Your Name'] # UPDATE
 s.email = ['yourname@yourcompany.com'] # UPDATE

 s.platform = Gem::Platform::RUBY
 if ENV['VERSION']
 s.version = ENV['VERSION'].dup
 else
 s.version = '0.0.0'
 end
 s.license = 'GPL-3.0' # UPDATE

 # Modify as needed
 s.files = Dir['lib/*'] + Dir['cmd_tlm/*'] + Dir['screens/*'] + ['cmd_tlm_server.txt', 'target.txt']

 s.has_rdoc = true

 s.required_ruby_version = '~> 2'

 # Runtime Dependencies
 s.add_runtime_dependency 'cosmos', '~> 3', '>= 3.7.0'
end

```

After organizing the files as required and creating the gemspec, create the actual gem with the following:

```

Windows - update VERSION as needed
set VERSION=1.0.0
gem build cosmos-xxxxxx.gemspec

Linux/Mac - update VERSION as needed
export VERSION=1.0.0
gem build cosmos-xxxxxx.gemspec

```

To publish your gem for other COSMOS users consider putting the source on [Github](#) and publishing your gem to [Rubygems](#).

## Creating Gem Based Tools

Creating a gem based tool is very similar to creating a gem based target. However, generally it will need to be done outside of your COSMOS project folder otherwise careful crafting in the “files” section of the gemspec file is required. In general, you will need to create a cosmos-xxxxxx.gemspec (replace xxxx with your tool name) file and a directory

structure like this:

```
|── cosmos-xxxxxx.gemspec
|── config
| └── data
| └── xxxxxx.png # Tool icon
└── lib
 └── # Tool code files
└── tools
 └── XXXXXX # Tool starting script (see Launcher, etc)
 └── mac
 └── XXXXXX # Mac Tool starting script (see Launcher.app, etc)
```

The gem name must start with “cosmos-”, and should then be followed by the actual tool name. For example: cosmos-satvis.gemspec could be the gem name for a tool called Satvis.

Example cosmos-xxxxxx.gemspec file for a tool:

```
encoding: ascii-8bit

require 'rbconfig'

Create the overall gemspec
spec = Gem::Specification.new do |s|
 s.name = 'cosmos-xxxxxx' # UPDATE WITH YOUR GEM NAME (must start with cosmos-)
 s.summary = 'Ball Aerospace COSMOS based tool' # UPDATE
 s.description = <<-EOF
 Example gem based tool # UPDATE
 EOF
 s.authors = ['Your Name'] # UPDATE
 s.email = ['yourname@yourcompany.com'] # UPDATE

 s.platform = Gem::Platform::RUBY
 if ENV['VERSION']
 s.version = ENV['VERSION'].dup
 else
 s.version = '0.0.0'
 end
 s.license = 'GPL-3.0' # UPDATE

 # Modify as needed
 s.files = Dir['config/data/*'] + Dir['lib/*'] + Dir['tools/**/*']

 s.has_rdoc = true

 s.required_ruby_version = '~> 2'

 # Runtime Dependencies
 s.add_runtime_dependency 'cosmos', '~> 3', '>= 3.7.0'
end
```

After organizing the files as required and creating the gemspec, create the actual gem with the following:

```
Windows - update VERSION as needed
set VERSION=1.0.0
gem build cosmos-xxxxxx.gemspec

Linux/Mac - update VERSION as needed
export VERSION=1.0.0
gem build cosmos-xxxxxx.gemspec
```

To publish your gem for other COSMOS users consider putting the source on [Github](#) and publishing your gem to Rubygems.

 BACK  NEXT 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the [Jekyll](#) project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

## Chaining CmdTlmServers

Improve this page

There are two ways to chain CmdTlmServers to achieve two distinctly different results. One is to chain a client CmdTlmServer to a master to enable a separate workstation to have access to all the command and telemetry. The second is to connect a master CmdTlmServer to a client to allow a remote client to act as an interface to a target. Both methods will be discussed here.

### Chain Client CmdTlmServer to Master

Chaining a client CmdTlmServer to a master CmdTlmServer allows for running the COSMOS tools on other workstations and removing that processing load from a master CmdTlmServer that is directly connected to your targets. This is great for setting up workstations dedicated to graphing data, or viewing telemetry, without disturbing the main operation as the client will get all the same information as the master CmdTlmServer.

#### Master CmdTlmServer Configuration

The default COSMOS Configuration already includes everything necessary to chain “child” CmdTlmServers. There is a default router called the PREIDENTIFIED\_ROUTER that is listening on port 7779 by default. This is used to chain CmdTlmServers and is also used by TlmGrapher to access the full telemetry stream.

The only issue on the master CmdTlmServer computer is that you must make sure that the firewall is either disabled or that access to port 7779 is permitted.

#### Child CmdTlmServer Configuration

1. Modify the example cmd\_tlm\_server\_chain.txt file below
  - Change localhost to the IP address of your master CmdTlmServer
  - Update the TARGET keywords to include all of your targets
2. Put the updated cmd\_tlm\_server\_chain.txt into config/tools/cmd\_tlm\_server/cmd\_tlm\_server\_chain.txt
3. Start CmdTlmServer with (probably create a modified launcher.txt and associated .bat file):
  - `ruby CmdTlmServer --config cmd_tlm_server_chain.txt`

#### Example CmdTlmServer Configuration for Child (cmd\_tlm\_server\_chain.txt)

```

Using this file WITH LOCALHOST requires changing the ports in system.txt
Otherwise don't change the ports!

TITLE 'COSMOS Command and Telemetry Server - Chain Configuration'

Don't log on the chained server
PACKET_LOG_WRITER DEFAULT packet_log_writer.rb nil false

Replace localhost below with the IP Address of the master CmdTlmServer
Update the target list below to the full list of targets in your system
To make this child unable to send commands change the first 7779 to nil
INTERFACE CHAININT tcpip_client_interface.rb localhost 7779 7779 10 5 PREIDENTIFIED

TARGET INST
TARGET INST2
TARGET EXAMPLE
TARGET TEMPLATED
TARGET COSMOS

```

## Connect Master CmdTlmServer to Client

Connecting a master CmdTlmServer to a client CmdTlmServer is used when a client machine needs to interface directly to a target but the master CmdTlmServer also wants to view that target. For example, a machine is physically located next to a target which requires a serial interface (thus a serial cord connection) but the master CmdTlmServer is across the room. This technique connects the local machine to the target and then connects the master CmdTlmServer to the client. The client doesn't need to define the full configuration of the server and only has to configure the local target it is interfacing with.

### Child CmdTlmServer Configuration

The child COSMOS configuration should be configured as a totally stand alone COSMOS system. Follow the rest of the COSMOS documentation to set up the command and telemetry definitions and the interface definition in the target's cmd\_tlm\_server.txt file. When this is complete, you will have a config/targets/<TARGET> folder with your target's definition.

The client CmdTlmServer defines a default router called the PREIDENTIFIED\_ROUTER that is listening on port 7779 by default. This will be used by the master CmdTlmServer to connect. Ensure that the firewall is either disabled or that access to port 7779 is permitted.

### Master CmdTlmServer Configuration

The master config/tools/cmd\_tlm\_server/cmd\_tlm\_server.txt file is where you define the interface to connect to the child CmdTlmServer.

1. Modify the example cmd\_tlm\_server.txt file below
  - Change localhost to the IP address of your client CmdTlmServer
  - Update <TARGET> to match the target name from the client
2. Ensure your system.txt file has either AUTO\_DECLARE\_TARGETS or explicitly declares the target via  
DECLARE\_TARGET <TARGET>
3. Ensure your config/targets/<TARGET> folder matches the target definition on the client
  - Since the child and master CmdTlmServer share the same target definition, it is convenient to make this target an SVN external in one of the COSMOS configurations to ensure consistency between them

### Example CmdTlmServer Configuration for Master (cmd\_tlm\_server.txt)

```
TITLE 'COSMOS Command and Telemetry Server'

PACKET_LOG_WRITER DEFAULT packet_log_writer.rb

Replace localhost below with the IP Address of the client CmdTlmServer
Change <TARGET> to the target name as defined in the client
INTERFACE <TARGET>_INT tcpip_client_interface.rb localhost 7779 7779 nil nil PREIDENTIFIED
TARGET <TARGET>
```

BACK

NEXT

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

## System Class

Improve this page

### This documentation is for COSMOS Developers

If you're simply trying to setup a COSMOS system you're probably looking for the [System Configuration](#) page. If you're trying to create a custom interface, background task, conversion, or build a custom tool then this is the right place.

The System class is the primary entry point into the COSMOS framework. It provides access to the targets, commands, and telemetry. It also captures system wide configuration items such as the available ports and paths used by the system. The System class is primarily responsible for loading the system configuration file and creating all the Target instances. It also saves and restores configurations using a MD5 checksum over the entire configuration to detect changes.

The [system.rb](#) source code on Github.

## Programming System

Almost all custom COSMOS code needs to interact with System as it provides access to the COSMOS command and telemetry. The System class is implemented as a [singleton](#) which basically means there is only one instance of the class. This makes sense because COSMOS can only have a single instance which controls access to its internal state.

### System.commands

[System.commands](#) provides access to all the command definitions in the COSMOS system. The primary developer access methods are:

1. `System.commands.target_names` - Returns an array of strings containing the target names
2. `System.commands.all` - Returns a hash keyed by the target name with a hash of Packets as the value. The second hash is identical to what is returned by `System.commands.packets("TARGET")`.
3. `System.commands.packets("TARGET")` - Returns a Hash keyed by the packet name with the Packet instance as the value
4. `System.commands.packet("TARGET", "PACKET")` - Returns the given Packet instance
5. `System.commands.identify(data)` - Identify a raw buffer of data as a Packet and return the Packet instance.

Additional methods are available which are not as commonly used:

1. `System.commands.build_cmd("TARGET", "PACKET", params)` - Creates a Packet instance initialized with the values in the params hash.
2. `System.commands.format(packet)` - Returns a string which represents how to send this command in Script Runner. For example: Given a COSMOS start logging command instance it returns "cmd('COSMOS STARTLOGGING')"

3. `System.commands.cmd_pkt_hazardous?(command)` - Returns an array where the first boolean value indicates whether the given command is hazardous or not. If the first value is true (hazardous), the second value is a string with information about the hazard.
4. `System.commands.cmd_hazardous?("TARGET", "PACKET", params)` - Returns the same data as `cmd_pkt_hazardous?` above.

Other methods are available but generally should not be used by developers.

### Command Sender Example

COSMOS uses `System.commands` in many of its own applications. Let's see how it's used in the [Command Sender](#). In the `update_targets` method it uses `System.commands.target_names` to populate the target drop down selection.

```
def update_targets
 @target_select.clearItems()
 target_names = System.commands.target_names
 ... # Code to check for hidden commands
 target_names.each do |target_name|
 @target_select.addItem(target_name)
 end
end
```

Once it checks for hidden commands it adds all the target names to the drop down selection in the tool. In the same way the `update_commands` method accesses `System.commands.packets` to update the packet drop down selection.

```
def update_commands
 @cmd_select.clearItems()
 target_name = @target_select.text
 if target_name
 commands = System.commands.packets(@target_select.text)
 command_names = []
 commands.each do |command_name, command|
 command_names << command_name unless command.hidden
 end
 command_names.sort!
 command_names.each do |command_name|
 @cmd_select.addItem(command_name)
 end
 end
end
```

Later in the `update_cmd_params` method we access the individual command to grab the command parameters:

```

def update_cmd_params(ignored_toggle = nil)
 ...
 target_name = @target_select.text
 target = System.targets[target_name]
 packet_name = @cmd_select.text
 if target_name and packet_name
 packet = System.commands.packet(target_name, packet_name)
 packet_items = packet.sorted_items
 ...
 end
end

```

If the target and packet selections have been set, we grab the specified packet using System.commands.packet and then have access to the packet items through packet.sorted\_items.

### Interface Example

Sometimes when you're creating a custom interface you want to respond to a COSMOS command within the interface itself and not forward on that command to the target. In the interface's connect method you can get a handle to the command you're interested in.

```

require 'cosmos/interfaces/tcpip_client_interface'
module Cosmos
 class TestInterface < Interface
 def connect
 super()
 @configure = System.commands.packet(@target_names[0], 'CONFIGURE')
 end
 end
end

```

In this example, we inherit from the COSMOS TcpipClientInterface. We grab a handle to the 'CONFIGURE' packet by using the @target\_names array. This array is populated by the COSMOS Server when the target is assigned. This allows you to dynamically get your target name since targets can be renamed by the server.

In the interface's write method we can check for the previously saved packet.

```

Defined inside the TestInterface class
def write(packet)
 if @configure.identify?(packet.buffer)
 value = packet.read("VALUE") # Do something ...
 else
 super(packet) # Allow TcpipClientInterface to write the packet
 end
end

```

We use the Packet class's identify? method to determine if the packet passed in is the one we're interested in. Then we can read values from the packet and take whatever actions we want. If this is not the 'CONFIGURE' packet then we call super(packet) to allow the TcpipClientInterface logic to send the packet to the target.

## System.telemetry

System.telemetry provides access to all the telemetry definitions in the COSMOS system. The primary developer access methods are:

1. `System.telemetry.target_names` - Returns an array of strings containing the target names
2. `System.telemetry.all` - Returns a hash keyed by the target name with a hash of Packets as the value. The second hash is identical to what is returned by System.telemetry.packets("TARGET").
3. `System.telemetry.packets("TARGET")` - Returns a Hash keyed by the packet name with the Packet instance as the value
4. `System.telemetry.packet("TARGET", "PACKET")` - Returns the given Packet instance
5. `System.telemetry.items("TARGET", "PACKET")` - Returns an array of PacketItem instances for the given target and packet
6. `System.telemetry.value("TARGET", "PACKET", "ITEM")` - Returns the telemetry value. Note this can take a fourth parameter indicating how to format the value.
7. `System.telemetry.identify!(data)` - Identify a raw buffer of data as a Packet and return the Packet instance.

Additional methods are available which are not as commonly used:

1. `System.telemetry.item_names("TARGET", "PACKET")` - Returns an array of item name strings for the given target and packet
2. `System.telemetry.packet_and_item("TARGET", "PACKET", "ITEM")` - Returns an array where the first item is the Packet instance and second item is the PacketItem instance
3. `System.telemetry.set_value("TARGET", "PACKET", "ITEM", value)` - Sets a telemetry value in a packet. Note that as soon as a new packet is received from the target this value will be overwritten.
4. `System.telemetry.latest_packets("TARGET", "ITEM")` - Returns an array of Packet instances with the specified target and item
5. `System.telemetry.values_and_limits_states([["TARGET", "PACKET", "ITEM"], ...])` - Returns an array of three arrays: The first contains the item(s) value, the second the item(s) limits state, and the third the item(s) limits settings.
6. `System.telemetry.stale` - Returns an array of all stale Packet instances. Packets are defined as stale if they haven't been received for System.staleness\_seconds.

Other methods are available but generally are not used by developers.

### Packet Viewer Example

COSMOS uses System.telemetry in many of its own applications. Let's see how it's used in the [Packet Viewer](#). In the `update_targets` method it uses System.telemetry.target\_names and System.telemetry.packets.

```

def update_targets
 @target_select.clearItems

 System.telemetry.target_names.each do |target_name|
 packets = System.telemetry.packets(target_name)
 has_non_hidden = false
 packets.each do |packet_name, packet|
 next if packet.hidden
 has_non_hidden = true
 break
 end
 @target_select.addItem(target_name) if has_non_hidden
 end
end

```

First we grab all the target names and then grab all the packets for each target. This is done to filter out targets in which all packets are hidden. Once it checks for hidden packets it adds all the target names to the drop down selection in the tool. In the same way the `update_packets` method access System.telemetry.packets to update the packet drop down selection.

Later in the `update_tlm_items` method we call System.telemetry.items to get the individual packet items.

```

def update_tlm_items(featured_item_name = nil)
 target_name = @target_select.text
 packet_name = @packet_select.text
 ...
 System.telemetry.items(target_name, packet_name).each do |item|
 tlm_items << [item.name, item.states, item.description, item.data_type == :DERIVED]
 end
 ...
end

```

Once we have the items we call individual `PacketItem` methods to get the name, states, description, and data type.

### Interface Example

Sometimes you want to create a fake interface which returns internally generated data instead of returning data from an external device. To do this you need to populate the packets and return them from the interface's read method. In the interface's initialize method we setup an array to store the packet instances. In the connect method We grab a handle to all the target's packets by using the `@target_names` array. This array is populated by the COSMOS Server when the target is assigned. This allows you to dynamically get your target name since targets can be renamed by the server.

```
require 'cosmos/interfaces/interface'

module Cosmos
 class TestInterface < Interface
 def initialize
 super()
 @pending_packets = Array.new
 @next_read_time = nil
 end

 def connect
 # Note we do NOT call super() here because the Interface base class simply
 # raises an exception. This forces us to reimplement it in derived classes.
 @next_read_time = Time.now
 @connected = true
 @packets = System.telemetry.packets(@target_names[0])
 end

 def connected?
 # No super (see connect)
 @connected
 end

 def disconnect
 # No super (see connect)
 @connected = false
 end
 end
end
```

In the interface's read method we both populate the packets and return them.

```

Defined inside the TestInterface class
def read(packet)
 if not @pending_packets.empty?
 @read_count += 1
 # Pop off the packet instance and clone it so it is standalone
 return @pending_packets.pop.clone
 end

 # Calculate time to sleep to make ticks 1s apart
 delta = @next_tick_time - Time.now
 sleep(delta) if delta > 0.0 # sleep up to 1s
 @next_tick_time += 1

 @packets.each do |name, packet|
 # Populate your packets with calculated values
 packet.write("DATA", "Sample data")
 @pending_packets.push(packet)
 end

 @read_count += 1
 @pending_packets.pop.clone # Return the first pending packet
end

```

The COSMOS Server will call the interface call method as rapidly as it can. The read method must return a Packet instance or nil to disconnect. Initially we check the pending\_packets array for available packets and return them. Once the pending\_packets array has been emptied we sleep a second to allow our interface to operate at 1Hz. An exercise left to the reader would be to pass the rate to the interface and use that. Next we iterate through the previously stored packet list, populate any values we want, and push the packets back on the pending\_packets array. Finally we pop off the first packet on the pending packets array.

## System.limits

[System.limits](#) provides access to all the limits definitions in the COSMOS system. The primary developer access methods are:

1. `System.limits.sets` - Returns an array of symbols defining the system limits sets
2. `System.limits.out_of_limits` - Returns an array indicating all items that are out of limits. The array values are arrays formatted as ["TARGET", "PACKET", "ITEM", :LIMIT\_STATE]. The last item is a symbol indicating the current limit state.
3. `System.limits.overall_limits_state` - Returns a symbol indicating the current overall limits state.
4. `System.limits.groups` - Returns a hash whose keys are the string limit group name and values are an array. Each item in the array is formatted as ["TARGET", "PACKET", "ITEM"] and thus identifies all the items in that particular limits group.
5. `System.limits.enable_group("GROUP")` - Enable a limits group
6. `System.limits.disable_group("GROUP")` - Disable a limits group

Additional methods are available which are not as commonly used:

1. `System.limits.enabled?("TARGET", "PACKET", "ITEM")` - Returns whether limits are enabled for the given item

2. `System.limits.enable("TARGET", "PACKET", "ITEM")` - Enable limit checking for the given item
3. `System.limits.disable("TARGET", "PACKET", "ITEM")` - Disable limit checking for the given item
4. `System.limits.get("TARGET", "PACKET", "ITEM")` - Return information about an items limits. The values are returned in an array as [limits\_set, persistence, enabled, red\_low, yellow\_low, red\_high, yellow\_high, green\_low (optional), green\_high (optional)]

Other methods are available but generally are not used by developers.

### Example

In general you should use the CmdTlmServer API methods instead of the System.limits methods directly.

## System.ports

System.ports returns a hash keyed by the name of the port with the value the port number. This hash will always contain the known COSMOS ports of 'CTS\_API', 'TLMVIEWER\_API', 'CTS\_PREIDENTIFIED', and 'CTS\_CMD\_ROUTER'.

## System.paths

System.paths returns a hash keyed by the name of the path with the value the file system path. This hash typically contains the known COSMOS paths of 'LOGS', 'TMP', 'SAVED\_CONFIG', 'TABLES', 'HANDBOOKS', 'PROCEDURES'.

## System.targets

System.targets returns a hash keyed by the name of the target with Target instance values. The Target instance has instance variables which return useful information about the target. The most frequently used are:

1. name - Name of the target
2. ignored\_parameters - Array of parameters which should be ignored by various tools
3. ignored\_items - Array of items which should be ignored by various tools
4. interface - Interface instance which is mapped to this target
5. cmd\_cnt - The number of command packets sent to this target
6. tlm\_cnt - The number of telemetry packets received from this target

Other methods are available but generally are not used by developers.

 BACK    NEXT 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

## Packet Class

Improve this page

### This documentation is for COSMOS Developers

If you're simply trying to setup a COSMOS system you're probably looking for the [System Configuration](#) page. If you're trying to create a custom interface, background task, conversion, or build a custom tool then this is the right place.

The Packet class is used to access the command and telemetry packet instances. The primary way to access Packet instances is through the [System](#) class. The Packet class provides access to information about the packet as well as all the packet items.

The [packet.rb](#) source code on Github.

## Programming Packet

Most custom COSMOS code needs to interact with Packet as it provides access to the internal values defined by your command and telemetry definitions. Packet inherits from Structure which it uses to provide some of the lower level functionality associated with reading and writing packets. Typically you shouldn't need to directly use any of the Structure methods. Packet instances can be either commands or telemetry and there is no way to directly know whether a packet instance is a command or telemetry packet. However, since you're typically getting packet instances through `System.commands` or `System.telemetry` this is rarely an issue in practice.

### Packet Instance Variables

Packet defines a large number of instance variables that provide information about the packet. The most commonly used are as follows:

1. `target_name` - Target name as a string
2. `packet_name` - Packet name as a string
3. `description` - Packet description as a string
4. `received_time` - Time object representing when this packet was received by the COSMOS Server
5. `received_count` - Number of times this packet was received by the COSMOS Server
6. `items` - Hash of all the `items` keyed by the uppercase item name
7. `sorted_items` - Array of all the `items` sorted by bit\_offset

If you're dealing with a Command packet instance there are additional instance variables that are useful:

1. `hazardous` - Boolean indicating whether the command is hazardous (see `hazardous`)
2. `hazardous_description` - String description of why the packet is hazardous
3. `hidden` - Boolean indicating whether this packet is hidden (see `hidden`)

- disabled - Boolean indicating whether this packet is disabled (see [disabled](#))

## Packet Methods

### read

Read is used to retrieve packet values. The parameters are as follows:

| PARAMETER  | DESCRIPTION                                                                                                       |
|------------|-------------------------------------------------------------------------------------------------------------------|
| name       | Name of the item to read                                                                                          |
| value_type | How the value should be read. Must be one of :RAW, :CONVERTED, :FORMATTED, or :WITH_UNITS. Defaults to :CONVERTED |
| buffer     | Raw data buffer to read the value from (defaults to the current packet)                                           |

Example:

```
value = packet.read('ITEM') # Converted value (default)
value = packet.read("ITEM", :RAW) # Raw value (no COSMOS conversions applied)
value = packet.read("ITEM", :FORMATTED) # String formatted value
value = packet.read("ITEM", :WITH_UNITS) # String formatted value with units
value = packet.read("ITEM", :CONVERTED, buffer) # Read the value from the passed buffer (RARELY USED)
```

First a comment about syntax. The first parameter in the read method is a string which in Ruby can be either single or double quotes. The second parameter is a Ruby Symbol which is similar to a string but is prefixed with a colon.

Note that there are four different ways to read a packet value as indicated above in the value\_type description.:RAW means to return the value defined at that location in the packet without applying any COSMOS conversions or STATE names.:CONVERTED (the default) means to apply COSMOS conversions and any defined State values (see [STATE](#)).:FORMATTED means to apply any format strings on the packet item (see [FORMAT\\_STRING](#)) and return the resulting string. Note that :FORMATTED always returns a STRING even if there is no formatting being applied. This is distinct from :RAW and :CONVERTED which return a value of the type you specify in the packet definition. Finally :WITH\_UNITS applies any units (see [UNITS](#)) to the formatted value and again returns a string value.

The final parameter is buffer. Typically you will just leave this off and write to the buffer contained by the current packet instance. However, if you have an unidentified buffer of data that you believe represents a particular packet instance, you can pass that into read.

### write

Write is used to set packet values. Keep in mind that if you're trying to write values to a packet which is part of a live telemetry stream, your value will be overwritten by the interface and never appear. The parameters are as follows:

| PARAMETER  | DESCRIPTION                                                                                |
|------------|--------------------------------------------------------------------------------------------|
| name       | Name of the item to read                                                                   |
| value      | Value to write                                                                             |
| value_type | How the value should be written. Must be either :RAW or :CONVERTED. Defaults to :CONVERTED |
| buffer     | Raw data buffer to read the value from (defaults to the current packet)                    |

Example:

```
packet.write('ITEM', value) # Converted value
packet.write("ITEM", value, :RAW) # Raw value
```

See [read](#) for a note about the syntax and what value\_type means.

There are a few gotchas associated with writing a value into a packet. If you're writing a value marked as STRING then you can pass almost any value you want as Ruby will automatically convert it into a string. If the packet item has any [write conversions](#) then using the :RAW value\_type will bypass that conversion. For example:

```
packet.write('STRING_ITEM', 10) # 10 will be converted to '10' automatically
packet.write("STRING_ITEM", 12.5, :RAW) # 12.5 will be converted to '12.5'
```

If you're writing into an integer or float value you must pass in something that can be converted to that value. This means that Strings will not work. However, float values work for integers (they are truncated) and integers work for floats. For example:

```
packet.write("INT_ITEM", "STRING") #=> ArgumentError : invalid value for Integer(): "STRING"
packet.write("INT_ITEM", 10.5) #=> packet.read("INT_ITEM") returns 10
packet.write("FLOAT_ITEM", 10) #=> packet.read("FLOAT_ITEM") returns 10.0
```

If there are no [write conversions](#) or [states](#) defined on the telemetry item then writing with :RAW or :CONVERTED will be equivalent.

Definition File:

```
APPEND_ITEM INT_ITEM_WITH_STATES 32 UINT "Item"
STATE OFF 0
STATE ON 1
```

Example:

```
packet.write("INT_ITEM_WITH_STATES", "ON") # Writes using the "ON" state value
packet.read("INT_ITEM_WITH_STATES", :RAW) #=> returns 1
packet.write("INT_ITEM_WITH_STATES", 0) # Writes the value 0
packet.read("INT_ITEM_WITH_STATES") #=> returns "OFF"
```

## read\_all

Read all is used to retrieve all the packet values. The parameters are as follows:

| PARAMETER  | DESCRIPTION                                                                                                        |
|------------|--------------------------------------------------------------------------------------------------------------------|
| value_type | How the values should be read. Must be one of :RAW, :CONVERTED, :FORMATTED, or :WITH_UNITS. Defaults to :CONVERTED |
| buffer     | Raw data buffer to read the value from (defaults to the current packet)                                            |

Example:

```

values = packet.read_all() # Converted values (default)
values = packet.read_all(:RAW) # Raw values (no COSMOS conversions applied)
values = packet.read_all(:FORMATTED) # String formatted values
values = packet.read_all(:WITH_UNITS) # String formatted values with units

```

The return result is an array of arrays. Each internal array has two elements, the item name and value. The overall structure is as follows: [[item name, item value], [...], ...].

### **length**

Length returns the size of the internal packet buffer.

### **buffer**

Buffer provides access to the internal packet buffer. Note that by default it returns a copy of the internal buffer so any modifications do NOT affect the packet.

| PARAMETER | DESCRIPTION                                               |
|-----------|-----------------------------------------------------------|
| copy      | Whether to return a copy of the buffer. Defaults to true. |

### **buffer=**

Buffer= sets the internal packet buffer. Note that it copies the passed in buffer so any future modifications to the passed in buffer do NOT affect the packet.

| PARAMETER | DESCRIPTION                                                 |
|-----------|-------------------------------------------------------------|
| buffer    | String containing the raw binary buffer to back this packet |

### **clone and dup**

Clone and dup make a copy of the packet instance with a new internal buffer. Thus the newly cloned packet can modify its values independently of the previous packet.

### **identified?**

Returns whether this packet has been identified which means it has its internal target\_name and packet\_name set. Initially when data is read from an interface a nil packet is created like so:

`Packet.new(nil, nil, :BIG_ENDIAN, nil, data)`. The packet then needs to be identified by calling identify?.

### **identify?**

Returns whether the buffer of data parameter represents this packet. It does this by iterating over all the packet items that were created with an ID value (see [id\\_parameter](#) and [id\\_item](#)) and checking whether the ID values are present in the buffer.

| PARAMETER | DESCRIPTION                                                        |
|-----------|--------------------------------------------------------------------|
| buffer    | String containing the raw binary buffer to identify as this packet |

### **restore\_defaults**

Set all items in the packet to their default values. This only makes sense for command packets which define default values for parameters.

| PARAMETER | DESCRIPTION                                                                 |
|-----------|-----------------------------------------------------------------------------|
| buffer    | Raw data buffer to write default values to (defaults to the current packet) |

skip\_item\_names

Array of item names to skip when setting defaults

Definition File:

```
COMMAND TGT PKT BIG_ENDIAN "Packet"
APPEND_PARAMETER VALUE 32 UINT MIN MAX 10 "Value"
APPEND_PARAMETER OTHER 32 UINT MIN MAX 1 "Other"
```

Example:

```
command = System.commands.packet("TGT", "PKT")
command.read("VALUE") #=> Returns 0
command.restore_defaults
command.read("VALUE") #=> Returns 10
command.read("OTHER") #=> Returns 1
command.write("VALUE", 0)
command.write("OTHER", 0)
Since we want to use the skip_item_names parameter we have to pass in the buffer
Since we want to directly modify this packet buffer we use command.buffer(false)
because the default command.buffer returns a COPY of the internal buffer
command.restore_defaults(command.buffer(false), ['OTHER'])
command.read("VALUE") #=> Returns 10 (default)
command.read("OTHER") #=> Returns 0 (not default)
```

### out\_of\_limits

Out of limits returns an array of arrays indicating all the items in the packet that are out of limits. The internal array has four elements: target name, packet name, item name, and the item limits state. The out of limits item states are :RED\_HIGH, :YELLOW\_HIGH, :RED\_LOW, or :YELLOW\_LOW.

Example:

```
packet = System.telemetry.packet("TGT", "PKT")
packet.out_of_limits.each do |tgt, pkt, item, state|
 puts "tgt:#{tgt} pkt:#{pkt} item:#{item} state:#{state}"
end
```

Note that this code only works as expected inside the COSMOS Server, for example in a background task. If this code is executed in Script Runner you will retrieve a local copy of the packet which will not have any out of limits items.

BACK

NEXT

Site design derived from the [Jekyll](#) project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

## PacketItem Class

Improve this page

### This documentation is for COSMOS Developers

If you're simply trying to setup a COSMOS system you're probably looking for the [System Configuration](#) page. If you're trying to create a custom interface, background task, conversion, or build a custom tool then this is the right place.

The PacketItem class is used to access an individual item within a [Packet](#). The primary way to access Packet instances is through the [System](#) class. The PacketItem class provides access to information about the item like its location in the packet, type, endianness, conversions, states, etc.

The [packet\\_item.rb](#) source code on Github.

## Programming PacketItem

Custom COSMOS code may need to interact with PacketItem as it provides access to the individual item defined by your command and telemetry definitions. PacketItem inherits from StructureItem which it uses to provide some of the lower level functionality associated with defining an item. Typically you're directly using the PacketItem class.

PacketItem instances can be either command items or telemetry items and there is no way to directly know whether an instance is from a command or telemetry packet. However, since you're typically getting packet instances through

[System.commands](#) or [System.telemetry](#) this is rarely an issue in practice.

### PacketItem Instance Variables

PacketItem defines a large number of instance variables that provide information about the packet. The most commonly used are as follows:

1. `name` - Item name as a string
2. `description` - Description of the item
3. `bit_offset` - Where in the binary buffer the item exists
4. `bit_size` - The number of bites which represent the item in the buffer
5. `data_type` - Data type which can be :INT, :UINT, :FLOAT, :STRING, :BLOCK, or :DERIVED. See [Command](#) or [Telemetry](#) for a description of the types.
6. `endianness` - Endianness of the item which is either :BIG\_ENDIAN or :LITTLE\_ENDIAN.
7. `read_conversion` - Conversion applied when reading the item (typically applied to telemetry items)
8. `write_conversion` - Conversion applied when writing the item (typically applied to command items)
9. `states` - States used to convert a numeric value to a string
10. `units` - Abbreviated units of the item, e.g. "V"
11. `range` - Valid range of values (nil for :STRING or :BLOCK types, only applies to command items)

12. **id\_value** - Value used to identify a packet
13. **default** - Default value for the item (only applies to command items)
14. **limits** - Limits for the item (only applies to telemetry items)

## PacketItem Methods

The only methods most developers will use on PacketItem instances are the accessor methods that access the instance variables defined above. There are also corresponding setter methods that set all the above variables.

Command Example:

```
System.commands.each do |packet|
 packet.sorted_items.each do |item|
 puts "#{item.name}::#{item.description} range:#{item.range} default:#{item.default}"
 end
end
```

Telemetry Example:

```
System.telemetry.each do |packet|
 packet.sorted_items.each do |item|
 puts "#{item.name}::#{item.description} states:#{item.states} limits:#{item.limits}"
 end
end
```

Note that once you have a Packet instance you can access the items using the following methods:

### packet.get\_item

Returns an individual item by name

```
item = packet.get_item("ITEM_NAME")
```

### packet.items

Returns a hash of items keyed by the item name

```
packet.items.each do |name, item|
 puts "item:#{name}: #{item.description}"
end
```

### packet.sorted\_items

Returns an array of items sorted by the bit\_offset

```
packet.sorted_items.each do |item|
 puts "item:#{item.name}: #{item.bit_offset}"
end
```

## packet.id\_items

Returns an array of all the ID items defined in the packet. ID items are defined by the [ID\\_PARAMETER](#) keyword in commands and the [ID\\_ITEM](#) keyword in telemetry (and their associated APPEND keywords).

```
packet.id_items.each do |item|
 puts "item:#{@item.name}: #{@item.id_value}"
end
```

## packet.limits\_items

Returns an array of all the items defined in the packet with limits. Limits items are defined by the [LIMITS](#) keyword in telemetry items.

```
packet.limits_items.each do |item|
 puts "item:#{@item.name}: #{@item.limits}"
end
```

[!\[\]\(714d94d803c98f5e4e26bfab630acfb1\_img.jpg\) BACK](#) [\*\*NEXT\*\* !\[\]\(28bd6d6841112fcb26e7a6075f59192b\_img.jpg\)](#)

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by [\*\*GitHub\*\*](#)



Navigate the docs... ▾

# JSON API

Improve this page

## This documentation is for COSMOS Developers

If you're looking for the methods available to write test procedures using the COSMOS scripting API, refer to the [Scripting Guide](#) page. If you're trying to interface to a COSMOS Command and Telemetry Server from an external application using any language then this is the right place.

This document provides the information necessary for external applications to interact with the COSMOS Command and Telemetry Server using the COSMOS API. External applications written in any language can send commands and retrieve individual telemetry points using this API. External applications also have the option of connecting to the COSMOS Command and Telemetry server to interact with raw tcp/ip streams of commands/telemetry. However, the COSMOS JSON API removes the requirement that external applications have knowledge of the binary formats of packets.

## JSON-RPC 2.0

The COSMOS API implements a relaxed version of the [JSON-RPC 2.0 Specification](#). Requests with an “id” of NULL are not supported. Numbers can contain special non-string literal’s such as NaN, and +/-inf. Request params must be specified by-position, by-name is not supported. Section 6 of the spec, Batch Operations, is not supported.

## Socket Connections

The COSMOS Command and Telemetry Server listens for connections to the COSMOS API on an HTTP server (default port of 7777).

## Supported Methods

The list of methods supported by the COSMOS API may be found in the [api.rb](#) source code on Github. The @api\_whitelist variable is initialized with an array of all methods accepted by the CTS. This page will not show the full argument list for every method in the API, but it should be noted that the JSON API methods correspond to the COSMOS scripting API methods documented in the [Scripting Guide](#). This page will show a few example JSON requests and responses, and the scripting guide can be used as a reference to extrapolate how to build requests and parse responses for methods not explicitly documented here.

## Existing Implementations

The COSMOS JSON API has been implemented in the following languages:

- [Python](#)

## Example Usage

## Sending Commands

The following methods are used to send commands: cmd, cmd\_no\_range\_check, cmd\_no\_hazardous\_check, cmd\_no\_checks

The cmd method sends a command to a COSMOS target in the system. The cmd\_no\_range\_check method does the same but ignores parameter range errors. The cmd\_no\_hazardous\_check method does the same, but allows hazardous commands to be sent. The cmd\_no\_checks method does the same but allows hazardous commands to be sent, and ignores range errors.

Two parameter syntaxes are supported.

The first is a single string of the form “TARGET\_NAME COMMAND\_NAME with PARAMETER\_NAME\_1 PARAMETER\_VALUE\_1, PARAMETER\_NAME\_2 PARAMETER\_VALUE\_2,...” The “with ...” portion of the string is optional. Any unspecified parameters will be given default values.

| PARAMETER      | DATA TYPE | DESCRIPTION                                                         |
|----------------|-----------|---------------------------------------------------------------------|
| command_string | string    | A single string containing all required information for the command |

The second is two or three parameters with the first parameter being a string denoting the target name, the second being a string with the command name, and an optional third being a hash of parameter names/values. This format should be used if the command contains parameters that take binary data that is not capable of being expressed as ASCII text. The cmd and cmd\_no\_range\_check methods will fail on all attempts to send a command that has been marked hazardous. To send hazardous commands, the cmd\_no\_hazardous\_check, or cmd\_no\_checks methods must be used.

| PARAMETER      | DATA TYPE | DESCRIPTION                               |
|----------------|-----------|-------------------------------------------|
| target_name    | String    | Name of the target to send the command to |
| command_name   | String    | The name of the command                   |
| command_params | Hash      | Optional hash of command parameters       |

Example Usage:

```
--> {"jsonrpc": "2.0", "method": "cmd", "params": ["INST COLLECT with DURATION 1.0, TEMP 0.0, TYPE 'I'"]
<-- {"jsonrpc": "2.0", "result": ["INST", "COLLECT", {"DURATION": 1.0, "TEMP": 0.0, "TYPE": "NORMAL"}]}
```

```
--> {"jsonrpc": "2.0", "method": "cmd", "params": ["INST", "COLLECT", {"DURATION": 1.0, "TEMP": 0.0, "TYPE": "I"}]
<-- {"jsonrpc": "2.0", "result": ["INST", "COLLECT", {"DURATION": 1.0, "TEMP": 0.0, "TYPE": "NORMAL"}]}
```

## Getting Telemetry

The following methods are used to get telemetry: tlm, tlm\_raw, tlm\_formatted, tlm\_with\_units

The tlm method returns the current converted value of a telemetry point. The tlm\_raw method returns the current raw value of a telemetry point. The tlm\_formatted method returns the current formatted value of a telemetry point. The tlm\_with\_units method returns the current formatted value of a telemetry point with its units appended to the end.

Two parameter syntaxes are supported.

The first is a single string of the form “TARGET\_NAME PACKET\_NAME ITEM\_NAME”

| PARAMETER | DATA TYPE | DESCRIPTION |
|-----------|-----------|-------------|
|           |           |             |

|            |        |                                                                            |
|------------|--------|----------------------------------------------------------------------------|
| tlm_string | String | A single string containing all required information for the telemetry item |
|------------|--------|----------------------------------------------------------------------------|

The second is three parameters with the first parameter being a string denoting the target name, the second being a string with the packet name, and the third being a string with the item name.

| PARAMETER   | DATA TYPE | DESCRIPTION                                        |
|-------------|-----------|----------------------------------------------------|
| target_name | String    | Name of the target to get the telemetry value from |
| packet_name | String    | Name of the packet to get the telemetry value from |
| item_name   | String    | Name of the telemetry item                         |

Example Usage:

```
--> {"jsonrpc": "2.0", "method": "tlm", "params": ["INST HEALTH_STATUS TEMP1"], "id": 2}
<-- {"jsonrpc": "2.0", "result": 94.9438, "id": 2}

--> {"jsonrpc": "2.0", "method": "tlm", "params": ["INST", "HEALTH_STATUS", "TEMP1"], "id": 2}
<-- {"jsonrpc": "2.0", "result": 94.9438, "id": 2}
```

 BACK  NEXT 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the [Jekyll](#) project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

Improve this page

- [Binary File Structure](#)

## Logging

This document describes the COSMOS logging framework and structure.

The COSMOS system creates a number of different logs both for data and for application events. The log file location is defined in the system.txt file by the '[PATH LOGS](#)' setting. The log files which get created in this directory follow a common naming convention. The first part is a date and time: YYYY\_MM\_DD\_HH\_MM\_SS\_. This is followed by words specific to the COSMOS application. COSMOS applications produce the following logs:

| APPLICATION    | LOG FILE NAME                     | DESCRIPTION                                                                                                                                                                                                                                                                                           |
|----------------|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Any            | <date>_exception.txt              | Exception file which is generated if a COSMOS application crashes. Any user provided interface code, background tasks, conversions, etc can result in an exception file if not properly written.                                                                                                      |
| Any            | <date>_unexpected.txt             | Unexpected text output. If user code tries to print to the standard output using 'puts', this output is captured and an unexpected log file is created when the application exits.                                                                                                                    |
| Cmd/Tlm Server | <date>_cmd.bin                    | All outgoing command packets                                                                                                                                                                                                                                                                          |
| Cmd/Tlm Server | <date>_tlm.bin                    | All incoming telemetry packets                                                                                                                                                                                                                                                                        |
| Cmd/Tlm Server | <date>_server_messages.txt        | All the server messages which are found in the bottom half of the Command and Telemetry Server application. Server messages are time stamped and categorized as INFO, WARN, or ERROR. Messages include red, yellow, limit violations, limit groups enable/disable, commands, and log file open/close. |
| Command Sender | <date>_cmdsender_messages.txt     | Commands sent through Command Sender are logged and timestamped.                                                                                                                                                                                                                                      |
| Script Runner  | <date>_sr_<filename>_messages.txt | Each time a script is run, a new messages log file is created. This log captures any 'puts' output in the script as well as any user input. It also captures user interaction with Script Runner such as clicking the Pause and Stop buttons (as well as Step when debugging).                        |

|             |                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|             |                               | Each time a test is run, a new messages log file is created. The <test> part of the filename is either the Suite name if a Suite is run, Suite_TestGroup if a Test Group is run, or Suite_TestGroup_TestCase if a test case is run. This log captures any 'puts' output in the script as well as any user input. It also captures user interaction with Script Runner such as clicking the Pause and Stop buttons (as well as Step when debugging). |
| Test Runner | <date>_sr_<test>_messages.txt | Test Runner test report which indicates the file run, metadata, Test Runner settings (checkboxes), Test Case Results and a summary of the tests run.                                                                                                                                                                                                                                                                                                |
| Test Runner | <date>_testrunner_results.zip | If the Test Runner <a href="#">CREATE_DATA_PACKAGE</a> configuration option is set, a zip file is created containing the the test report, the server messages and the command and telemetry bin files.                                                                                                                                                                                                                                              |

## Binary File Structure

The Command and Telemetry Server binary log files contain command and telemetry packets. Each log file starts with a 128 byte header that is used internally by COSMOS to denote the version of definition files that were used when the file was created.

| Item       | Data Type            | Description                                                        |
|------------|----------------------|--------------------------------------------------------------------|
| Marker     | 8 byte ASCII String  | The ASCII string 'COSMOS2_'                                        |
| Type       | 4 byte ASCII String  | Either 'CMD_' for command logs or 'TLM_' for Telemetry logs        |
| MD5        | 32 byte ASCII String | 32 byte MD5 Sum calculated over the configuration files being used |
| Underscore | 1 byte ASCII String  | An Underscore character                                            |
| Hostname   | 83 byte ASCII String | Left justified hostname of the computer that created the file      |

The file header is followed by command or telemetry packets (depending on the log type), each with a Big Endian header as follows. Note this is the same binary header used by the preidentified streaming protocol:

| Item                    | Data Type                          | Description                                                                                                                                                               |
|-------------------------|------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Write Flags             | 8-bit unsigned integer             | 0x80 indicates whether the data is stored telemetry (see below). 0x40 indicates there is extra data following this byte. 0x3F bits are currently undefined. (since 4.3.0) |
| Extra Length (optional) | 32-bit unsigned big endian integer | Number of extra bytes (Optional, depends on Write Flags) (since 4.3.0)                                                                                                    |
| Extra (optional)        | 32-bit unsigned big endian integer | JSON encoded extra data (Optional, depends on Write Flags) (since 4.3.0)                                                                                                  |
| Time Seconds            | 32-bit unsigned big endian integer | Seconds since Unix epoch (Jan 1st, 1970 – Midnight)                                                                                                                       |
| Time Microseconds       | 32-bit unsigned big endian integer | Microseconds of Second since Unix epoch (Jan 1st, 1970 – Midnight)                                                                                                        |

|                    |                                    |                                                                              |
|--------------------|------------------------------------|------------------------------------------------------------------------------|
| Target Name Length | 8-bit unsigned integer             | Length in bytes of the target name                                           |
| Target Name        | Variable length ASCII String       | String that indicates the name of the target that the data was sourced from. |
| Packet Name Length | 8-bit unsigned integer             | Length in bytes of the packet name                                           |
| Packet Name        | Variable length ASCII String       | String that indicates the name of the packet of data.                        |
| Packet Length      | 32-bit unsigned big endian integer | Length in bytes of the packet                                                |
| Packet             | Variable length block of data      | The binary data packet (endianness defined by packet definition)             |

The variable length nature of command and telemetry packets requires a log parser to start from the beginning of each log file when processing packets.

If the Write Flags indicate the data is stored telemetry (MSB set) COSMOS processes and stores the data in the telemetry log file but does not update the current value table. Thus stored telemetry does not affect displays or scripts.

 **BACK** **NEXT** 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

✍ Improve this page

- [Terminology](#)
- [Overall Architecture and Context Diagram](#)
- [Overall Requirements](#)
- [Command and Telemetry Server](#)
- [Replay](#)
- [Command Sender](#)
- [Script Runner](#)
- [Test Runner](#)
- [Packet Viewer](#)
- [Telemetry Viewer](#)
- [Telemetry Grapher](#)
- [Data Viewer](#)
- [Limits Monitor](#)
- [Telemetry Extractor](#)
- [Command Extractor](#)
- [Table Manager](#)
- [Handbook Creator](#)
- [Launcher](#)

## Requirements and Design

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.

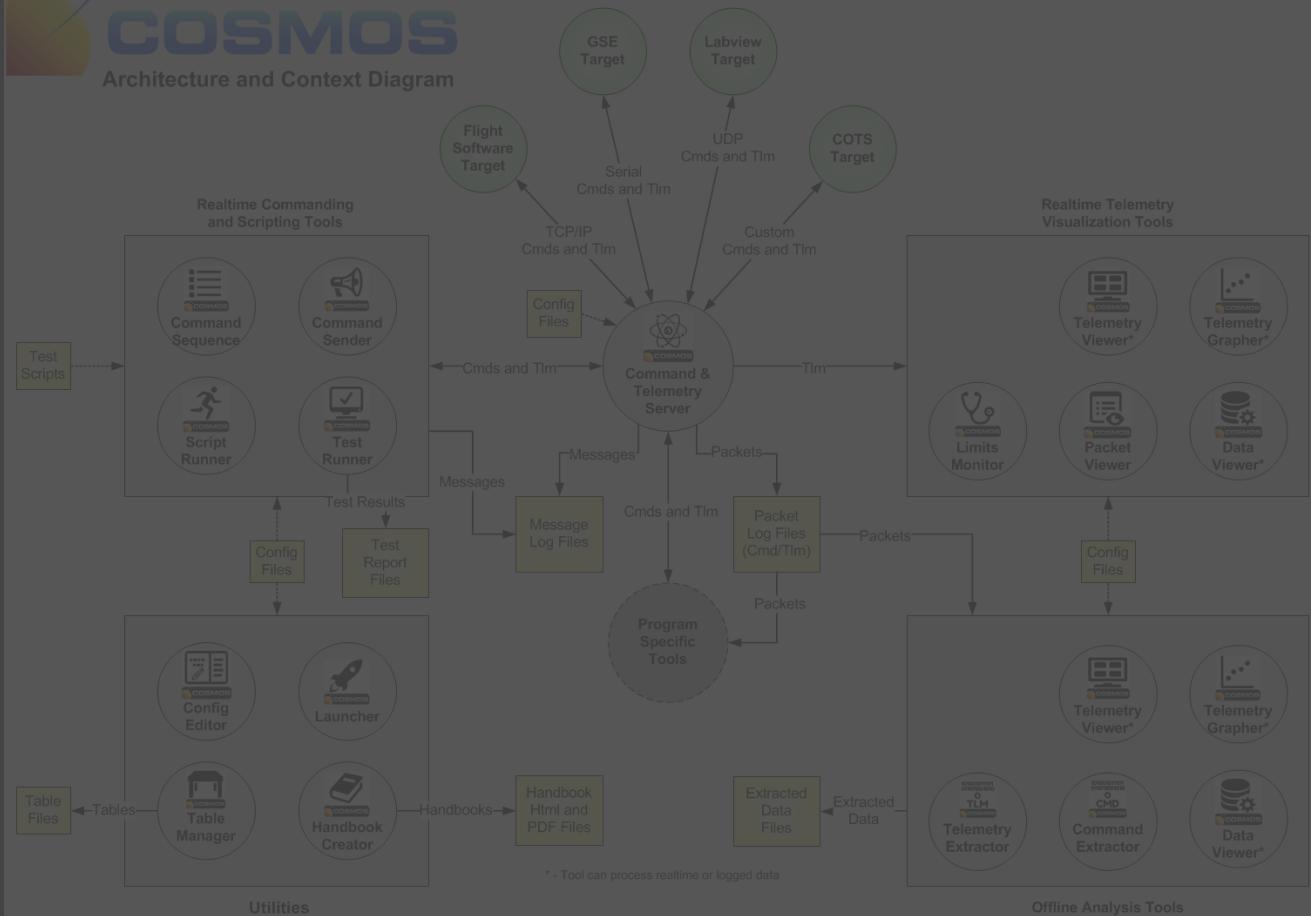
## Terminology

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 to 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    | Deleted                                                                         | Deleted                                          | Deleted    |
| 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() COSMOSAPI 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 occurrence 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 succeed. 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 starting a script from any line.                                                                              | Place the mouse cursor at the desired first line and 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 user interaction after each line.                         | Press Step 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 if 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 |

|       |                                                                                                                                               |                                                                                                                                     |                   |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|-------------------|
|       |                                                                                                                                               |                                                                                                                                     |                   |
| PV-5  | Packet Viewer shall provide a mechanism 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 point 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 outputting 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 |

 BACK  NEXT 

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the [Jekyll](#) project documentation which is licensed under the terms of the MIT License.

Proudly hosted by [GitHub](#)



Navigate the docs... ▾

## XTCE Support

Improve this page

Ball Aerospace COSMOS now has support for the [XTCE Command and Telemetry Definition Standard](#). This is an open standard designed to allow command and telemetry definitions to be transferred between different ground systems. COSMOS can run directly using the .xtce files, or can convert them into the COSMOS configuration file format.

### Running COSMOS using an .xtce definition file

A single .xtce file containing the command and telemetry definitions for a target can be used in place of the normal COSMOS command and telemetry definition files. Simply place the target's .xtce file in the target's cmd\_tlm folder and COSMOS will use it for the command and telemetry definitions.

### Converting a .xtce file into a COSMOS configuration

Use the following command to convert a .xtce file into COSMOS configuration files. The converted configuration files will be placed into a target folder in the given output directory.

```
xtce_converter --import <xtce_filename> <output_dir>
```

### Converting a COSMOS Configuration to XTCE

Use the following command to convert your current cosmos project into .xtce files, one per target. The converted .xtce files will be placed into a target folder in the given output directory.

```
xtce_converter --export <output_dir>
```

### High-level Overview of Current Support

1. Integer, Float, Enumerated, String, and Binary Parameter/Argument Types are Supported
2. All DataEncodings are supported
3. Telemetry and Commands are Supported
4. Packet Identification is supported
5. States are supported
6. Units are supported
7. PolynomialCalibrators are supported
8. Only one SpaceSystem per .xtce file

9. Packets should not have gaps between items

## Supported Elements and Attributes

The following elements and associated attributes are currently supported.

- SpaceSystem
- TelemetryMetaData
- CommandMetaData
- ParameterTypeSet
- EnumerationList
- ParameterSet
- ContainerSet
- EntryList
- DefaultCalibrator
- DefaultAlarm
- RestrictionCriteria
- ComparisonList
- MetaCommandSet
- DefaultCalibrator
- ArgumentTypeSet
- ArgumentList
- ArgumentAssignmentList
- EnumeratedParameterType
- EnumeratedArgumentType
- IntegerParameterType
- IntegerArgumentType
- FloatParameterType
- FloatArgumentType
- StringParameterType
- StringArgumentType
- BinaryParameterType
- BinaryArgumentType
- IntegerDataEncoding
- FloatDataEncoding
- StringDataEncoding
- BinaryDataEncoding'
- SizelnBits
- FixedValue
- UnitSet
- Unit
- PolynomialCalibrator

- Term
- StaticAlarmRanges
- WarningRange
- CriticalRange
- ValidRange
- Enumeration
- Parameter
- Argument
- ParameterProperties
- SequenceContainer
- BaseContainer
- LongDescription
- ParameterRefEntry
- ArgumentRefEntry
- BaseMetaCommand
- Comparison
- MetaCommand
- BaseMetaCommand
- CommandContainer
- ArgumentAssignment

## Ignored Elements

The following elements are simply ignored by COSMOS:

- Header
- AliasSet
- Alias

## Unsupported Elements

Any elements not listed above are currently unsupported. Near term support for the following elements and features are planned and priority will be determined by user requests.

- SplineCalibrator
- Alternate methods of specifying offsets into containers
- Output to the XUSP standard
- Additional Data Types
- Container References

If there is a particular element or feature you need supported please submit a ticket on Github.

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

## Contributing

Improve this page

So you've got an awesome idea to throw into COSMOS. Great! This 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 ▶

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

## Contact Us

Improve this page

Please reach out to us with any questions at [cosmos@ball.com](mailto:cosmos@ball.com)

### Business Development and Support Contracts

Mike Lammertin  
Phil Inslee

### Technical Leads

Ryan Melton  
Jason Thomas  
Donald Hall

BACK

NEXT

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by



Navigate the docs... ▾

## Release History

Improve this page

### 4.3.0 / 2018-08-30

Welcome to COSMOS 4.3.0!

The highlight of this release is built in support for differentiating between stored telemetry and realtime telemetry. If your system downlinks stored telemetry that you don't want to interfere with the COSMOS realtime current value table, your interface/protocol code can set the stored flag on a packet before returning it to COSMOS to have COSMOS log the packet, but not update the current telemetry values.

Lots of other new features as well, and a few bug fixes including fixing running on the latest version of Mac OSX. See below for the full list.

**Breaking Changes:** The COSMOS log file format and pre-defined protocol formats have changed to support stored telemetry. COSMOS 4.3 is backwards compatible with earlier formats, but older versions of COSMOS won't be able to read files from COSMOS 4.3+.

#### New Features:

- [#780](#) Search bar support for Command Sequence
- [#785](#) Adjust font size
- [#804](#) Add classification bar to all tools
- [#808](#) Add CANVASELLIPSE widget
- [#814](#) Changes to support stored telemetry
- [#815](#) Create a temporary screen from a script
- [#818](#) Catch all state for any undefined state values
- [#819](#) Ensure x values added in incrementing order in TlmGrapher
- [#823](#) Add protocol to ignore any packet
- [#826](#) Allow hash style access to Excel
- [#829](#) Add STAY\_ON\_TOP feature to telemetry screens
- [#851](#) Ensure step\_mode displays Step button
- [#859](#) Change Script Runner highlight color when paused

#### Maintenance:

- [#796](#) Handle Command Sequence Export Issues
- [#807](#) Revert Items with STATE cannot define FORMAT\_STRING
- [#812](#) Ruby 2.5 Instrumented Code outside of methods in a class
- [#855](#) Remove Qt warning message on Windows 10

- #858 Fix JRuby build issues

### Bug Fixes:

- #837 Packet Viewer's "Hide Ignored Items" Incorrectly Persists Across Change of Packet
- #840 Latest version of COSMOS crashing
- #845 Support get\_tlm\_details in disconnect mode

### Migration Notes from COSMOS 4.2.x:

To upgrade to the latest version of COSMOS, run "bundle update cosmos" in your COSMOS project folder.

See the COSMOS documentation for directions on setting up DART: <http://cosmosrb.com/docs/home/>

## 4.2.4 / 2018-05-16

This is the second patch release for 4.2. It greatly improves the ingest speed for DART (100x), improves decom speed, reduces database size, and fixes some bugs. If you are using DART, please upgrade and follow the migration directions at the end of these release notes.

The highlight of COSMOS 4.2 is a new tool called the Data Archival and Retrieval Tool (DART). DART is a long term trending database built on top of the PostgreSQL database. It integrates directly with TlmGrapher, TlmExtractor, CmdExtractor, DataViewer, and Replay, allowing you to do historical queries of logged telemetry (and commands) by specifying a time range. Queries are super fast and it performs automatic data reduction at minute/hour/day granularity. Consider setting it up for your project and start data mining today!

See the COSMOS documentation for directions on setting up DART: <http://cosmosrb.com/docs/home/>

### New Features:

- #787 No way to control data bits on serial interface
- #788 ROUTERS should support PROTOCOL keyword

### Maintenance:

- #784 Comparable and spaceship operator behavior changing
- #791 Table Manager doesn't expose top level layout

### Bug Fixes:

- #779 Dart updates for ingest speed, correct time zone, TlmGrapher crash
- #786 Status tab crash on Ruby 2.5
- #790 Telemetry check doesn't support strings with multiple spaces

### Migration Notes from COSMOS 4.1.x:

To upgrade to the latest version of COSMOS, run "bundle update cosmos" in your COSMOS project folder.

If you already setup DART for your program please follow the following additional steps: In a terminal in your COSMOS project folder run:

```
rake db:migrate
rake db:seed
```

See the COSMOS documentation for directions on setting up DART: <http://cosmosrb.com/docs/home/>

## 4.2.3 / 2018-04-17

COSMOS 4.2 is here! This is the first true patch release for 4.2. The highlight of COSMOS 4.2 is a new tool called the Data Archival and Retrieval Tool (DART). DART is a long term trending database built on top of the PostgreSQL database. It integrates directly with TlmGrapher, TlmExtractor, CmdExtractor, DataViewer, and Replay, allowing you to do historical queries of logged telemetry (and commands) by specifying a time range. Queries are super fast and it performs automatic data reduction at minute/hour/day granularity. Consider setting it up for your project and start data mining today!

See the COSMOS documentation for directions on setting up DART: <http://cosmosrb.com/docs/home/>

### New Features:

- #767 Support Ruby 2.5
- #771 Add CmdSender Search

### Maintenance:

- #772 OpenGL gem isn't supported in Ruby 2.5

### Bug Fixes:

- #769 TIMEGRAPH widget non-functional
- #775 Toggle disconnect broken in TestRunner

### Migration Notes from COSMOS 4.1.x:

To upgrade to the latest version of COSMOS, run “bundle update cosmos” in your COSMOS project folder.

See the COSMOS documentation for directions on setting up DART: <http://cosmosrb.com/docs/home/>

## 4.2.2 / 2018-04-11

COSMOS 4.2 is here! Thirty four tickets went into this release, but the highlight is a new tool called the Data Archival and Retrieval Tool (DART). DART is a long term trending database built on top of the PostgreSQL database. It integrates directly with TlmGrapher, TlmExtractor, CmdExtractor, DataViewer, and Replay, allowing you to do historical queries of logged telemetry (and commands) by specifying a time range. Queries are super fast and it performs automatic data reduction at minute/hour/day granularity. Consider setting it up for your project and start data mining today!

See the COSMOS documentation for directions on setting up DART: <http://cosmosrb.com/docs/home/>

### New Features:

- #698 Initial DART Release
- #650 Gracefully handle large array items
- #673 Button widget should spawn thread to avoid blocking GUI
- #676 Allow individual interfaces to be disconnect mode
- #699 Test cases added to TestRunner should be ordered in drop down
- #705 Cmd line arg for ScriptRunner to start in disconnect mode
- #706 Warn if ITEMS or PARAMETERS are redefined
- #711 Allow ERB to know the target name
- #715 Allow individual Limits Monitor items to be removed (not ignored)
- #719 Warn if limits\_group doesn't exist in limits\_groups\_background\_task

- #729 CmdSender production mode to disable MANUALLY ENTERED
- #734 Support DERIVED with APPEND
- #737 Implement single stepping with the F10 key
- #754 Add Replay Mode to Include All Routers
- #765 TlmGrapher sampled analysis

### Maintenance:

- #682 Fix Ruby interpreter warnings
- #687 Add ConfigEditor AHK tests
- #688 Windows 10 Installation Error - RDoc parsing failure in qtruby4.rb
- #692 Fix METADATA usage in demo
- #738 PacketViewer scroll to item on search
- #748 Syntax highlighting prioritizes string over comment
- #750 TestRunner hides syntax errors with broad rescue
- #752 Demo INST commanding screen broken
- #757 Increase TlmGrapher timeout to better support Replay
- #759 Allow underscores and dashes in log filename labels

### Bug Fixes:

- #690 Automatic SYSTEM META definition doesn't include RECEIVED\_XX
- #691 tools/mac apps won't open
- #701 XTCE String types should not have ByteOrderList
- #709 Can't set breakpoint in subscript
- #713 Launcher crashes if newline in crc.txt
- #723 crc\_protocol needs better input validation
- #727 Install issue on Windows 10
- #732 losing/gaining data when routing at different incoming rates
- #735 Statistics Processor doesn't handle nil or infinite
- #740 About dialog crashes if USER\_VERSION not defined

### Migration Notes from COSMOS 4.1.x:

To upgrade to the latest version of COSMOS, run “bundle update cosmos” in your COSMOS project folder.

See the COSMOS documentation for directions on setting up DART: <http://cosmosrb.com/docs/home/>

## 4.1.1 / 2017-12-07

### New Features:

- #663 Built-in protocols to support allow\_empty\_data
- #666 Add ability to create target in ConfigEditor
- #679 TlmViewer screen audit shouldn't count reserved item names

### Maintenance:

- #660 Update Opengl gem requirement version

- #665 Refactor xtce parser

### Bug Fixes:

- #661 Render function bug?
- #674 Add TlmViewerConfig spec and fix to\_save

### Migration Notes from COSMOS 4.0.x:

Any custom tools in other languages that use the COSMOS API will need to be updated.

To upgrade to the latest version of COSMOS, run “bundle update cosmos” in your COSMOS project folder.

## 4.1.0 / 2017-11-17

Welcome to COSMOS 4.1! The COSMOS API has transitioned from a custom TCP protocol to HTTP. This will make interfacing to COSMOS from programming languages other than Ruby much easier. There are also many new APIs that allows the full functionality of the CmdTlmServer and Replay to be controlled remotely. See below for the full list of changes!

### New Features:

- #531 Command sequence export option
- #558 subscription APIs should work in disconnect mode
- #559 Build Replay Functionality into CmdTlmServer
- #578 Add ability to query available screens from Telemetry Viewer
- #579 Create zip of configuration in saved config
- #581 Make various log files readonly
- #587 AUTO\_INTERFACE\_TARGETS ignore existing interfaces
- #591 Investigate creating single COSMOS UDP socket for case where read\_port == write\_src\_port
- #599 Create subscription API for CTS messages
- #601 Add background task and other status APIs
- #606 Enhancement to Open File Script Interface - File Filter
- #612 CSV support for strings and symbols
- #620 Move Script Runner step button next to start/go
- #625 Add APIs to start/stop background tasks
- #626 Add API functions to get ignored parameters/items
- #635 get\_cmd\_param\_list should return the type for each command parameter
- #642 Handle Infinity/NaN without invalid JSON

### Maintenance:

- #510 Investigate changing JsonDrb to running over HTTP
- #603 CmdTlmServer invalid constructor argument
- #614 Pass server message color to message subscriptions
- #619 Script Runner instrumenting comments and whitespace
- #630 Add ConfigEditor and CmdSequence to install launcher
- #647 Make packet item sort consistent
- #649 Add codecov support

## Bug Fixes:

- #562 #562 Template protocol should fill Id fields
- #577 #577 Telemetry Viewer has fatal exception when called with JSON-RPC method display
- #593 #593 Race condition in Cosmos.kill\_thread
- #607 #607 Support latest version of wkhtmltopdf for pdf creation and properly set working dir
- #610 #610 target shouldn't report error requiring file in target lib
- #616 #616 Ignore untested Windows version message in Windows 10
- #617 #617 Ruby 2.4's inherent Warning class shadows Qt::MessageBox::Warning
- #633 #633 combo\_box is mutating input
- #639 #639 Partial rendering in config parser should enforce that rendered partials start with underscore
- #654 #654 Test Runner crashes with no config file
- #655 #655 Metadata system triggers a nil router error in api.rb with basic setup
- #659 #659 Hazardous commands throwing errors

## Migration Notes from COSMOS 4.0.x:

Any custom tools in other languages that use the COSMOS API will need to be updated.

To upgrade to the latest version of COSMOS, run “bundle update cosmos” in your COSMOS project folder.

## 4.0.3 / 2017-10-24

**Important Bug Fix:** UdpInterface was only working for localhost on earlier versions of COSMOS 4.0.x. Please upgrade to COSMOS 4.0.3 if you need support for UDP.

## New Features:

- #585 Add packet level config\_name

## Maintenance:

None

## Bug Fixes:

- #590 UdpReadSocket must be created before UdpWriteSocket if read\_port == write\_src\_port

## Migration Notes from COSMOS 3.x:

COSMOS 4 includes several breaking changes from the COSMOS 3.x series.

The first and simplest is that the Command and Telemetry Server now opens an additional port at 7780 by default, that provides a router that will send out each command that the system has sent. This can allow external systems to also log all commands sent by COSMOS. For most people this change will be transparent and no updates to your COSMOS configuration will be required.

The second is that the Command and Telemetry Server now always supports a meta data packet called SYSTEM META. This packet will always contain the MD5 sum for the current running COSMOS configuration, the version of COSMOS running, the version of your COSMOS Project, and the version of Ruby being used. You can also add your own requirements for meta data with things like the name of the operator currently running the system, or the name of a specific test you are currently running. In general you shouldn't need to do anything for this change unless you were using the previous metadata functionality in COSMOS. If you were, then you will need to migrate your meta data to the new SYSTEM META packet, and change the parameters in your CmdTlmServer or TestRunner configurations regarding

meta data. If you weren't using metadata before, then you will probably just notice this new packet in your log files, and in your telemetry stream.

Finally the most exciting breaking change is in how COSMOS interfaces handle protocols. Before, the COSMOS TCP/IP and Serial interface classes each took a protocol like LENGTH, TERMINATED, etc that defined how packets were delineated by the interface. Now each interface can take a set of one or more protocols. This allows COSMOS to much more easily support nested protocols, such as the frame focused protocols of CCSDS. It also allows for creating helpful reusable protocols such as the new CRC protocol for automatically adding CRCs to outgoing commands and verifying incoming CRCs on telemetry packets. It's a great change, but if you have any custom interface classes you have written, they will probably require some modification. See the Interfaces section at [cosmosrb.com](http://cosmosrb.com) to see how the new interface classes work. We will also be writing up a blog post to help document the process of upgrading. Look for this in a week or two.

To upgrade to the latest version of COSMOS, run "bundle update cosmos" in your COSMOS project folder.

## 4.0.2 / 2017-09-29

**Important Bug Fix:** UdpInterface was only working for localhost on earlier versions of COSMOS 4.0.x. Please upgrade to COSMOS 4.0.2 if you need support for UDP.

### New Features:

- [#577](#) LIMITSBAR widget shouldn't allow RAVV
- [#565](#) Template Protocol support for logging rather than disconnect on timeout

### Maintenance:

- [#551](#) TImViewer meta AUTO\_TARGET needs parameter
- [#553](#) Create cosmosrb documentation based on metadata
- [#554](#) TEMP1 limits getting disabled in demo is confusing

### Bug Fixes:

- [#564](#) Items with STATES don't respect LIMITS
- [#568](#) CSV shouldn't call compact
- [#569](#) combo\_box and vertical\_message\_box don't report correct user selection
- [#580](#) Udp interface does not work for non-localhost

### Migration Notes from COSMOS 3.8.x:

COSMOS 4 includes several breaking changes from the COSMOS 3.x series.

The first and simplest is that the Command and Telemetry Server now opens an additional port at 7780 by default, that provides a router that will send out each command that the system has sent. This can allow external systems to also log all commands sent by COSMOS. For most people this change will be transparent and no updates to your COSMOS configuration will be required.

The second is that the Command and Telemetry Server now always supports a meta data packet called SYSTEM META. This packet will always contain the MD5 sum for the current running COSMOS configuration, the version of COSMOS running, the version of your COSMOS Project, and the version of Ruby being used. You can also add your own requirements for meta data with things like the name of the operator currently running the system, or the name of a specific test you are currently running. In general you shouldn't need to do anything for this change unless you were using the previous metadata functionality in COSMOS. If you were, then you will need to migrate your meta data to the new SYSTEM META packet, and change the parameters in your CmdTImServer or TestRunner configurations regarding meta data. If you weren't using metadata before, then you will probably just notice this new packet in your log files, and

in your telemetry stream.

Finally the most exciting breaking change is in how COSMOS interfaces handle protocols. Before, the COSMOS TCP/IP and Serial interface classes each took a protocol like LENGTH, TERMINATED, etc that defined how packets were delineated by the interface. Now each interface can take a set of one or more protocols. This allows COSMOS to much more easily support nested protocols, such as the frame focused protocols of CCSDS. It also allows for creating helpful reusable protocols such as the new CRC protocol for automatically adding CRCs to outgoing commands and verifying incoming CRCs on telemetry packets. It's a great change, but if you have any custom interface classes you have written, they will probably require some modification. See the Interfaces section at [cosmosrb.com](http://cosmosrb.com) to see how the new interface classes work. We will also be writing up a blog post to help document the process of upgrading. Look for this in a week or two.

To upgrade to the latest version of COSMOS, run "bundle update cosmos" in your COSMOS project folder.

## 4.0.1 / 2017-08-23

### New Features:

- [#527](#) Editing config files should now bring up ConfigEditor
- [#528](#) ConfigEditor missing some keywords
- [#534](#) Create ConfigEditor Mac app
- [#536](#) Clickable canvas objects open screens
- [#542](#) Automatically populate COMMAND SYSTEM META
- [#543](#) Allow SYSTEM META items to be read only

### Maintenance:

- None

### Bug Fixes:

- [#533](#) TestRunner strips all comments when running
- [#538](#) META\_INIT broken
- [#540](#) Background task packet subscription get\_packet broken
- [#547](#) convert\_packet\_to\_data should copy buffer

### Migration Notes from COSMOS 3.8.x:

COSMOS 4 includes several breaking changes from the COSMOS 3.x series.

The first and simplest is that the Command and Telemetry Server now opens an additional port at 7780 by default, that provides a router that will send out each command that the system has sent. This can allow external systems to also log all commands sent by COSMOS. For most people this change will be transparent and no updates to your COSMOS configuration will be required.

The second is that the Command and Telemetry Server now always supports a meta data packet called SYSTEM META. This packet will always contain the MD5 sum for the current running COSMOS configuration, the version of COSMOS running, the version of your COSMOS Project, and the version of Ruby being used. You can also add your own requirements for meta data with things like the name of the operator currently running the system, or the name of a specific test you are currently running. In general you shouldn't need to do anything for this change unless you were using the previous metadata functionality in COSMOS. If you were, then you will need to migrate your meta data to the new SYSTEM META packet, and change the parameters in your CmdTlmServer or TestRunner configurations regarding meta data. If you weren't using metadata before, then you will probably just notice this new packet in your log files, and in your telemetry stream.

Finally the most exciting breaking change is in how COSMOS interfaces handle protocols. Before, the COSMOS TCP/IP and Serial interface classes each took a protocol like LENGTH, TERMINATED, etc that defined how packets were delineated by the interface. Now each interface can take a set of one or more protocols. This allows COSMOS to much more easily support nested protocols, such as the frame focused protocols of CCSDS. It also allows for creating helpful reusable protocols such as the new CRC protocol for automatically adding CRCs to outgoing commands and verifying incoming CRCs on telemetry packets. It's a great change, but if you have any custom interface classes you have written, they will probably require some modification. See the Interfaces section at [cosmosrb.com](http://cosmosrb.com) to see how the new interface classes work. We will also be writing up a blog post to help document the process of upgrading. Look for this in a week or two.

To upgrade to the latest version of COSMOS, run “bundle update cosmos” in your COSMOS project folder.

## 4.0.0 / 2017-08-04

COSMOS 4 is here!

48 tickets have gone into this release, and it brings with it two new tools and some great under the hood improvements.

New Tools:

COSMOS now has a dedicated Configuration Editor and Command Sequence Builder.

The config editor gives you contextual help when building config files, and make it super easy to define packets and configure tools without having to have the online documentation up in front of you. It's going to make setting up COSMOS even easier than it was before.

Command Sequence builder allows you to define series of commands that should be sent at either absolute or relative timestamps to each other. This is great for planning time specific commanding. You can execute these on the ground directly from the tool, or you can convert them to your own internal format and upload to the system you are commanding.

Highlighted changes:

1. New protocol system allows assigning multiple protocols to each interface to support layered protocols, and common functionality like CRC checking/adding to commands.
2. The ability to View the most recent raw data received or sent on each interface
3. The Command and Telemetry Server can now run on JRuby in –no-gui mode (may help performance for huge projects with 50+ targets)
4. New router provides the ability to get a copy of every command sent out from COSMOS in a stream
5. New SYSTEM META packet output by the CmdTlmServer
6. Lots more! See the full ticket list below

### New Features:

- #229 Gem Based Targets should support DataViewer and other tool configurations
- #234 Add a method to system.txt to add files used in the marshall file MD5 sum calculation
- #253 Create “generators” for targets, tools, etc
- #258 Create COSMOS Command Sequence Tool
- #261 Provide a method for specifying binary data in STRING and BLOCK default values
- #278 Consider adding wait\_methods to internal API for use in Background tasks
- #281 Add support for stretch and spacers in widget layouts
- #319 Add the ability to grab telemetry ARRAY\_ITEMS
- #337 Support specifying default parameters to default log reader and log writer in system.txt

- #347 COSMOS GLobal Time Zone Setting (Local/UTC)
- #356 Interface protocols
- #360 Add raw stream preamble and postamble data to “View Raw”
- #381 Float Infinity and NaN as command values
- #401 tolerance scripting calls should support array telemetry items
- #404 Packet Viewer easy access to edit configuration files
- #405 Telemetry Viewer easy access to edit screen definition
- #423 Add “Cmd router” to CmdTlmServer to support external logging of all commands
- #424 TlmViewer should call update\_widget for a screen with no value items and with CmdTlmServer not running
- #426 Standardize meta data to SYSTEM META packet
- #432 Export processed config files
- #442 Label value widgets should support right aligned labels
- #459 Script Editor code completion enhancements
- #479 Limits Monitor doesn’t detect newly connected targets
- #489 Built in support for limits group enable and disable
- #497 Update serial\_interface.rb to support hardware flow control
- #498 Script helper for activities that should cause an exception
- #511 Make CmdTlmServer run on JRuby
- #512 Create a CRC Protocol
- #513 Create a GUI config file editor
- #516 Recreate COSMOS C Extension Code in Pure Ruby
- #517 Make hostname for tools to connect to CTS API configurable in system.txt
- #519 Replay should support alternate packet log readers

### **Maintenance:**

- #354 Targets need to be path namespaced to avoid conflicts
- #323 Catch Signals in CmdTlmServer
- #341 Document COSMOS JSON API on cosmosrb.com
- #398 Documentation, code cleanup
- #429 Command Endianness and Parameter Endianness
- #437 Remove CMD\_TLM\_VERSION from system.txt
- #438 Cache script text as part of instrumenting script
- #446 Windows 10 Install fails
- #476 Separate apt and yum package install lines
- #477 Deprecate userpath.txt
- #484 require\_file should re-raise existing exception

### **Bug Fixes:**

- #456 Replay doesn’t shut down properly if closed while playing
- #481 show\_backtrace not working in ScriptRunner
- #494 Details dialog crashes for items with LATEST packet

- #502 Target REQUIRE should also search system path
- #506 Don't call read\_interface if data is cached in protocols for another packet

## Migration Notes from COSMOS 3.8.x:

COSMOS 4 includes several breaking changes from the COSMOS 3.x series.

The first and simplest is that the Command and Telemetry Server now opens an additional port at 7780 by default, that provides a router that will send out each command that the system has sent. This can allow external systems to also log all commands sent by COSMOS. For most people this change will be transparent and no updates to your COSMOS configuration will be required.

The second is that the Command and Telemetry Server now always supports a meta data packet called SYSTEM META. This packet will always contain the MD5 sum for the current running COSMOS configuration, the version of COSMOS running, the version of your COSMOS Project, and the version of Ruby being used. You can also add your own requirements for meta data with things like the name of the operator currently running the system, or the name of a specific test you are currently running. In general you shouldn't need to do anything for this change unless you were using the previous metadata functionality in COSMOS. If you were, then you will need to migrate your meta data to the new SYSTEM META packet, and change the parameters in your CmdTlmServer or TestRunner configurations regarding meta data. If you weren't using metadata before, then you will probably just notice this new packet in your log files, and in your telemetry stream.

Finally the most exciting breaking change is in how COSMOS interfaces handle protocols. Before, the COSMOS TCP/IP and Serial interface classes each took a protocol like LENGTH, TERMINATED, etc that defined how packets were delineated by the interface. Now each interface can take a set of one or more protocols. This allows COSMOS to much more easily support nested protocols, such as the frame focused protocols of CCSDS. It also allows for creating helpful reusable protocols such as the new CRC protocol for automatically adding CRCs to outgoing commands and verifying incoming CRCs on telemetry packets. It's a great change, but if you have any custom interface classes you have written, they will probably require some modification. See the Interfaces section at [cosmosrb.com](http://cosmosrb.com) to see how the new interface classes work. We will also be writing up a blog post to help document the process of upgrading. Look for this in a week or two.

To upgrade to the latest version of COSMOS, run "bundle update cosmos" in your COSMOS project folder.

## 3.9.2 / 2017-05-18

### New Features:

- #147 TlmExtractor Full Column Names Mode
- #148 TlmExtractor Share individual columns
- #189 ScriptRunner Breakpoints don't adapt to edits
- #233 Add Config Option to Increase Tcpip Interface Timeout to TlmGrapher
- #280 Method for determining interface packet count
- #313 Add command line option to automatically start ScriptRunner
- #336 Add Log Analyze Feature to TlmExtractor/CmdExtractor
- #395 Implement Stylesheets Throughout
- #408 Easy way to find which targets use an interface?
- #433 Scripting support for TlmViewer close all screens
- #434 TlmViewer option for no resize of screens
- #436 PacketViewer option to ignore target.txt ignored items
- #441 PacketViewer should identify derived items in the GUI

## Maintenance:

None

## Bug Fixes:

- [#417](#) Table Manager not checking ranges
- [#419](#) Support multiple arrays in string based commands

## Migration Notes from COSMOS 3.8.x:

**The Table Manager configuration file format has changed.** Documentation will be updated the first week of April.

You can migrate existing config files using:

```
bundle exec ruby tools\TableManager --convert config\tools\table_manager\old_table_def.txt
```

To upgrade to the latest version of COSMOS, run “bundle update cosmos” in your COSMOS project folder.

## 3.9.1 / 2017-03-29

### New Features:

- [#382](#) CmdTlmServer Start/Stop for background tasks
- [#385](#) Quick access to COSMOS gem code
- [#388](#) Legal Dialog should show COSMOS version
- [#409](#) Update LINC interface to support multiple targets on the same interface

## Maintenance:

- [#369](#) Table Manager refactoring
- [#386](#) Batch file for offline installation

## Bug Fixes:

- [#236](#) Test Runner doesn't support status\_bar
- [#329](#) Using XTCE file instead of .txt cmd\_tlm file didn't work as online docs suggest
- [#378](#) TlmViewer displaying partials in the screen list
- [#402](#) Mac installation is failed - Please help.
- [#411](#) xtce explicit byte order list processing isn't correct
- [#412](#) subscribe\_packet\_data needs to validate parameters

## Migration Notes from COSMOS 3.8.x:

**The Table Manager configuration file format has changed.** Documentation will be updated the first week of April.

You can migrate existing config files using:

```
bundle exec ruby tools\TableManager --convert config\tools\table_manager\old_table_def.txt
```

To upgrade to the latest version of COSMOS, run “bundle update cosmos” in your COSMOS project folder.

## 3.8.3 / 2016-12-06

### New Features:

- #230 Make AUTO\_TARGETS ignore target folders that have already been manually referenced
- #257 Increment received\_count in packet log reader
- #264 Validate command/telemetry conversions during startup
- #292 Target directory for API methods
- #314 Update .bat files to handle spaces in path
- #316 Add option to radiobutton widget to be checked by default
- #326 Script Runner Crash using message\_box with boolean parameter
- #339 Add 'Help' menu item to open cosmosrb.com -> Documentation
- #340 Packet Viewer - Allow select cell and copy value as text
- #357 Add support for mixed endianness within tables
- #359 Table Manager support MIN/MAX UINTX macros

### Maintenance:

- #349 Optimize cmd() to not build commands twice
- #362 restore\_defaults should take an optional parameter to exclude specified parameters
- #365 Windows installer can have issues if .gem files are present in same folder
- TestRunner support for newer Bundler (Abstract error when starting)

### Bug Fixes:

- #322 Udp interface thread does not gracefully shutdown
- #327 TlmGrapher Screenshot in Linux captures the screenshot dialog box
- #332 ERB template local variables don't support strings
- #338 Setting received\_time and received\_count on a packet should clear the read conversion cache
- #342 Cut and Paste Error in top\_level.rb
- #344 CmdTlmServer connect/disconnect button doesn't work after calling connect\_interface from script
- #359 Table Manager doesn't support strings
- #372 TestRunner reinstantiating TestSuite/Test objects every execution

### Migration Notes from COSMOS 3.7.x:

None

To upgrade to the latest version of COSMOS, run "bundle update cosmos" in your COSMOS project folder.

## 3.8.2 / 2016-07-05

### New Features:

### Maintenance:

- COSMOS Downloads graph rake task updates

### Bug Fixes:

- #303 Need to clear read conversion cache on Packet#clone
- #304 Win32 Serial Driver clean disconnect
- #309 Fix Script Runner insert\_return when not running

### Migration Notes from COSMOS 3.7.x:

None

## 3.8.1 / 2016-05-12

### New Features:

- #184 Limits Monitor show green for blue limits items
- #190 Simpler MIN MAX syntax for command definitions
- #254 Get buffer from commands
- #259 Proper support for user selected text editor on linux
- #262 PackerViewer option for listing derived items last
- #271 Time.formatted option for no microseconds
- #288 check\_tolerance should enforce a positive tolerance
- #301 Update use of COSMOS\_DEVEL

### Maintenance:

- #268 xtce\_converter doesn't support byte order list
- #277 Test Runner support for Script Runner options
- #285 xtce converter doesn't support LocationInContainerInBits

### Bug Fixes:

- #256 Defining initialize method in Cosmos::Test class breaks the class when using Test Selection in TestRunner
- #273 Wrap Qt::Application.instance in main\_thread
- #287 Installer issue on newer versions of Ubuntu and Debian related to libssl
- #293 Units applied after a read\_conversion that returns a string modifies cached conversion value
- #294 String#convert\_to\_value should always just return the starting string if the conversion fails
- #298 COSMOS IoMultiplexer breaks gems that invoke stream operator on STDOUT/ERR

### Migration Notes from COSMOS 3.7.x:

None

## 3.8.0 / 2016-02-26

With this release COSMOS now has initial support for the XTCE Command and Telemetry Definition standard.

### New Features:

- #251 Create COSMOS XTCE Converter
- #252 Add polling rate command line option to PacketViewer

### Bug Fixes:

- [#245](#) TlmGrapher Crashes on Inf
- [#248](#) Can't script commands containing 'with' in the name

### Migration Notes from COSMOS 3.7.x:

None

## 3.7.1 / 2015-12-29

### Bug Fixes:

- [#228](#) Fix typo in udp\_interface
- [#231](#) MACRO\_APPEND with multiple items not working
- [#235](#) Improve IntegerChooser and FloatChooser Validation
- [#236](#) TestRunner doesn't support status\_bar
- [#240](#) Make sure super() is called in all bundled conversion classes
- [#241](#) Don't reformat BLOCK data types with a conversion in Structure#formatted

### Migration Notes from COSMOS 3.6.x:

1. Background task arguments are now broken out instead of being received as a single array
2. udp\_interface now takes an optional argument for bind\_address
3. line\_graph\_script has been significantly updated to support modifying plots from the script.

## 3.7.0 / 2015-11-25

### New Features:

- [#213](#) Vertical Limits Bar
- [#214](#) TlmGrapher show full date for plotted points
- [#219](#) State Color Widget
- [#225](#) Set Bind Address in UDP interface

### Maintenance:

- [#223](#) C Extension Improvements

### Bug Fixes:

- [#199](#) Investigate TlmGrapher Formatted Time Item
- [#211](#) Background task with arguments not working
- [#217](#) Graph Right Margin Too Small

### Migration Notes from COSMOS 3.6.x:

1. Background task arguments are now broken out instead of being received as a single array
2. udp\_interface now takes an optional argument for bind\_address
3. line\_graph\_script has been significantly updated to support modifying plots from the script.

## 3.6.3 / 2015-10-30

### New Features:

- #200 ScriptRunner Find Dialog Does Not Cross Windows
- #201 Table Manager to support arbitrary inputs on State Fields
- #209 Add UTS Timestamp Flag to TlmGrapher Plots

### **Maintenance:**

- #194 Allow up to one minute for TlmViewer to start when calling display() from a script
- #203 load\_utility should raise LoadError like load and require
- #205 Add testing for array and matrix

### **Bug Fixes:**

- #191 Installing COSMOS Issue on Windows 7
- #193 Fix ask() on linux and qt 4.6.2
- #197 Improve linc interface

### **Migration Notes from COSMOS 3.5.x:**

None

## **3.6.2 / 2015-08-10**

### **Bug Fixes:**

- #187 Must require tempfile in config\_parser.rb on non-windows systems

### **Migration Notes from COSMOS 3.5.x:**

None

## **3.6.1 / 2015-08-10**

### **Bug Fixes:**

- #185 target.txt order not being preserved

### **Migration Notes from COSMOS 3.5.x:**

None

## **3.6.0 / 2015-08-07**

Huge new feature in this release: All COSMOS configuration files are now interpreted with the ERB preprocessor! This allows you to use Ruby code within the configuration files to help build them. You can also render partials of common information such as packet headers so you only have to define them once. See the INST target in the updated Demo project for examples.

### **Bug Fixes:**

- #168 Select unreliable unblocks when closing sockets on linux
- #177 MACRO\_APPEND in descending order is broken
- #179 ScriptRunnerFrame Context Menu Crash
- #182 Overriding LOG\_WRITERS in cmd\_tlm\_server.txt can cause issues

## **New Features:**

- [#170](#) Consider supporting a preprocessor over COSMOS config files
- [#171](#) Script Runner should have file open and save GUI dialogs
- [#174](#) Add View in Command Sender in Server

## **Maintenance:**

- [#80](#) Investigate performance of nonblocking IO without exceptions

## **Migration Notes from COSMOS 3.5.x:**

None

## **3.5.3 / 2015-07-14**

### **Bug Fixes:**

- [#169](#) Make windows bat files support running outside of the current directory

## **New Features:**

- N/A

## **Maintenance:**

- N/A

## **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 3.5.1

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.

## **3.5.2 / 2015-07-14**

### **Bug Fixes:**

- [#167](#) Use updated url for wkhtmltopdf downloads

## **New Features:**

- [#166](#) Add install script for Ubuntu

## **Maintenance:**

- N/A

## **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 3.5.1

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.

## 3.5.1 / 2015-07-08

This release fixes a bug and completes the installation scripts for linux/mac.

### Bug Fixes:

- [#165](#) Change launch\_tool to tool\_launch in Launcher

### New Features:

- N/A

### Maintenance:

- [#102](#) Create Installation Scripts

### 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 3.5.1

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.

## 3.5.0 / 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

BACK

NEXT

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Custom Search



Proudly hosted by **GitHub**



Navigate the docs... ▾

## Papers

Improve this page

### AMOS 2017

- [A Cloud-based, Open-Source, Command-and-Control Software Paradigm for Space Situational Awareness \(SSA\)](#)

### Small Sat 2016

- [Ball Aerospace COSMOS Open Source Command and Control System](#)

### Aerospace Testing Seminar 2015

- [Ball Aerospace's COSMOS Open Source Test System](#)
- [System Level Integration and Test Leveraging Software Unit Testing Techniques](#)
- [The Hidden Benefits of Automated Testing](#)



#### Giving a presentation on COSMOS? Please let us know!

We love seeing how people are using Ball Aerospace COSMOS to solve their problems. Please email [rmelton@ball.com](mailto:rmelton@ball.com) and let us know about your presentation.

BACK

NEXT

The contents of this website are © 2018 Ball Aerospace under the terms of the GPLv3 License.

Site design derived from the Jekyll project documentation which is licensed under the terms of the MIT License.

Proudly hosted by GitHub