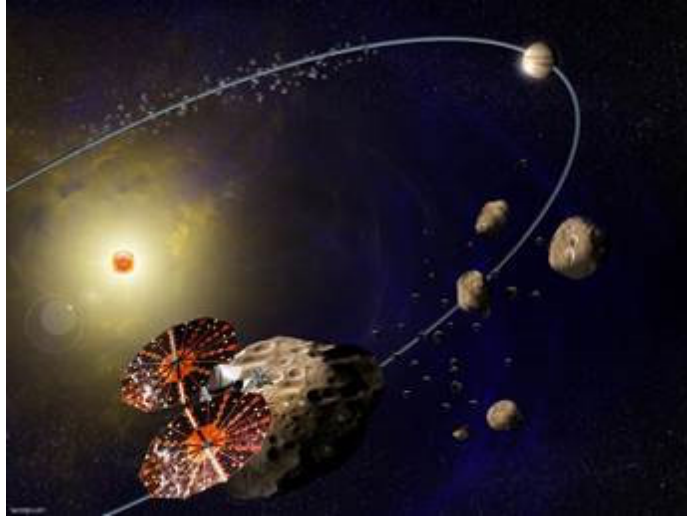


1 L'LORRI GSEOS User Manual



L'LORRI GSEOS User Manual

GSE Software, Inc. 2018

1.1 Version

Document Revision: 1
L'LORRI GSEOS Version: 1.0.004
Date: May/18/2018

Change History

Revision	Date	Changes
1	5/18/2018	Initial version.

1.2 Introduction

The LUCY LORRI S/C Emulator (SCE) implements the LUCY LORRI mission specific requirements in the GSEOS environment. The system has been implemented according to spec:

TBD

The GSEOS to SCE interface was implemented according to:

LUCY LORRI GSEOS to Emulator ICD (Rev. 1)
May 18, 2018

The emulator implements the SCE (Spacecraft Emulator) to GSEOS interface based on the FTF message transfer. It allows the L'LORRI spacecraft emulator engineers to issue low level commands (in the form of raw byte streams) and send them to the SCE using the proper FTF message protocol to the Opal Kelly board via USB. It also allows to display incoming (SCE -> GSEOS) telemetry, analog, and status data. The next higher decoding level decodes the incoming telemetry streams according to the L'LORRI protocol stack (IP/UDP/CIP). Command data is wrapped into the proper protocol levels before being sent to the SCE.

The following sections give a more detailed overview of the configuration and usage of the L'LORRI GSEOS and the predefined system screens and tools to monitor the SCE - GSEOS emulator traffic as well as instrument data.

1.3 Installation

The latest version of the L'LORRI GSEOS can be downloaded from the GSEOS web site at <http://www.gseos.com/download.php>.

The 7z archive is password encrypted. When unzipping the archive make sure you preserve the folders in the archive. The entire system can be x-copy deployed, that means you can copy the entire folder to any location on any machine without impacting the installation. There are no external dependencies other than the main system libraries. Please use a 7z compatible archive tool.

Linux Installation

GSEOS is a 32-bit application and requires the according Linux i386 32-bit libraries if running on a 64-bit platform. We currently support both Ubuntu 14.04LTS and 16.04LTS both in 32-bit and 64-bit.

To install the 32-bit libraries please run the following commands (as root, sudo):

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386 libstdc++6:i386
sudo apt-get install libx11-6:i386
sudo apt-get install libxrender1:i386
sudo apt-get install libxrandr2:i386
sudo apt-get install libxft2:i386
sudo apt-get install libsm6:i386
sudo apt-get install libsqlite3-0:i386
```

GSEOS includes all other libraries and should not have other dependencies. However, please check with ldd if there are any unresolved references on your installation and install the appropriate packages.

Opal Kelly

The emulator design is based on the Opal Kelly XEM 7310 board and requires the Opal Kelly Frontpanel host software.

It is not necessary to install any Opal Kelly drivers. The archive comes with the 32-bit Opal Kelly drivers. You don't need to install these, the only system configuration you have to make is the addition of one file to the following directory:

```
sudo cp 60-opalkelly.rules /etc/udev/rules.d/
```

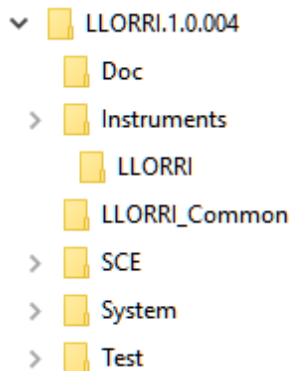
The file 60-opalkelly.rules is in the root folder of the L'LORRI GSEOS distribution. This file includes a generic udev rule to set the permissions on all attached Opal Kelly USB devices to allow user access. Once this file is in place, you will need to reload the rules by either rebooting or using the following command:

```
sudo /sbin/udevadm control --reload_rules
```

With these files in place, the Linux device system should automatically provide write permissions to XEM devices attached to the USB.

Directory Structure

The L'LORRI directory tree should look similar to the following:



The SCE sub-directory contains all the SCE specific screens and startup configuration. The L'LORRI_Common folder holds mission common configuration files like the L'LORRI block definitions, text reference files, etc. The System folder contains binary GSEOS system files. The Instruments folder contains the individual instrument folder (L'LORRI).

The Doc folder contains this document and the general GSEOS documentation.

The only changes to customize your system should be made in the Instruments/L'LORRI folder! This allows for simply x-copy upgrades. Please don't modify any of the other folders or files! If you think there is a need to do so please check with us first.

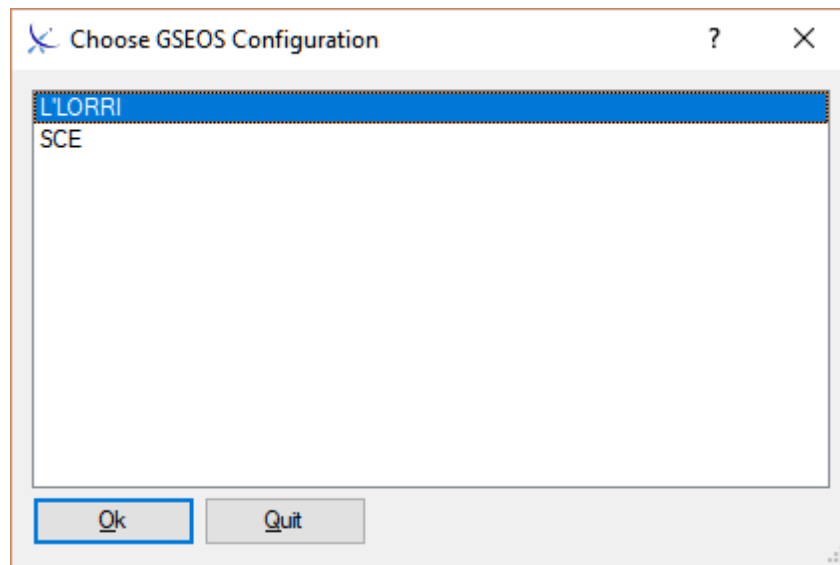
1.4 Quick Start

Once the archive is extracted you will find a folder 'L'LORRI' in your destination directory (it might include a version number like 2.1.007). The L'LORRI folder is considered the GSEOS root folder. As mentioned earlier you can move and copy this folder to any location

on your machine or copy it to a different PC altogether. You can also run GSEOS from a USB drive.

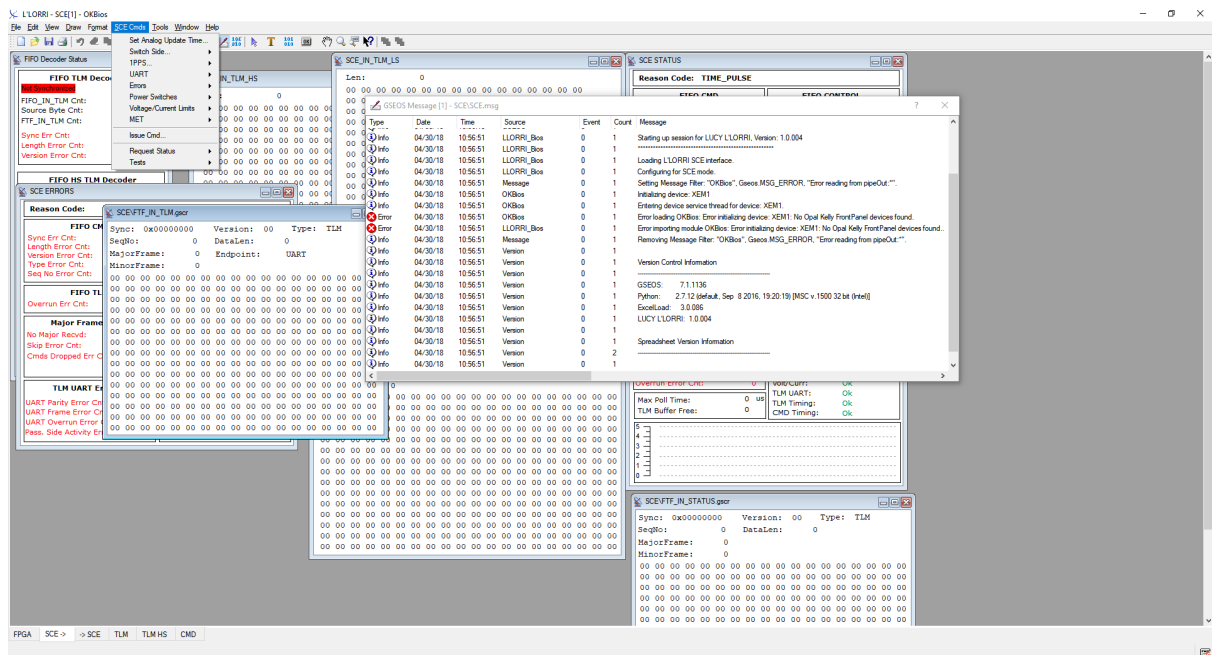
To start the emulator software run `./gseos` in the root folder. Make sure that `./gseos` has execute permissions set. If you extract the files on a Windows machine and move it to Linux you will lose the file permissions and need to manually set execute permissions.

When starting GSEOS you will see a selection dialog that allows you to start either the SCE configuration or the L'RLAPH instrument configuration. (You can add your own sub-configurations if you choose to do so, please refer to the general GSEOS documentation for specifics on how to configure configuration selection dialogs and configuration files). For the rest of this manual we will only cover the general SCE configuration. (The instrument configuration also has access to the SCE and common L'LORRI screens and configuration files). The following picture shows the selection dialog:



The 'SCE' entry runs the L'LORRI S/C Emulator configuration. The L'LORRI entry will launch the L'LORRI instrument specific setup.

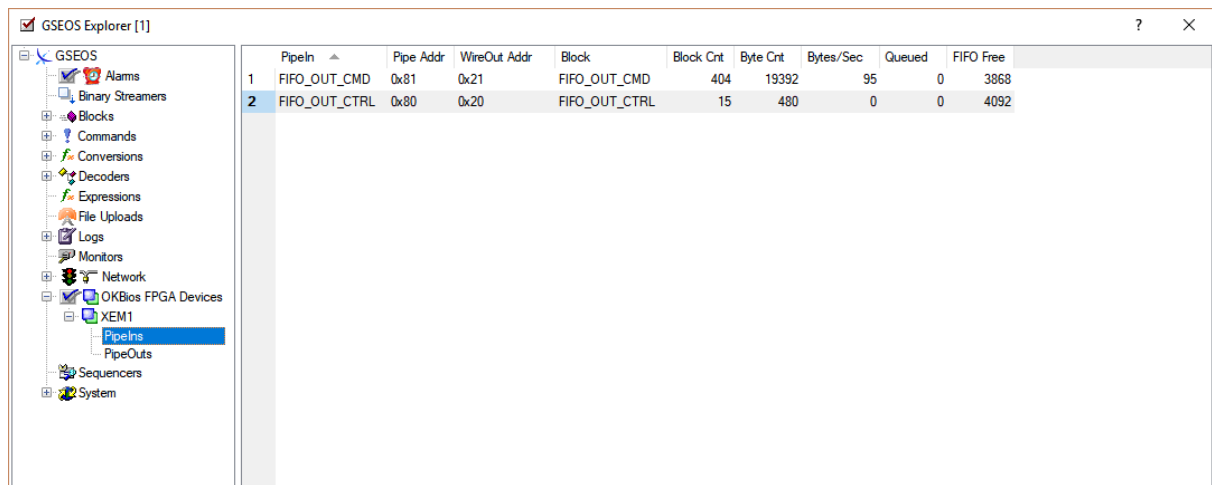
The image below shows a screenshot of the L'LORRI GSEOS:



The main desktop has six tabs: FPGA, SCE Outbound, SCE Inbound, TLM, TLM HS, and CMD. The FPGA and SCE tabs contain low level SCE related screens. The TLM (TLM HS) and CMD tabs contain higher level screens with decoded telemetry and command status information.

To communicate with the SCE you have to install the Opal Kelly drivers (version 4.5 or above) and plug in the USB connector to the SCE.

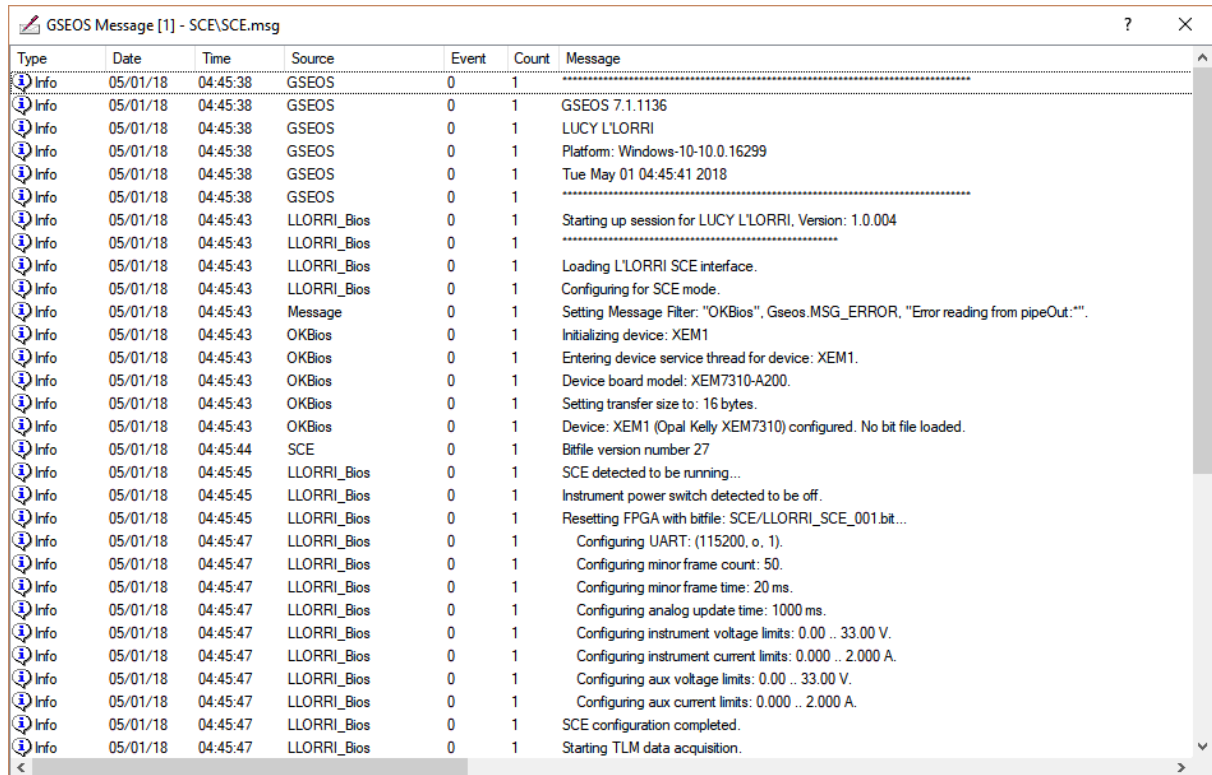
If everything is set up correctly you will see the Opal Kelly connections in the GSEOS Explorer. Push F10 to show the Explorer window. Navigate to the OKBios FPGA Devices node. You should find the XEM1 device under the Opal Kelly Devices node:



If an error occurred connecting to the Opal Kelly device there will be an error entry in the GSEOS message window.

Let's assume the system is configured properly and the Opal Kelly hardware is connected to the emulator.

On startup the SCE is configured with your SCE [configuration](#). The message window shows the configuration applied to the SCE (you can toggle the message window using the F11 key):



Type	Date	Time	Source	Event	Count	Message
Info	05/01/18	04:45:38	GSEOS	0	1	*****
Info	05/01/18	04:45:38	GSEOS	0	1	GSEOS 7.1.1136
Info	05/01/18	04:45:38	GSEOS	0	1	LUCY L'LORRI
Info	05/01/18	04:45:38	GSEOS	0	1	Platform: Windows-10-10.0.16299
Info	05/01/18	04:45:38	GSEOS	0	1	Tue May 01 04:45:41 2018
Info	05/01/18	04:45:38	GSEOS	0	1	*****
Info	05/01/18	04:45:43	LLORRI_Bios	0	1	Starting up session for LUCY L'LORRI, Version: 1.0.004
Info	05/01/18	04:45:43	LLORRI_Bios	0	1	*****
Info	05/01/18	04:45:43	LLORRI_Bios	0	1	Loading L'LORRI SCE interface.
Info	05/01/18	04:45:43	LLORRI_Bios	0	1	Configuring for SCE mode.
Info	05/01/18	04:45:43	Message	0	1	Setting Message Filter: "OKBios", Gseos.MSG_ERROR, "Error reading from pipeOut: ""
Info	05/01/18	04:45:43	OKBios	0	1	Initializing device: XEM1
Info	05/01/18	04:45:43	OKBios	0	1	Entering device service thread for device: XEM1.
Info	05/01/18	04:45:43	OKBios	0	1	Device board model: XEM7310-A200.
Info	05/01/18	04:45:43	OKBios	0	1	Setting transfer size to: 16 bytes.
Info	05/01/18	04:45:43	OKBios	0	1	Device: XEM1 (Opal Kelly XEM7310) configured. No bit file loaded.
Info	05/01/18	04:45:44	SCE	0	1	Bitfile version number 27
Info	05/01/18	04:45:45	LLORRI_Bios	0	1	SCE detected to be running...
Info	05/01/18	04:45:45	LLORRI_Bios	0	1	Instrument power switch detected to be off.
Info	05/01/18	04:45:45	LLORRI_Bios	0	1	Resetting FPGA with bitfile: SCE/LLORRI_SCE_001.bit...
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring UART: (115200, o, 1).
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring minor frame count: 50.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring minor frame time: 20 ms.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring analog update time: 1000 ms.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring instrument voltage limits: 0.00 .. 33.00 V.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring instrument current limits: 0.000 .. 2.000 A.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring aux voltage limits: 0.00 .. 33.00 V.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring aux current limits: 0.000 .. 2.000 A.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	SCE configuration completed.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Starting TLM data acquisition.

The version number of the SCE is listed in the message file as well. This way you can make sure you are running the correct SCE version (bitfile). If you install the archive from a download this should always contain the proper version of the SCE bitfile for the according L'LORRI GSEOS version.

There is a way to override the default setting in case you have a newer bitfile version that you want to try. However, we don't recommend using this approach since this might lead to running incompatible versions of GSEOS and the SCE software.

1.5 Configuration

The basic SCE configuration is set in the gseos.ini file in your specific instrument folder. Any customization for your specific instrument should only require to change files in your specific folder, please do not modify files in the SCE folder or the L'LORRI_Common folder since these may be updated with newer releases!

There is a 'selection' gseos.ini in the top level that lists the available configurations.

Currently only L'LORRI, and SCE are supported. If you have additional configurations you can add your additional configurations to the top level file. You can also have nested 'selection' configurations on your Instrument level, e.g. L'LORRI EM, L'LORRI FM, L'LORRI MOC RT, etc. The advantage of adding additional configurations to the top level gseos.ini is that you don't have to go through two navigation levels when selecting your configuration. (However, that would require you to modify the top level gseos.ini which will be overwritten with new releases!)

Please refer to the general GSEOS documentation for more information on the various configuration options.

The following sections/entries let you configure the system:

[\[Bios\]](#)

The [Bios] section lets you configure general Bios settings like starting Time, Protocol settings, TLM transaction limit, raw commanding, etc.

[Protocol Settings](#)

The LUCY protocol stack consists of IP, UDP, CIP, and IDP layers. The IP protocol requires to specify a source and destination IP address, the UDP layer requires a source and destination port. For L'LORRI the source and destination ports are identical. The IP addresses you specify must be in dotted numerical form, DNS names can NOT be used. You can specify the IP addresses and the UDP port using the following entries in the [Bios] section of the gseos.ini file:

```
[Bios]
Instr_IP_Address = 1.2.3.4
Instr_Port       = 3333
CDH_IP_Address  = 5.6.7.8
```

[Time Settings](#)

You can configure the spacecraft time that is used when the system starts up with the [Bios] section as well. The time specified is the number of seconds since the start of the epoch. It can be a floating point value to indicate fractions of a second. You can also configure the time increment, that is the increment that gets added to the time for every TIME TIC pulse.

The default for the time increment is 1s. Mind you that the TimeIncrement setting does not modify the physical time between two time pulses, only the time that gets clocked out in the TIME message. If you don't specify a start time the current PC time will be used. This has the advantage that if you record test data the time will be different every time you run a test.

```
[Bios]
StartTime = 344443.456
TimeIncrement = 1.5
```

[Low speed TLM transaction limit](#)

The low speed transaction size can be limited with the Max_TLM_LS_TxBytes setting. The default is 4700 bytes. This setting can be configured on an instrument basis depending on the instruments needs. If the SBC receives a low speed telemetry transaction (IP packet) that exceeds the configured limit the DecStatus_TLM_LS_IP.MaxTxLenExceededErrCnt error counter will be incremented and transaction will be discarded. To set your custom transaction limit you can use the following setting to your gseos.ini file [Bios] section:

```
[Bios]
Max_TLM_LS_TxBytes = 2000
```

The minimum limit for the transaction size must allow for the IP/UDP/CIP headers.

Raw Command Mode

The emulator can be in one of two modes: Regular command mode or Raw command mode. The default is regular command mode. To switch to raw command mode you have to add the following setting to your gseos.ini file [Bios] section:

```
[Bios]
RawCmdMode = Yes
```

For more information about commanding in general please refer to the [Command Handling](#) section, for information about raw commanding check the [Raw Commanding](#) chapter.

SCE UART Configuration

The UART between the SCE and instrument can be configured with the [UART] section. You can set the baudrate, parity, and number of stopbits:

```
[UART]
Baudrate   = 115200
Parity     = o
Stopbits   = 1
```

The above configuration shows the default settings. There is no need to specify these settings if the default settings are fine for you.

SCE Configuration

SCE specific settings can be configured with the [SCE] section. The following options are recognized:

Entry	Default	Description
MaxCmdQueueLen	3	The maximum length of the SCE command queue. This determines how many major frames of command data GSEOS can upload to the FPGA. Say this is set to 5, the next five seconds (given the system is configured for 1s major frames) worth of command data will be uploaded to the FPGA (given that sufficient FIFO space is available). This also means that the delay between issuing a command and it reaching the instrument will be five or more seconds.
StartAcquisition	Auto	This setting allows you to defer the emulator starting acquisition of TLM traffic. See also the FAQ on ' How do I defer the emulator startup? '. The default setting is 'Auto' which means the emulator starts up as soon as possible (after initialization). To manually control this you can set this entry to 'Manual'. In Manual mode the emulator will not start up and you have to

		manually issue the <code>SCE_CONFIG_START_ACQUISITION()</code> command to commence acquisition of TLM data.
AnalogUpdateTime	1000	The time in ms GSEOS gets updated analog samples from the SCE.
MinorFrameCnt	50	The number of minor frames per major frame (1PPS period). Must be between 1 and 1000.
MinorFrameTime	20	The minor frame time in ms. The product of MinorFrameCnt and MinorFrameTime determines the 1PPS interval (this should be 1sec for nominal operations).
InstrVoltLimitHigh	33.00	The high instrument voltage limit in V. The maximum value for this setting is 40.00V. The maximum resolution is 10mV.
InstrVoltLimitLow	0.00	The low instrument voltage limit in V. The maximum value for this setting is 40.00V. The maximum resolution is 10mV. It must be less than the InstrVoltLimitHigh setting.
InstrCurrLimitHigh	2.000	The high instrument current limit in A. The maximum value for this setting is 3.000A. The maximum resolution is 10mA.
InstrCurrLimitLow	0.000	The low instrument current limit in A. The maximum value for this setting is 3.000A. The maximum resolution is 10mA. It must be less than the InstrCurrLimitHigh setting.
AuxVoltLimitHigh	33.00	The high aux. voltage limit in V. The maximum value for this setting is 40.00V. The maximum resolution is 10mV.
AuxVoltLimitLow	0.00	The low aux. voltage limit in V. The maximum value for this setting is 40.00V. The maximum resolution is 10mV. It must be less than the AuxVoltLimitHigh setting.
AuxCurrLimitHigh	2.000	The high aux. current limit in A. The maximum value for this setting is 3.000A. The maximum resolution is 10mA.
AuxCurrLimitLow	0.000	The low aux. current limit in A. The maximum value for this setting is 3.000A. The maximum resolution is 10mA. It must be less than the AuxCurrLimitHigh setting.

Sample

```
[SCE]
AnalogUpdateTime      = 2000
MinorFrameCnt         = 100
MinorFrameTime        = 10

InstrVoltLimitHigh    = 29.22
InstrVoltLimitLow     = 2.01
InstrCurrLimitHigh    = 24.000
```

These settings are sent to the SCE on initial connection. The configurations sent to the SCE are reported in the GSEOS message window. If you configure default settings no updates are sent to the SCE.

GSEOS Message [1] - SCE\SCE.msg						
Type	Date	Time	Source	Event	Count	Message
Info	05/01/18	04:45:38	GSEOS	0	1	*****
Info	05/01/18	04:45:38	GSEOS	0	1	GSEOS 7.1.1136
Info	05/01/18	04:45:38	GSEOS	0	1	LUCY L'LORRI
Info	05/01/18	04:45:38	GSEOS	0	1	Platform: Windows-10-10.0.16299
Info	05/01/18	04:45:38	GSEOS	0	1	Tue May 01 04:45:41 2018
Info	05/01/18	04:45:38	GSEOS	0	1	*****
Info	05/01/18	04:45:43	LLORRI_Bios	0	1	Starting up session for LUCY L'LORRI, Version: 1.0.004
Info	05/01/18	04:45:43	LLORRI_Bios	0	1	*****
Info	05/01/18	04:45:43	LLORRI_Bios	0	1	Loading L'LORRI SCE interface.
Info	05/01/18	04:45:43	LLORRI_Bios	0	1	Configuring for SCE mode.
Info	05/01/18	04:45:43	Message	0	1	Setting Message Filter: "OKBios", Gseos.MSG_ERROR, "Error reading from pipeOut: ""
Info	05/01/18	04:45:43	OKBios	0	1	Initializing device: XEM1
Info	05/01/18	04:45:43	OKBios	0	1	Entering device service thread for device: XEM1.
Info	05/01/18	04:45:43	OKBios	0	1	Device board model: XEM7310-A200.
Info	05/01/18	04:45:43	OKBios	0	1	Setting transfer size to: 16 bytes.
Info	05/01/18	04:45:43	OKBios	0	1	Device: XEM1 (Opal Kelly XEM7310) configured. No bit file loaded.
Info	05/01/18	04:45:44	SCE	0	1	Bitfile version number 27
Info	05/01/18	04:45:45	LLORRI_Bios	0	1	SCE detected to be running...
Info	05/01/18	04:45:45	LLORRI_Bios	0	1	Instrument power switch detected to be off.
Info	05/01/18	04:45:45	LLORRI_Bios	0	1	Resetting FPGA with bitfile: SCE/LLORRI_SCE_001.bit...
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring UART: (115200, o, 1).
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring minor frame count: 50.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring minor frame time: 20 ms.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring analog update time: 1000 ms.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring instrument voltage limits: 0.00 .. 33.00 V.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring instrument current limits: 0.000 .. 2.000 A.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring aux voltage limits: 0.00 .. 33.00 V.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Configuring aux current limits: 0.000 .. 2.000 A.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	SCE configuration completed.
Info	05/01/18	04:45:47	LLORRI_Bios	0	1	Starting TLM data acquisition.

Command Transactions

The timing of individual commands can be controlled using command transactions. You can specify command transactions in the gseos.ini configuration file as well. The chapter on [Command Configuration](#) details how to configure command transactions.

1.6 System Screens

The L'LORRI screens are organized into different groups. The FPGA related screens can be found on the FPGA desktop tab and are described in the [FPGA](#) section. Screens displaying the incoming data from SCE to GSEOS can be found in the [Incoming Data](#) section. Screens displaying the outgoing data from GSEOS to the SCE are described in the [Outgoing Data](#) section. Screens related to commanding can be found in the [CMD Data](#) section.

[FPGA](#)

OKBiosDecStatus:	General OKBios decoder and FIFO status.
OkBiosWireStatus:	Screen displaying the OKBios Wire status.
FIFO Decoder Status:	Displays the incoming FIFO decoder status.
FIFO_IN_TLM:	TLM FIFO incoming data.
FIFO_IN_TLM_HS:	TLM HS FIFO incoming data.
FIFO_IN_STATUS:	STATUS FIFO incoming data.
FIFO_OUT_CMD:	CMD FIFO outgoing data.
FIFO_OUT_CTRL:	CTRL FIFO outgoing data.

[Incoming Data](#)

FTF Decoder Status:	GSEOS FTF decoder status.
FTF_IN_TLM:	Incoming FTF frames with TLM data.
FTF_IN_STATUS:	Incoming FTF frames with STATUS data.
SCE_IN_TLM:	TLM data.
SCE_IN_TLM_HS:	TLM HS data.
SCE_IN_STATUS:	SCE STATUS data.
SCE_IN_SETTINGS:	SCE SETTINGS data.
SCE_IN_MESSAGE:	SCE Message data.
SCE_IN_ERRORS:	SCE Error counters.

[Outgoing Data](#)

FTF Out Decoder Status:	GSEOS FTF outgoing FTF decoder status.
FTF_OUT_CTRL:	Commands destined for the SCE.
FTF_OUT_CMD:	Outgoing command data.
SCE SETTINGS:	The SCE settings as reported by the SCE.
SCE ERRORS:	The SCE errors as reported by the SCE.

[TLM](#)

FIFO_IN_TLM:	Unsynchronized raw FIFO TLM data stream.
FTF_IN_TLM:	Displays synchronized inbound TLM data as FTF frames.
TLM:	Displays the TLM data which is a complete CIP packet.
TLM Decoder Status:	Displays the TLM decoder status.

[TLM HS](#)

FIFO_IN_TLM_HS:	Unsynchronized raw FIFO TLM HS data stream.
FTF_IN_TLM_HS:	Displays synchronized inbound TLM HS data as FTF frames.
TLM_HS:	Displays the TLM HS data.
TLM_HS Decoder Status:	Displays the TLM HS decoder status.

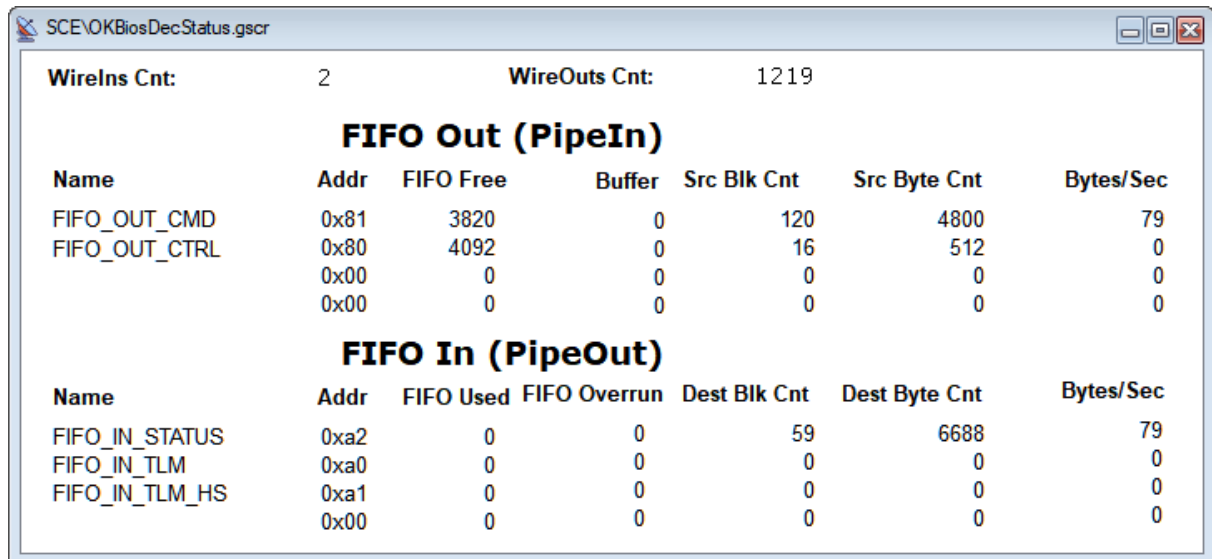
[CMD Data](#)

BinCmd:	Displays the CmdString and BinCmd blocks.
FTF_OUT_CMD:	Outgoing command data.
CMD Decoder Status:	L'LORRI command decoder status.

1.6.1 FPGA

The FPGA related screens display information on the lowest interface level like FIFOs and Wires of the FPGA.

OkBiosDecStatus



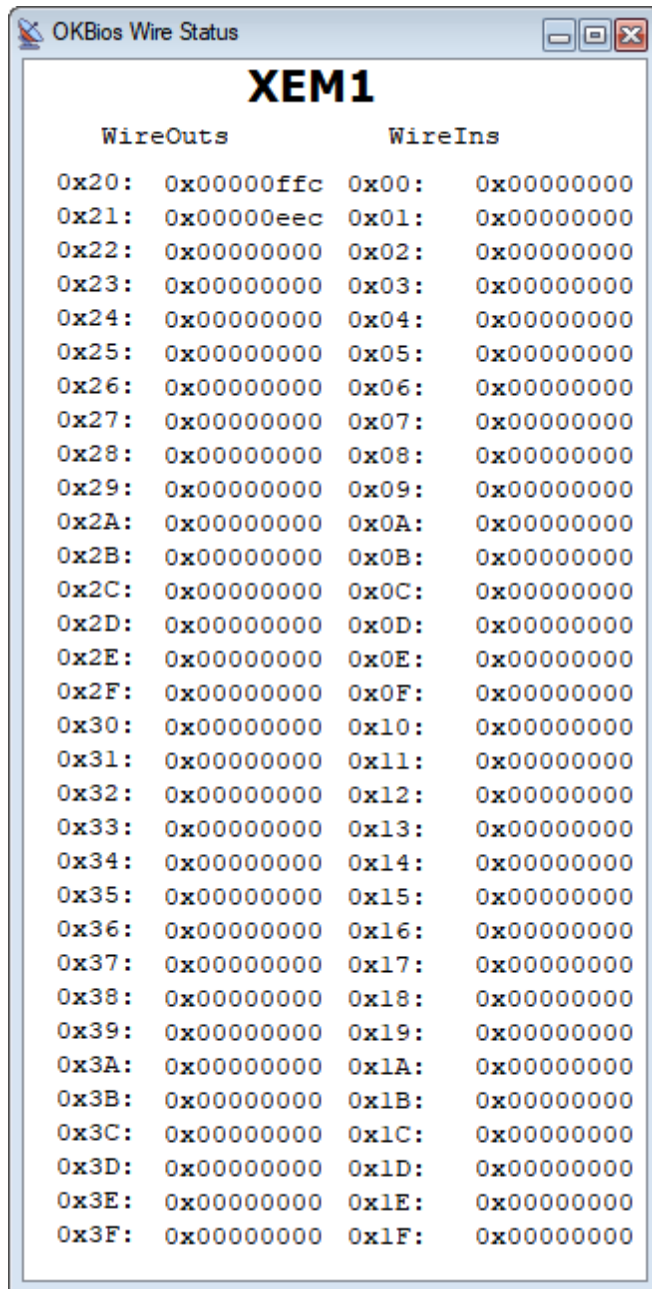
The screenshot shows a window titled "SCE\OKBiosDecStatus.gscr" with a status bar at the top indicating "WireIns Cnt: 2" and "WireOuts Cnt: 1219". The main content area is divided into two sections: "FIFO Out (PipeIn)" and "FIFO In (PipeOut)".

FIFO Out (PipeIn)						
Name	Addr	FIFO Free	Buffer	Src Blk Cnt	Src Byte Cnt	Bytes/Sec
FIFO_OUT_CMD	0x81	3820	0	120	4800	79
FIFO_OUT_CTRL	0x80	4092	0	16	512	0
	0x00	0	0	0	0	0
	0x00	0	0	0	0	0

FIFO In (PipeOut)						
Name	Addr	FIFO Used	FIFO Overrun	Dest Blk Cnt	Dest Byte Cnt	Bytes/Sec
FIFO_IN_STATUS	0xa2	0	0	59	6688	79
FIFO_IN_TLM	0xa0	0	0	0	0	0
FIFO_IN_TLM_HS	0xa1	0	0	0	0	0
	0x00	0	0	0	0	0

This screen displays the OkBios decoder status, it lists the incoming and outgoing FIFOs and the amount of data transferred in/out.

OkBiosWireStatus



The screenshot shows a window titled "OKBios Wire Status" with a close button. Inside the window, the text "XEM1" is displayed in large, bold, black letters. Below this, there are two columns of data: "WireOuts" and "WireIns". Each column contains 20 rows of hexadecimal values. The "WireOuts" column starts with 0x00000ffc and the "WireIns" column starts with 0x00000000. All other values in both columns are 0x00000000.

WireOuts	WireIns
0x20: 0x00000ffc	0x00: 0x00000000
0x21: 0x00000eec	0x01: 0x00000000
0x22: 0x00000000	0x02: 0x00000000
0x23: 0x00000000	0x03: 0x00000000
0x24: 0x00000000	0x04: 0x00000000
0x25: 0x00000000	0x05: 0x00000000
0x26: 0x00000000	0x06: 0x00000000
0x27: 0x00000000	0x07: 0x00000000
0x28: 0x00000000	0x08: 0x00000000
0x29: 0x00000000	0x09: 0x00000000
0x2A: 0x00000000	0x0A: 0x00000000
0x2B: 0x00000000	0x0B: 0x00000000
0x2C: 0x00000000	0x0C: 0x00000000
0x2D: 0x00000000	0x0D: 0x00000000
0x2E: 0x00000000	0x0E: 0x00000000
0x2F: 0x00000000	0x0F: 0x00000000
0x30: 0x00000000	0x10: 0x00000000
0x31: 0x00000000	0x11: 0x00000000
0x32: 0x00000000	0x12: 0x00000000
0x33: 0x00000000	0x13: 0x00000000
0x34: 0x00000000	0x14: 0x00000000
0x35: 0x00000000	0x15: 0x00000000
0x36: 0x00000000	0x16: 0x00000000
0x37: 0x00000000	0x17: 0x00000000
0x38: 0x00000000	0x18: 0x00000000
0x39: 0x00000000	0x19: 0x00000000
0x3A: 0x00000000	0x1A: 0x00000000
0x3B: 0x00000000	0x1B: 0x00000000
0x3C: 0x00000000	0x1C: 0x00000000
0x3D: 0x00000000	0x1D: 0x00000000
0x3E: 0x00000000	0x1E: 0x00000000
0x3F: 0x00000000	0x1F: 0x00000000

The OkBiosWireStatus shows the incoming and outgoing wire values. The WireIns will only get set if you publish the according WireIns block, if you set the wires programmatically they won't be reflected in the WireIns block.

FIFO Decoder Status

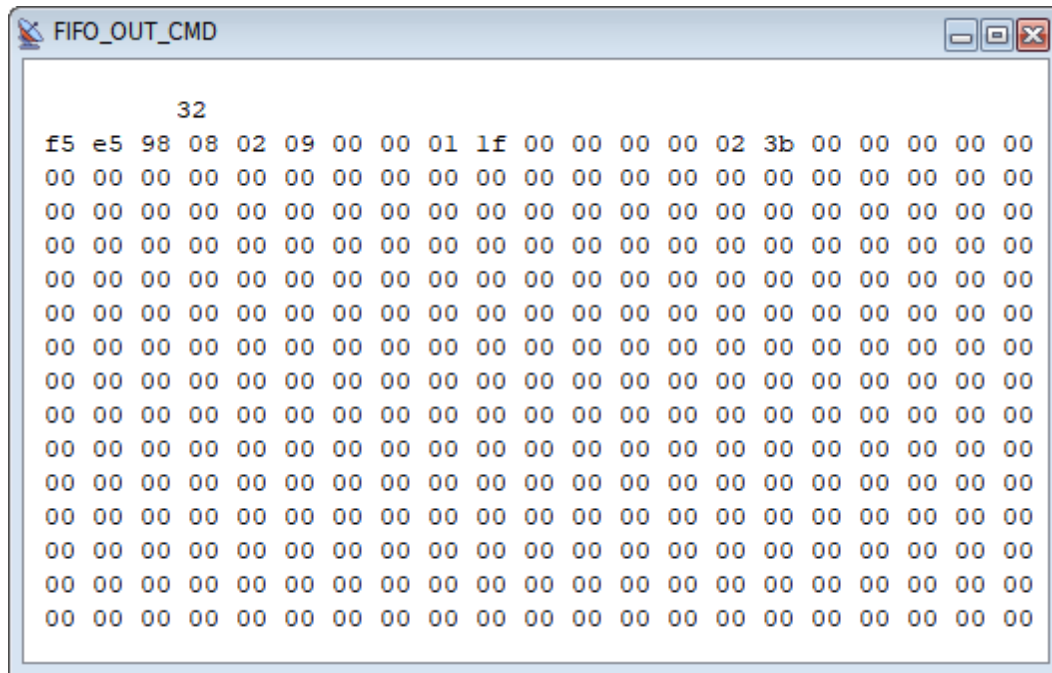
FIFO TLM Decoder	
Synchronized	
FIFO_IN_TLM Cnt:	14
Source Byte Cnt:	672
FTF_IN_TLM Cnt:	14
Sync Err Cnt:	0
Length Error Cnt:	0
Version Error Cnt:	0

FIFO HS TLM Decoder	
Not Synchronized	
FIFO_IN_TLM_HS Cnt:	0
Source Byte Cnt:	0
FTF_IN_TLM_HS Cnt:	0
Sync Err Cnt:	0
Length Error Cnt:	0
Version Error Cnt:	0

FIFO STATUS Decoder	
Synchronized	
FIFO_IN_STATUS Cnt:	184
Source Byte Cnt:	16560
FTF_IN_STATUS Cnt:	210
Sync Err Cnt:	0
Length Error Cnt:	0
Version Error Cnt:	0

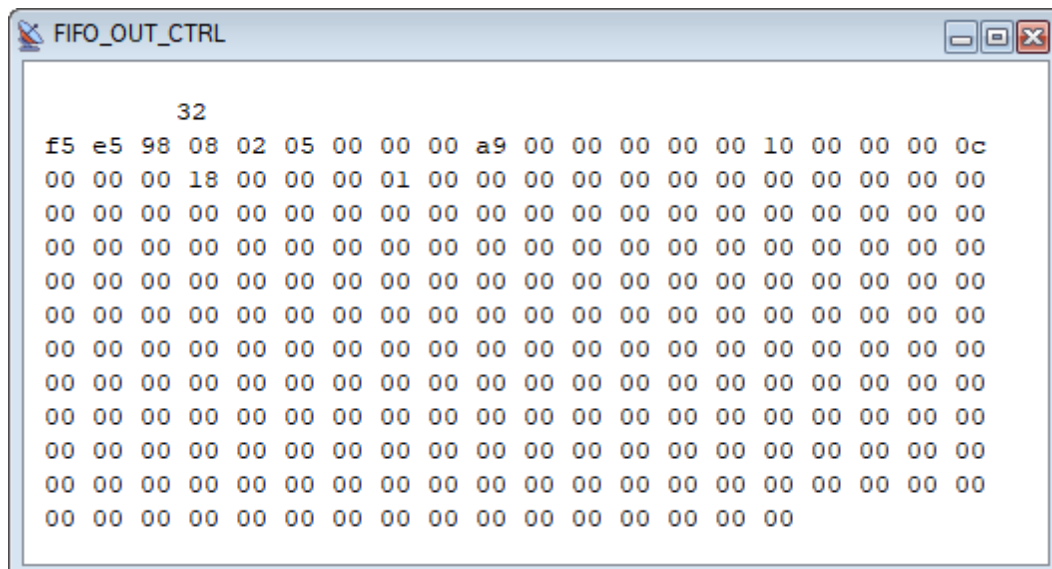
Displays the status of the incoming TLM and STATUS FIFOs. The screen shows the number of incoming FIFO_IN_TLM and FIFO_IN_STATUS blocks, the number of incoming bytes and the number of generated FTF_IN_TLM and FTF_IN_STATUS blocks. If any errors occur during the decoding they will be listed here as well. If communication with the SCE gets lost the status might remain in the synchronized state. Although it is the correct state indication it might be misleading since there is no communication with the SCE. In this case a stale indication will notify you of the SCE data being stale.

FIFO_OUT_CMD



CMD FIFO outgoing data. Similar to the two above screens, just for the outgoing command data to the CMD FIFO.

FIFO_OUT_CTRL

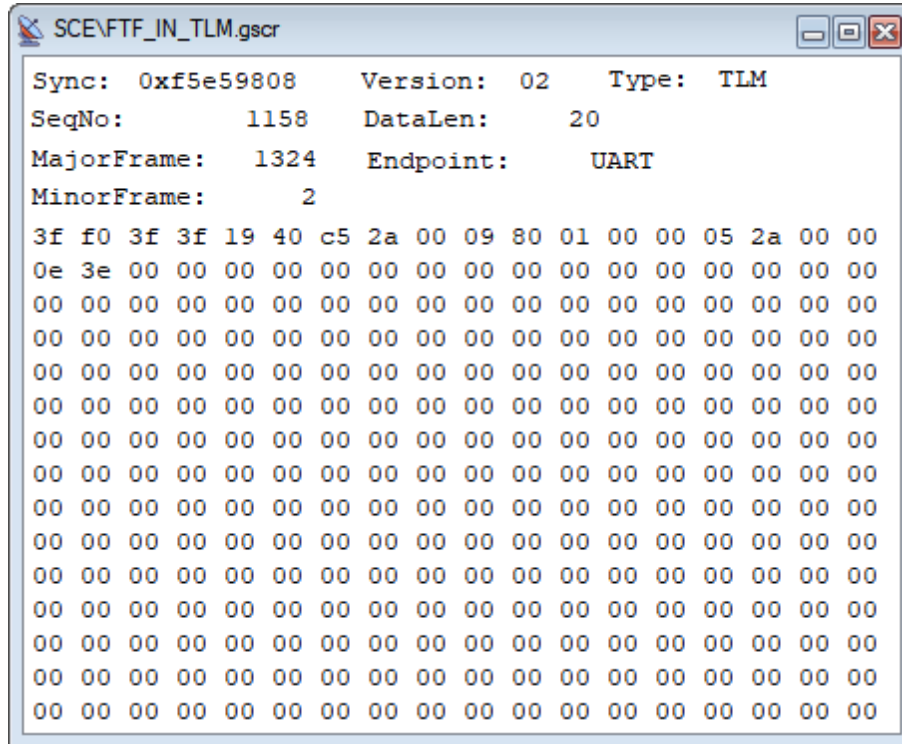


CTRL FIFO outgoing data. Same as above, just for CTRL FIFO.

1.6.2 Incoming Data

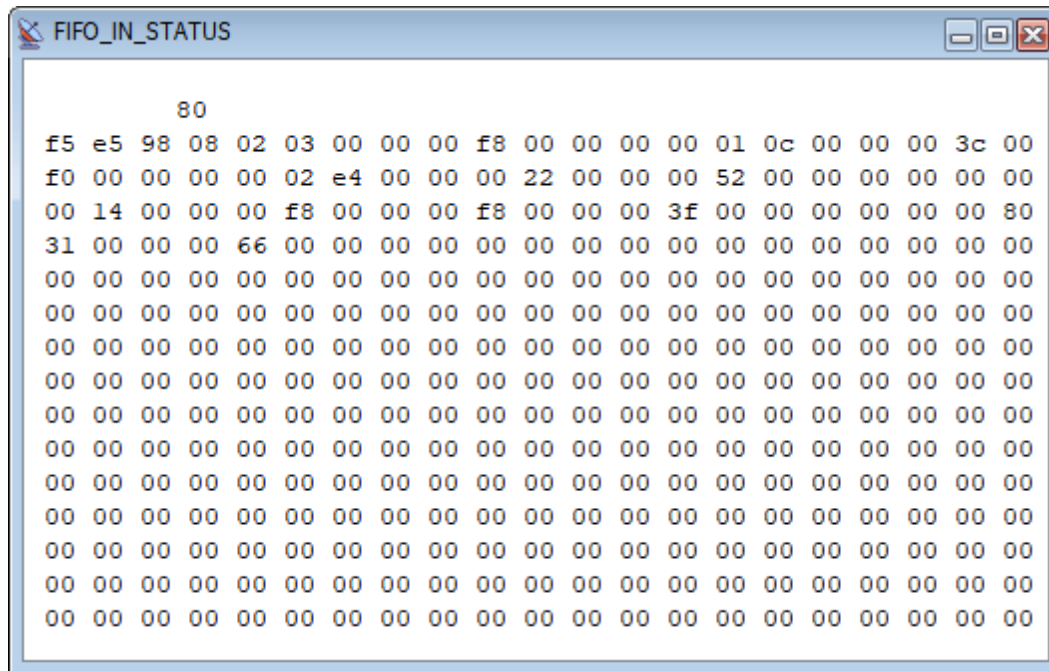
The incoming data screens display the various SCE data types. The chapter on [TLM Processing](#) shows how the blocks are decoded within the system.

FTF_IN_TLM



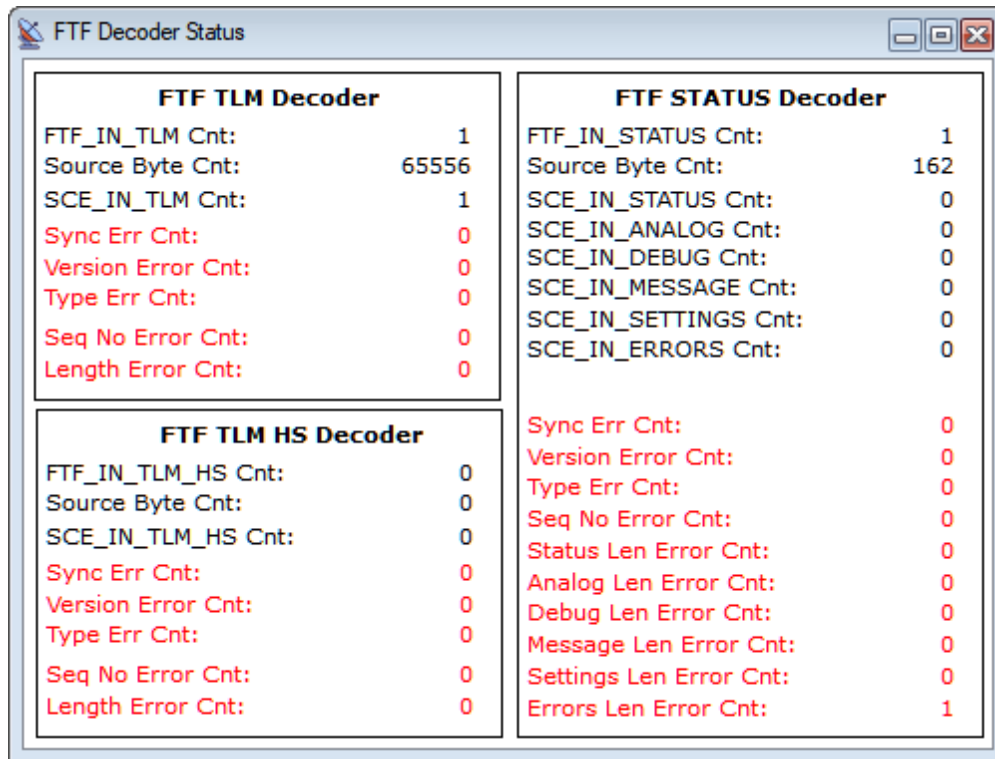
FTF frame with TLM data. This block has the FTF header information attached like the major and minor frame numbers, sequence number, etc. The data payload is displayed in the data portion.

FTF_IN_STATUS



Similar to the FTF_IN_TLM, just for STATUS data. For STATUS data there is a one-to-one correlation of FTF packet and payload. The FTF_IN_STATUS block is further decoded into the various FTF types like STATUS, ANALOG, MESSAGE, DEBUG.

FTF Decoder Status



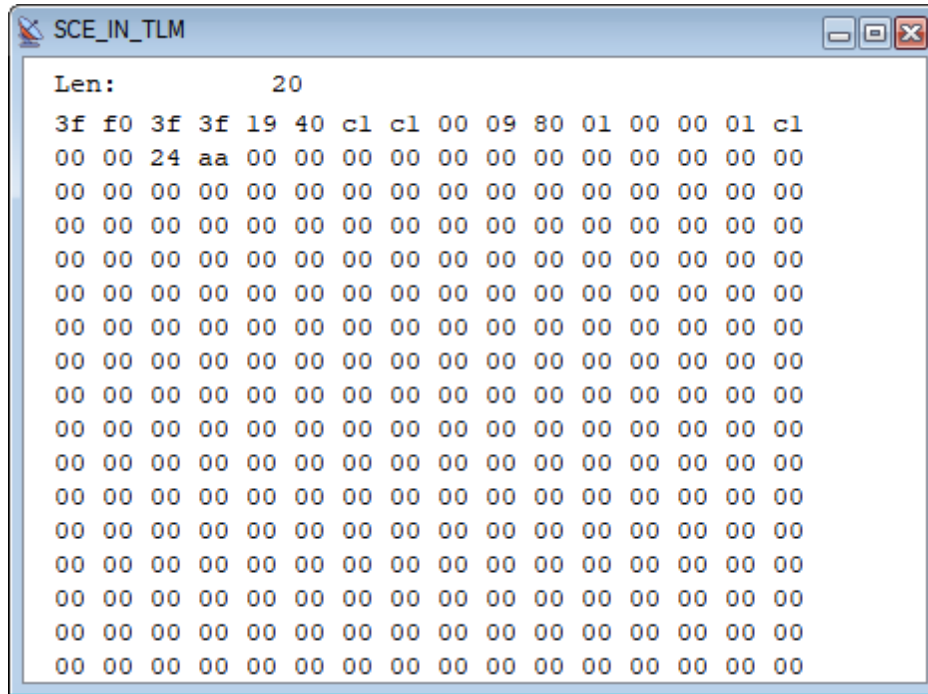
FTF TLM Decoder	
FTF_IN_TLM Cnt:	1
Source Byte Cnt:	65556
SCE_IN_TLM Cnt:	1
Sync Err Cnt:	0
Version Error Cnt:	0
Type Err Cnt:	0
Seq No Error Cnt:	0
Length Error Cnt:	0

FTF TLM HS Decoder	
FTF_IN_TLM_HS Cnt:	0
Source Byte Cnt:	0
SCE_IN_TLM_HS Cnt:	0
Sync Err Cnt:	0
Version Error Cnt:	0
Type Err Cnt:	0
Seq No Error Cnt:	0
Length Error Cnt:	0

FTF STATUS Decoder	
FTF_IN_STATUS Cnt:	1
Source Byte Cnt:	162
SCE_IN_STATUS Cnt:	0
SCE_IN_ANALOG Cnt:	0
SCE_IN_DEBUG Cnt:	0
SCE_IN_MESSAGE Cnt:	0
SCE_IN_SETTINGS Cnt:	0
SCE_IN_ERRORS Cnt:	0
Sync Err Cnt:	0
Version Error Cnt:	0
Type Err Cnt:	0
Seq No Error Cnt:	0
Status Len Error Cnt:	0
Analog Len Error Cnt:	0
Debug Len Error Cnt:	0
Message Len Error Cnt:	0
Settings Len Error Cnt:	0
Errors Len Error Cnt:	1

The FTF Decoder Status screen displays the current FTF TLM and FTF TLM_HS Decoder status. It indicates when the decoder is synchronized/unsynchronized, the number of source bytes received and the number of incoming FTF_IN_TLM, FTF_IN_STATUS blocks. It also displays any error counters. This can be used to debug the FTF protocol.

SCE_IN_TLM



This screen shows the raw incoming TLM bytes. The data is not synchronized and is further decoded by the TLM ITF decoder into ITF frames and consecutively into CCSDS packets.

SCE_IN_STATUS

SCE STATUS

Reason Code: TIME_PULSE

FIFO CMD		FIFO CONTROL	
Not Synchronized		Not Synchronized	
Frame Cnt:	0	Frame Cnt:	0
Sync Err Cnt:	0	Sync Err Cnt:	0
Length Error Cnt:	0	Length Error Cnt:	0
Version Error Cnt:	0	Version Error Cnt:	0
Type Error Cnt:	0	Type Error Cnt:	0
Seq No Error Cnt:	0	Seq No Error Cnt:	0

FIFO TLM		FIFO STATUS	
Frame Cnt:	0	Frame Cnt:	0
Overrun Err Cnt:	0	Overrun Err Cnt:	0

Major Frame Status		Power Switches	
Current Major:	0	Instr. A:	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Top Major:	0	Instr. B:	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Delay:	0 ms	FP Instr. Pwr:	<input checked="" type="checkbox"/>
Underflow Error Cnt:	0	FP Aux Pwr:	<input checked="" type="checkbox"/>
Skip Error Cnt:	0	Aux 0:	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
		Aux 1:	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
		Aux 2:	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
		Aux 3:	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
		Aux 4:	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
		Aux 5:	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
		Aux 6:	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
		Aux 7:	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>

Test Data	
1PPS:	Disabled
Loopback:	Disabled
Test Data Gen. (LS):	Disabled
Test Data Gen. (HS):	Disabled

TLM UART		Fault Flags	
UART Side	A	Global:	Ok
Parity Error Cnt:	0	Error This Frame:	Ok
Frame Error Cnt:	0	FTF Tx:	Ok
Overrun Error Cnt:	0	Volt/Curr:	Ok
		TLM UART:	Ok
		TLM Timing:	Ok
		CMD Timing:	Ok

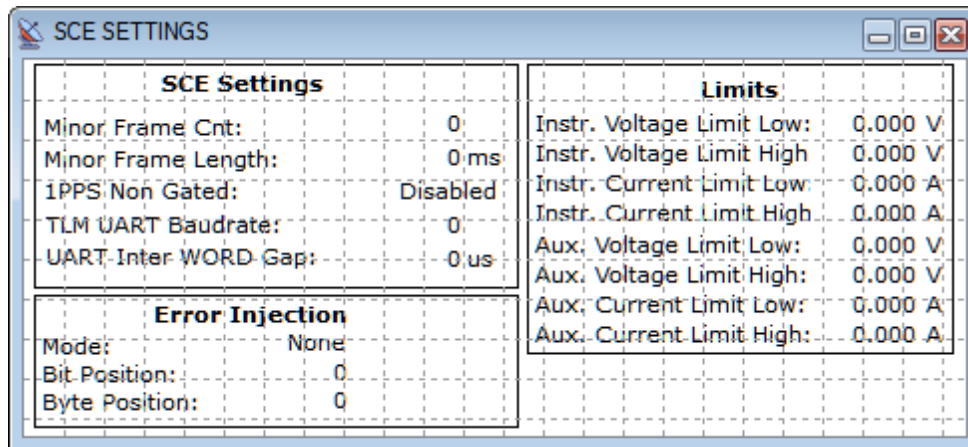
Max Poll Time:	
Max Poll Time:	0 us
TLM Buffer Free:	0

5
4
3
2
1
0

The SCE Status screen displays the information GSEOS receives from the SCE in the status message. It displays the current synchronization status and error counters for error conditions the SCE has detected. It also displays some of the currently configured settings in the lower left pane. The upper section displays the FTF frames received/sent by the SCE. The Major Frame Status section shows the currently processing major frame, the round trip delay and any errors encountered. The underflow error indicates that the

command data was not written in time from GSEOS to the SCE. The Skip Error counter indicates that the SCE observed a skip in the major frame number coming from GSEOS. The SCE_IN_STATUS block is generated whenever the 1PPS pulse is set by the SCE and triggers the processing and sending of the next major frame worth of command data to the SCE. The power switch status is displayed as well.

SCE_IN_SETTINGS



The screenshot shows a window titled "SCE SETTINGS" with three main sections: SCE Settings, Error Injection, and Limits. The SCE Settings section includes fields for Minor Frame Cnt (0), Minor Frame Length (0 ms), 1PPS Non Gated (Disabled), TLM UART Baudrate (0), and UART Inter WORD Gap (0 us). The Error Injection section includes Mode (None), Bit Position (0), and Byte Position (0). The Limits section includes fields for Instr. Voltage Limit Low/High (0.000 V) and Instr. Current Limit Low/High (0.000 A), as well as Aux. Voltage Limit Low/High (0.000 V) and Aux. Current Limit Low/High (0.000 A).

SCE Settings	
Minor Frame Cnt:	0
Minor Frame Length:	0 ms
1PPS Non Gated:	Disabled
TLM UART Baudrate:	0
UART Inter WORD Gap:	0 us

Error Injection	
Mode:	None
Bit Position:	0
Byte Position:	0

Limits	
Instr. Voltage Limit Low:	0.000 V
Instr. Voltage Limit High:	0.000 V
Instr. Current Limit Low:	0.000 A
Instr. Current Limit High:	0.000 A
Aux. Voltage Limit Low:	0.000 V
Aux. Voltage Limit High:	0.000 V
Aux. Current Limit Low:	0.000 A
Aux. Current Limit High:	0.000 A

The SCE_IN_SETTINGS screen shows the contents of the SCE_IN_SETTINGS block. Besides the startup settings that are [configured](#) with the gseos.ini configuration file it also shows any updates to the settings due to commands like voltage and current limits. This block gets updated whenever the SCE encounters new settings or upon request (Command: SCE_CTRL_REQUEST_SETTINGS()).

SCE_IN_ERRORS

The screenshot shows a window titled "SCE ERRORS" with a grid of error counters. All values are currently 0.

Reason Code:	
FIFO CMD	FIFO CONTROL
Sync Err Cnt: 0	Sync Err Cnt: 0
Length Error Cnt: 0	Length Error Cnt: 0
Version Error Cnt: 0	Version Error Cnt: 0
Type Error Cnt: 0	Type Error Cnt: 0
Seq No Error Cnt: 0	Seq No Error Cnt: 0
FIFO TLM	FIFO STATUS
Overrun Err Cnt: 0	Overrun Err Cnt: 0
Major Frame Errors	Voltage/Current Errors
No Major Recvd: 0	Instr. Voltage Err Cnt: 0
Skip Error Cnt: 0	Instr. Current Err Cnt: 0
Cmds Dropped Err Cnt: 0	Aux. Voltage Err Cnt: 0
	Aux. Current Err Cnt: 0
TLM UART Errors	Time Pulse Errors
UART Parity Error Cnt: 0	CMD Overrun Err Cnt: 0
UART Frame Error Cnt: 0	CMD Underrun Err Cnt: 0
UART Overrun Error Cnt: 0	TLM Overrun Err Cnt: 0
Pass. Side Activity Err. Cnt: 0	TLM Underrun Err Cnt: 0

The SCE_IN_ERRORS screen displays all the errors and error counters that are reported by the SCE. This block gets updated whenever the SCE encounters a new error or upon request (Command: SCE_CTRL_REQUEST_ERRORS()).

Following a brief description of the various error counters:

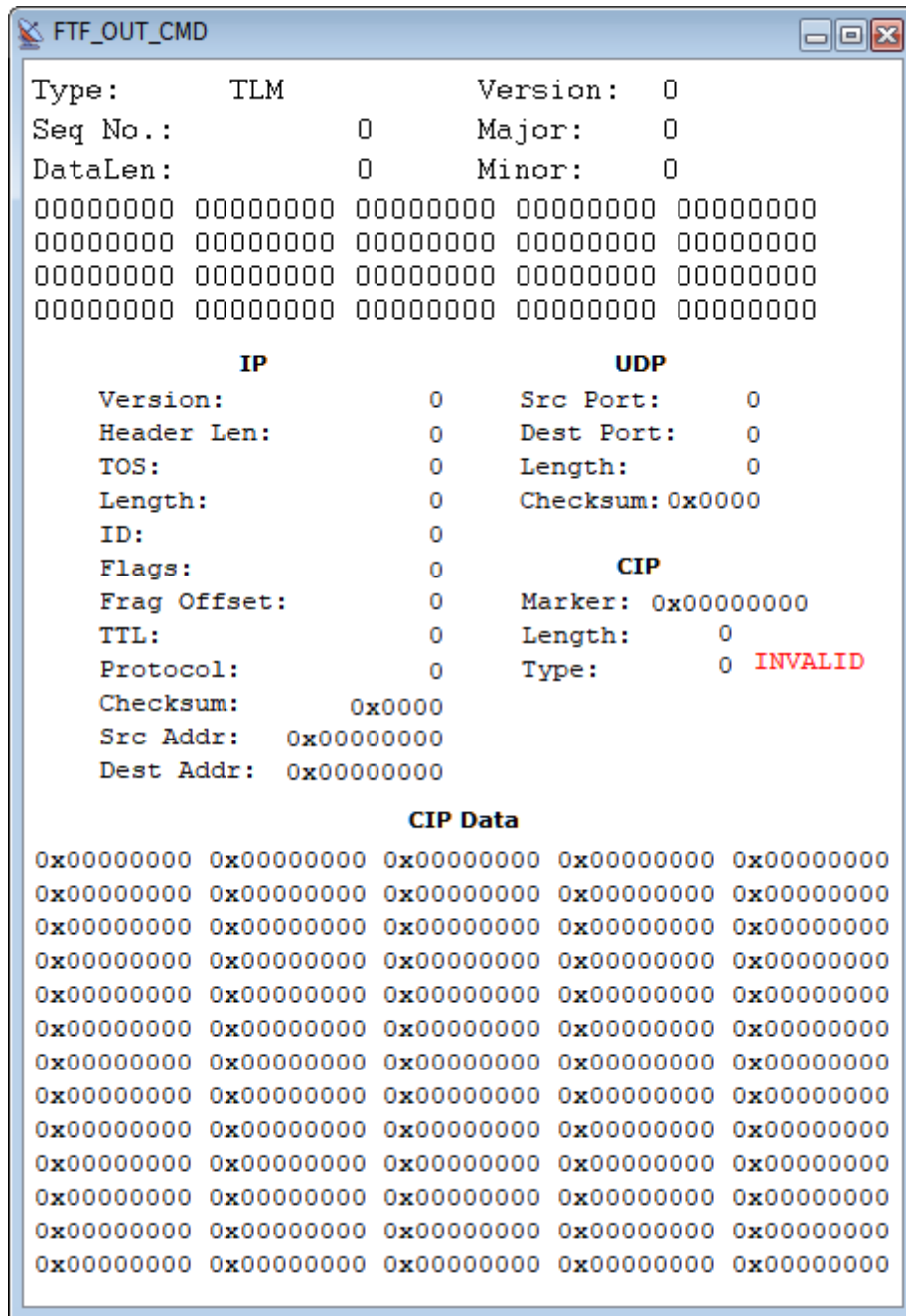
Name	Description
Sync Error	Synchronization error on the CMD or CTRL FIFO.
Length Error	FTF length error
Version Error	FTF version error
Type Error	FTF type error
Seq No Error	FTF sequence number error
Overrun Error	The TLM or STATUS FIFO ran out of space and data was lost
CMD Overrun Error	The SCE received too many commands to clock out in the requested major frame, one or more have been dropped.
CMD Underrun Error	The SCE ran out of major frame data. This usually happens when the machine is heavily loaded or GSEOS doesn't get a chance to process in a timely manner due to a slow running script.
TLM Overrun Error	The SCE FIFO is full and GSEOS did not read the data so the SCE dropped some data.
TLM Underrun Error	No TLM data was detected during a major frame.

Instr. Voltage Error	The instrument voltage exceeded the high or low limits.
Instr. Current Error	The instrument current exceeded the high or low limits.
Aux. Voltage Error	The aux. voltage exceeded the high or low limits.
Aux. Current Error	The aux. current exceeded the high or low limits.
UART Parity Error	A parity error was detected on the CMD UART.
UART Frame Error	A frame error was detected on the CMD UART.
UART Overrun Error	An overrun error was detected on the CMD UART.

1.6.3 Outgoing Data

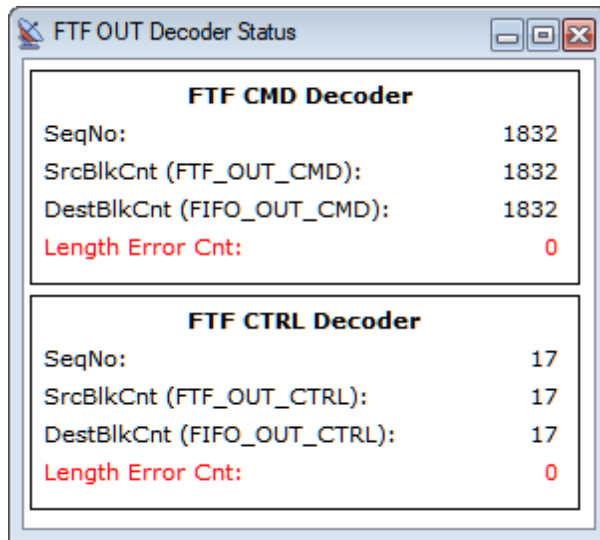
The outgoing data screens display any data going from GSEOS to the SCE. The [Command Processing](#) shows how the blocks are decoded within the system.

FTF_OUT_CMD



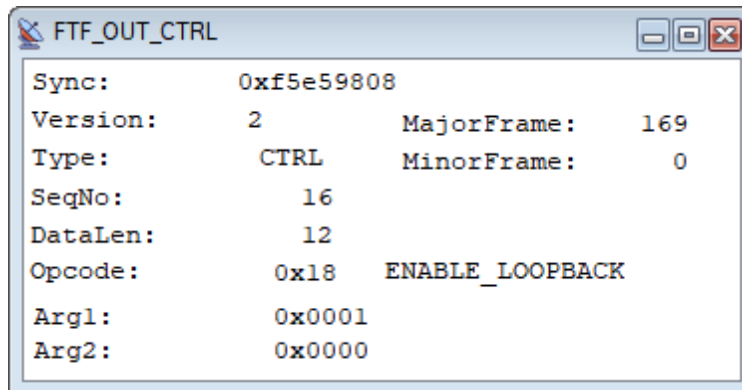
The FTF_OUT_CMD block gets decoded into FIFO_OUT_CMD blocks and finally transferred to the SCE. All outgoing command data is eventually decoded into this block. It contains the FTF header and data sections. Since there is exactly one command transaction (IP/UDP) per FTF the IP/UDP headers and data are also displayed on this screen.

FTF OUT Decoder Status



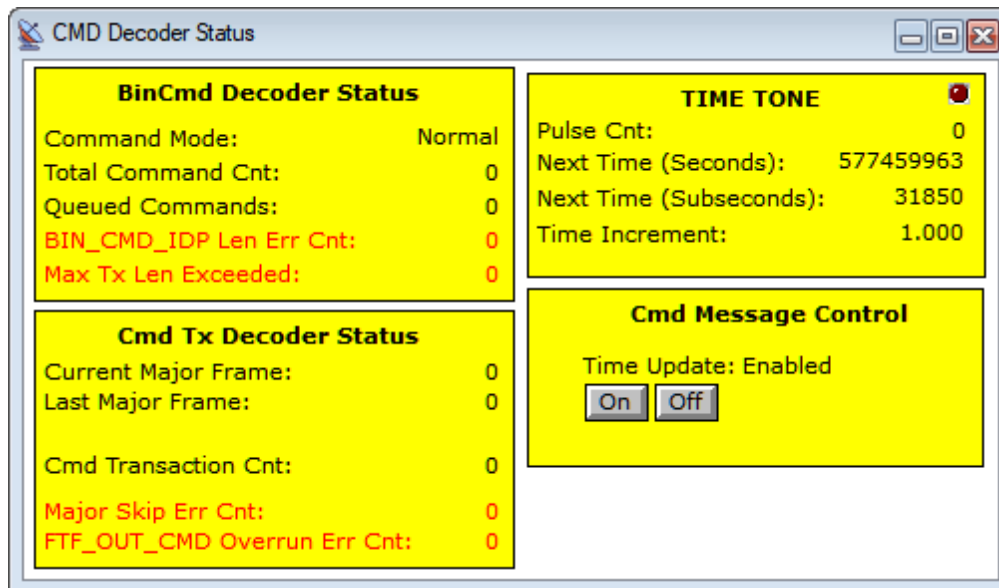
The FTF OUT Decoder Status screen displays the current outgoing FTF command decoder status like the current sequence number, the count of source blocks received as well as error counters. It shows both the outgoing CMD FTF and CTRL FTF decoding status.

FTF_OUT_CTRL



The FTF_OUT_CTRL screen shows the contents of the FTF_OUT_CTRL block. This block is generated when the user issues a command that is directed to the SCE. These are mostly configuration commands sent by GSEOS on SCE initialization and user directives.

L'LORRI CMD Decoder Status



The L'LORRI Cmd Decoder Status screen shows the status of the L'LORRI command decoder. There are two decoders involved in the command decoding (see the [Command Processing](#) section for more details). The top section shows the BinCmd Decoder which puts commands onto the command queue, the lower left section shows the status of the Cmd Queue Decoder which takes commands off the queue, bundles them into command transactions and transmits them to the SCE. On the right there are settings for the Time Update message.

The Total Cmd Cnt entry lists the total number of commands issued and the Queued Cmd Cnt lists the number of commands currently on the command queue awaiting transfer to the SCE.

If a command is larger then the BIN_CMD_IDP block the BIN_CMD_IDP Len Err Cnt

The right part of this screen shows the current time as delivered in the Time Update message. Every time GSEOS receives a SCE STATUS MESSAGE indicating the assertion of the 1PPS pulse the pulse counter is incremented. It is possible to enable and disable the TIME message at run time. However, since the system maintains a command queue and the insertion of the TIME message has an effect on the maximum size of command data available in a transaction the enable/disable request is queued together with the commands (synchronous). The status is marked as pending if a request is on the queue, it will change to Enabled or Disabled as soon as the setting takes effect.

The FTF_IN_TLM block displays the FTF frame including header information and the included MSG/CCSDS packet. There is a one-to-one relation between the FTF_IN_TLM and a CCSDS packet, that is one FTF_IN_TLM block contains exactly one packet.

FTF Decoder Status

FTF TLM Decoder		FTF STATUS Decoder	
FTF_IN_TLM Cnt:	1	FTF_IN_STATUS Cnt:	1
Source Byte Cnt:	65556	Source Byte Cnt:	162
SCE_IN_TLM Cnt:	1	SCE_IN_STATUS Cnt:	0
Sync Err Cnt:	0	SCE_IN_ANALOG Cnt:	0
Version Error Cnt:	0	SCE_IN_DEBUG Cnt:	0
Type Err Cnt:	0	SCE_IN_MESSAGE Cnt:	0
Seq No Error Cnt:	0	SCE_IN_SETTINGS Cnt:	0
Length Error Cnt:	0	SCE_IN_ERRORS Cnt:	0
FTF TLM HS Decoder		Sync Err Cnt:	0
FTF_IN_TLM_HS Cnt:	0	Version Error Cnt:	0
Source Byte Cnt:	0	Type Err Cnt:	0
SCE_IN_TLM_HS Cnt:	0	Seq No Error Cnt:	0
Sync Err Cnt:	0	Status Len Error Cnt:	0
Version Error Cnt:	0	Analog Len Error Cnt:	0
Type Err Cnt:	0	Debug Len Error Cnt:	0
Seq No Error Cnt:	0	Message Len Error Cnt:	0
Length Error Cnt:	0	Settings Len Error Cnt:	0
		Errors Len Error Cnt:	1

The FTF Decoder Status screen shows details about the decoding of FTF frames, this should give you a good overview what data is received from the SCE.

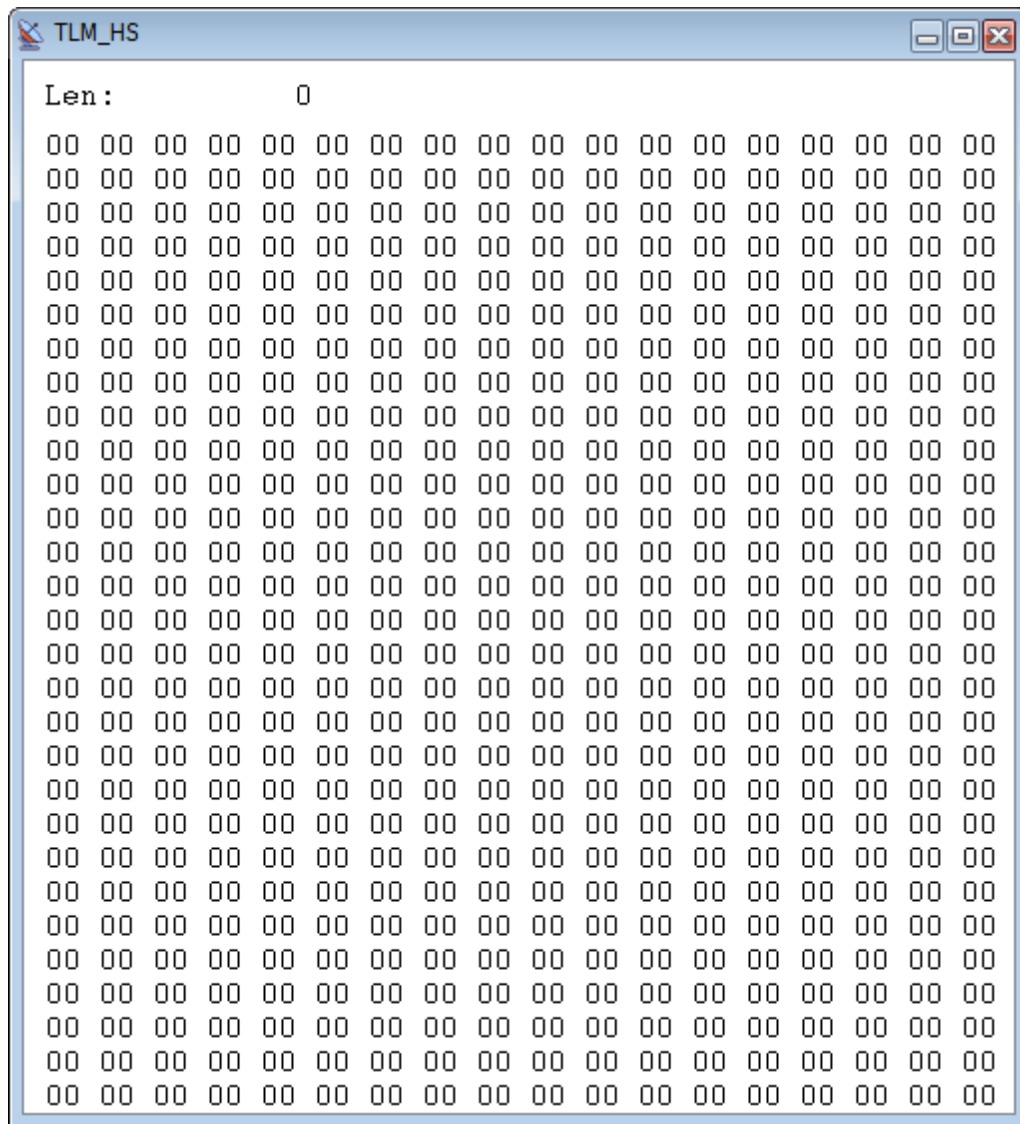
Telemetry decoding takes the SBC_IN_TLM block and decodes it into the high and low speed SBC_IN_TLM_LS and SBC_IN_TLM_HS blocks. Besides these two output blocks the decoder status is exported in the TLM_DecStatus block. The images below depict the individual screens.

SCE_IN_TLM_HS



The SCE_IN_TLM_HS block contains a stream of unsynchronized data. It is the instrument teams responsibility to decode the data according to their requirements. The amount of valid bytes is indicated in the Len field of the block. This screen simply displays the hex data of the block. This block is copied into the TLM_HS block which is the basis for all instrument specific high speed data decoding.

TLM_HS



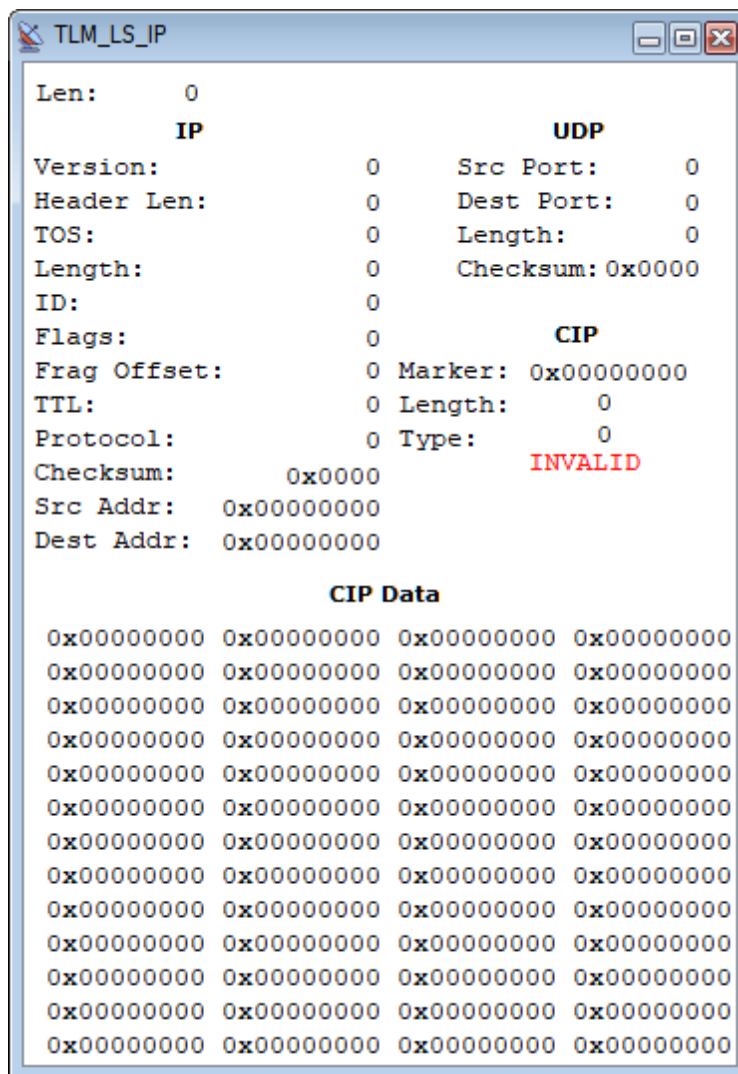
The TLM_HS block contains the high speed data as delivered from the instrument. This block is the basis for instrument specific decoding of the high speed telemetry.

SCE_IN_TLM_LS



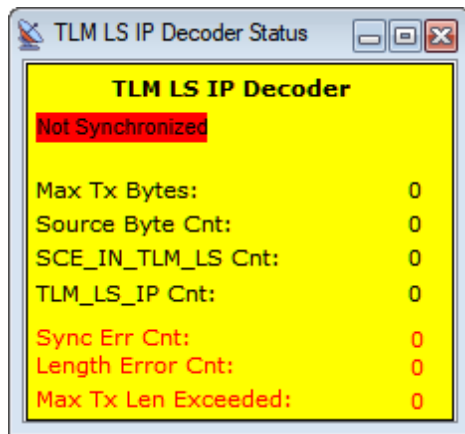
The SCE_IN_TLM_LS block contains the unsynchronized low speed telemetry data. The IP decoder synchronizes on IP headers and generates properly formatted IP packets in the TLM_LS_IP block.

TLM_LS_IP



The SCE_IN_TLM_LS block contains the LUCY protocol stack. This includes the IP/UDP/CIP protocols. The headers of the individual protocols are broken out and can be examined in this screen. Keep in mind that there can be multiple CIP packets per IP/UDP transaction. Only the first CIP header and data is displayed here. The CIP decoder will process this block and generate the according number of TLM_LS_IDP blocks (the CIP payload).

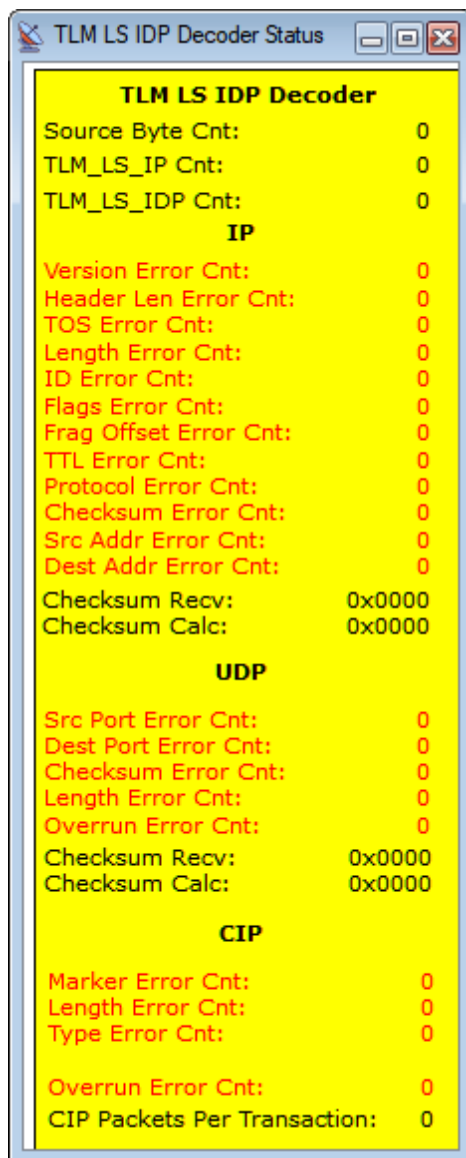
TLM LS IP Decoder Status



The TLM LS IP Decoder Status reports the status of the IP decoder. It shows the number of source bytes received as well as the number of incoming and generated blocks. If there is a length error in the incoming block the length error counter will be incremented accordingly. If we fall out of synchronization the Sync Error counter is incremented.

The next decoder level is the TLM LS IDP decoder.

TLM LS IDP Decoder Status



The TLM LS IDP Decoder Status reports the status of the IDP decoder. It shows the number of source bytes received as well as the number of incoming and generated blocks. If there is a length error in the incoming block the length error counter will be incremented accordingly. The TLM LS IDP Decoder Status screen reports any errors encountered in the IP/UDP/CIP protocol layers as well as the currently computed and received checksums of these packets.

The screen is divided into four sections. The top section lists the incoming and outgoing block counters. The second section shows the IP decoding errors as well as the received and calculated checksums. The next section does the same for the UDP protocol layer. The last section lists the error counters and the number of CIP packets for the current transaction. You can use this status screen to determine the proper implementation of the according protocol levels in the flight software.

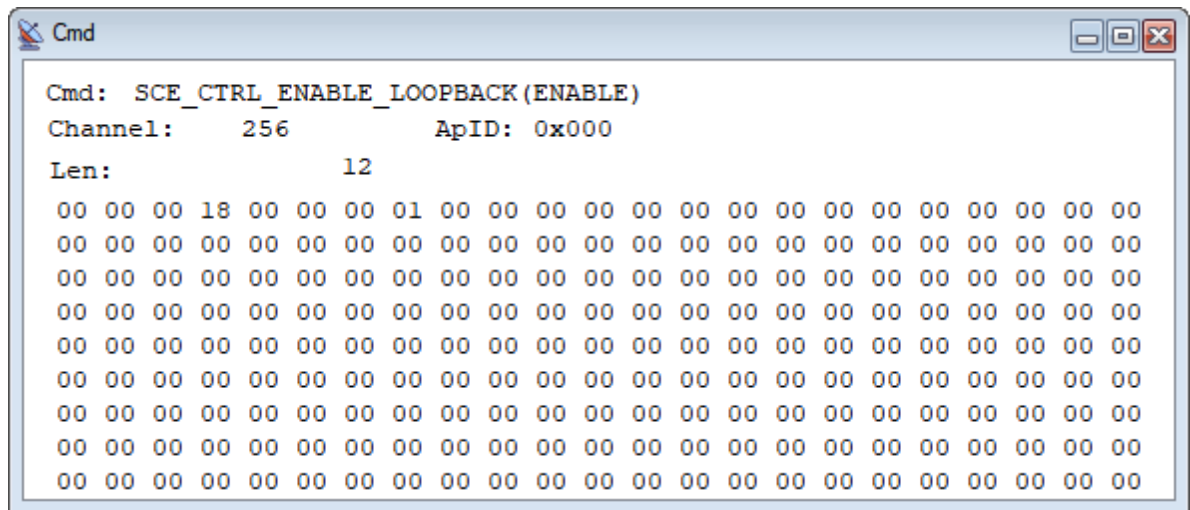
The IDP decoder decodes the synchronized TLM low speed data (IP packets) and generates CIP packets. One TLM_LS_IDP block is generated per CIP packet contained

TLM_LS_IDP



The command screens show any data related to commanding. The BinCmd screen shows all binary GSEOS commands.

© 1998-2018 GSE Software, Inc.



The BinCmd block is the general GSEOS binary command mechanism which is used for all kinds of commanding. Commands to the SCE and to the instrument are issued on separate command channels. The BIN_CMD_IDP and SCEBinCmd Decoders decode the BinCmd block depending on the command channel into FTF_OUT_CMD and FTF_OUT_CTRL blocks.

[illegible]

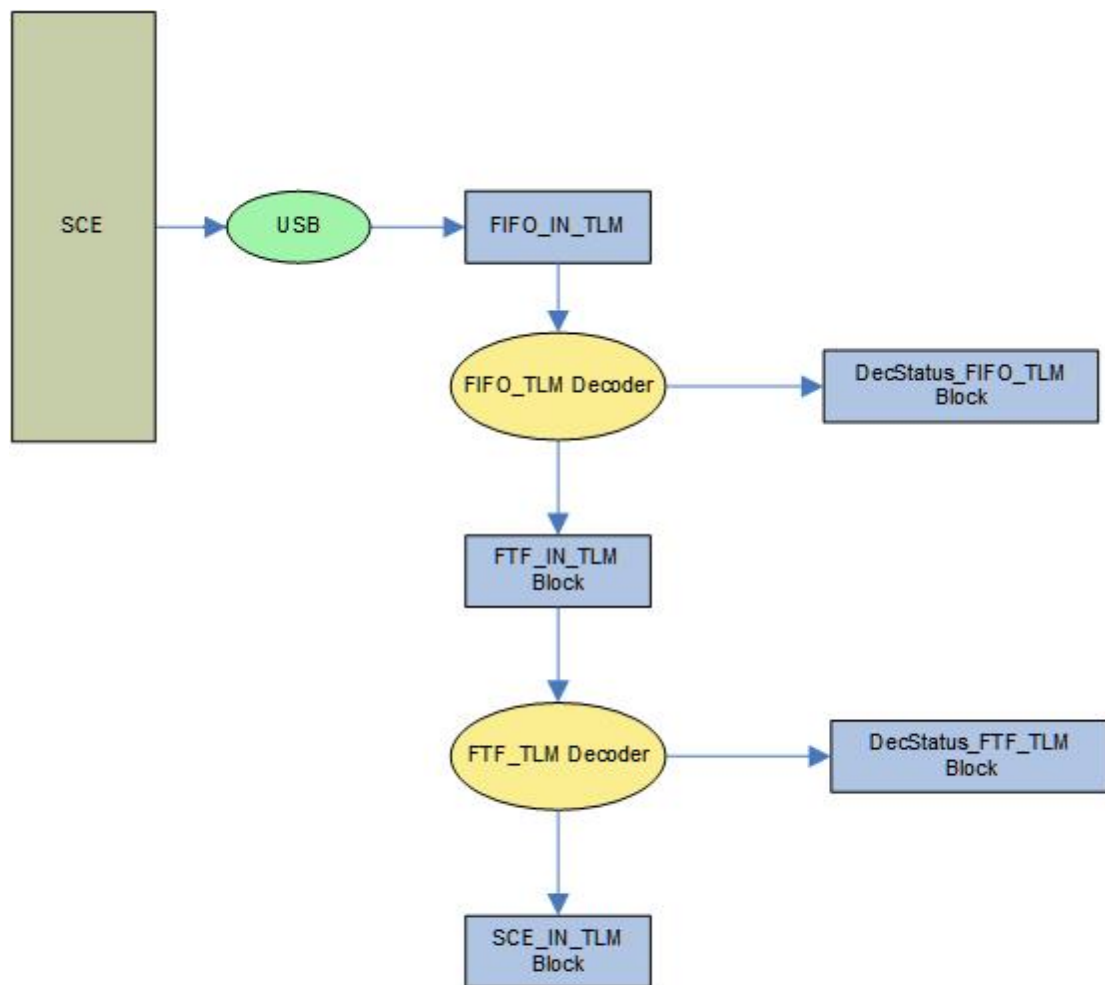
© 1998-2018 GSE Software, Inc.

1.7 TLM Processing

The SCE has several serial RS-422 interfaces to the instrument. The SCE delivers data from the high speed ports over the FIFO_IN_TLM_HS block (raw unsynchronized data) and data from the low speed ports with the FIFO_IN_TLM_LS. The FIFO_TLM_Decoder (there is one for LS and one for HS) decoder generates the blocks SCE_IN_TLM_HS and SCE_IN_TLM_LS respectively from these two data streams. These contain complete FTF frames as delivered from the SCE. There is a decoder status block DecStatus_FIFO_TLM (DecStatus_FIFO_TLM_HS) that indicates any decoding errors and provides status counters.

The following decoder state (FTF_TLM_Decoder) checks for validity of the FTF header and dispatches according to FTF type. The HS data ends up in SCE_IN_TLM_HS, the LS data in SCE_IN_TLM_LS. Since the HS data is still TBD it is simply forwarded into the TLM_HS block without further decoding.

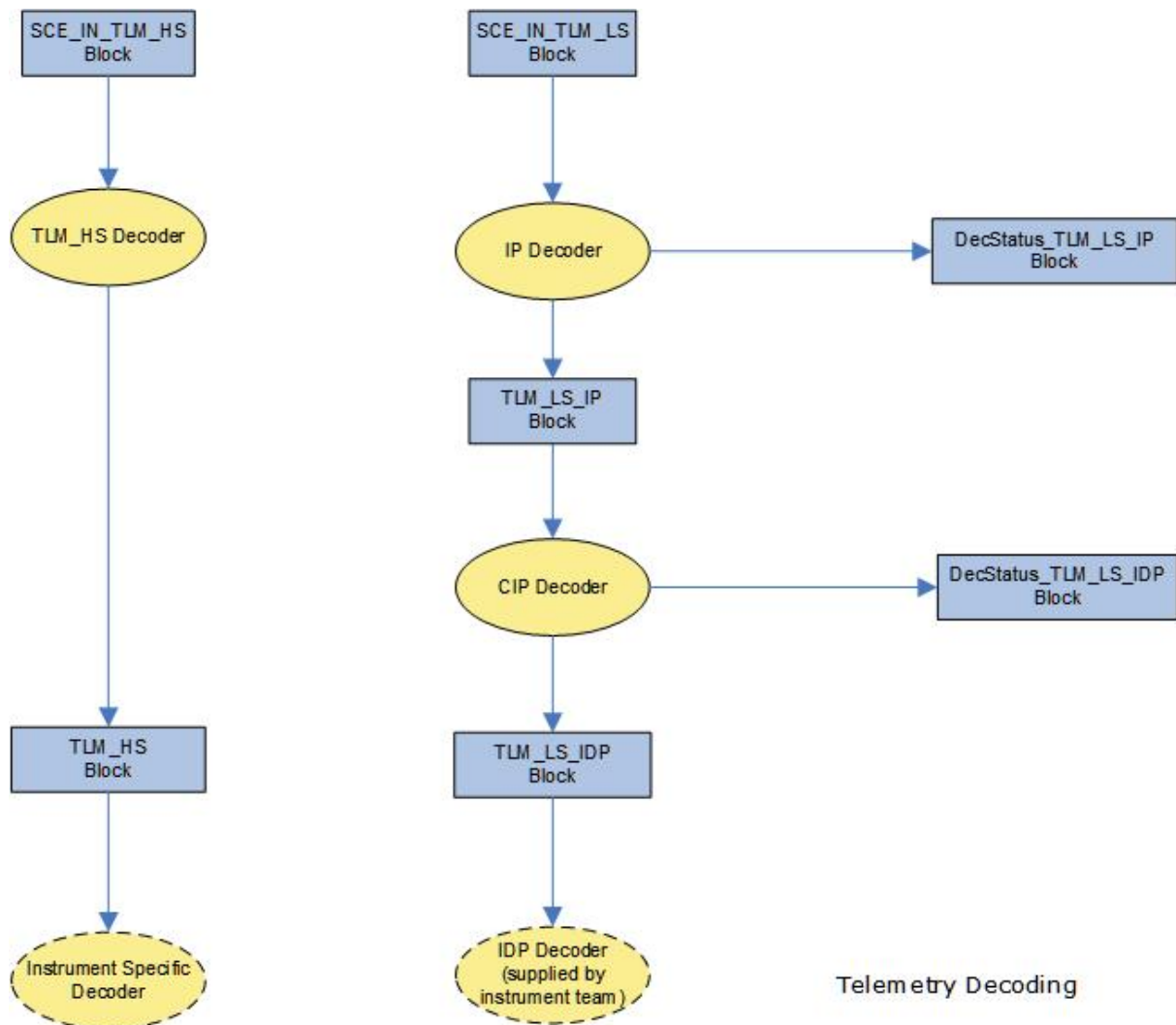
Below is the block/decoder topology of the first stage of decoding. For simplicity we leave the HS/LS designation out, there are two parallel paths through the system, one for HS, one for LS.



SCE -> GSEOS Incoming TLM Data

Low speed data is further decoded according to the LUCY protocol stack: IP/UDP/CIP. The IDP protocol is up to the instrument team to decode.

The IP decoder will synchronize on the IP header and forward a complete IP packet in the TLM_LS_IP block that contains the complete IP/UDP/CIP protocol stack. The CIP decoder will process these three protocol headers and report any status and error information detected during the decoding process. The final output will be the payload of the CIP protocol and will be provided in the TLM_LS_IDP block. One IP/UDP packet can contain a multitude of CIP packets, so the decoder can potentially generate several TLM_LS_IDP blocks per decoded TLM_LS_IP block. On top of the CIP protocol is the IDP protocol layer which is instrument specific. The instrument teams will use the TLM_LS_IDP block and apply their own decoding according to their specific protocol. The diagram below shows the data flow of the telemetry data through the different GSEOS blocks and decoders.



1.8 Command Configuration

Instrument Commands

L'LORRI defines different kinds of commands: Instrument commands and the Time Update message are both considered commands for this discussion.

The rest of this chapter will mostly deal with instrument commands.

Defining Commands

Before a command can be sent from GSEOS it has to be defined. The GSEOS User Manual outlines the command definition syntax. See below for a brief example of a simple command definition:

```
<GSEOS Version="366">
  <Cmd Mnemonic="TOF_ANODE_A_THR" Opcode="0xa0" NumBits="8"
  Description="Set the Anode A threshold">
```

```

    <Arg NumBits="8" DataRangeLow="0" DataRangeHigh="63"
Description="Threshold"/>
  </Cmd>
</GSEOS>

```

The above XML script defines a single command "TOF_ANODE_A_THR". The command definition defines the opcode, arguments, argument types, and a command description. Commands are stored in a file with the extension .cpd (Command Processor Definition) and loaded into GSEOS.

When GSEOS executes the commands it will compose all command parts (opcode, arguments, constants) into a binary command string.

Assigning Commands to GUI elements

Once your commands are defined you can assign them to GSEOS user interface elements like command buttons or menus. When you issue a command you can supply any arguments if not specified in the user interface. To place a command on a command menu you add it to a command menu file. The example below shows our command assigned to a command menu. Commands menus are defined in .cm (Command Menu) files. Please refer to the GSEOS documentation for more detailed information about command menus and command buttons.

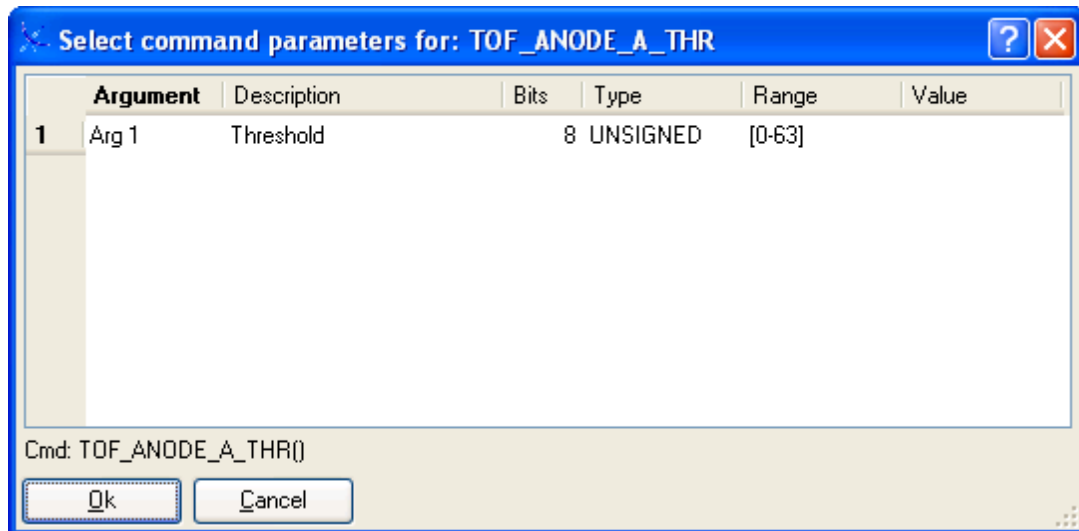
```

Menu &MyCommands
{
    Menuitem &ANODE A THR..., TOF_ANODE_A_THR()
}

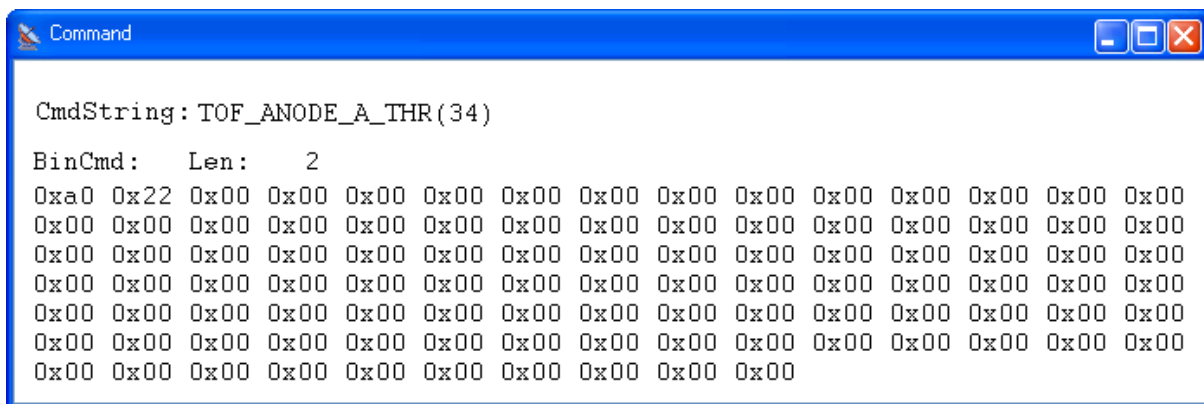
```

Command Flow

When you issue your just defined command from the command menu the following command dialog will pop up to query for the command argument.



When you fill in the argument value and issue the command GSEOS will process the command string according to your command definition and generate a CmdString block with the command text and a BinCmd block with the binary representation of your command data. You can verify the command data with the Cmd.gscr screen.



The BinCmd command decoder will decode the BinCmd block into an BIN_CMD_IDP block. Currently this decoder is a simple copy function. If an instrument team requires dynamic processing of the binary data and to convert it into their specific IDP format they can replace the generic BinCmd decoder with their own version. The BIN_CMD_IDP block contains the command data formatted as required by the IDP specification. The BIN_CMD_IDP block is then put onto the GSEOS command queue for further processing. See the chapter on [Command Customization](#) for more details.

Command Timing

LUCY L'LORRI commands are transmitted from the S/C to the instrument in bundles called transactions. A transaction is one IP packet containing exactly one UDP packet, which in turn contains one or more CIP packets. Each CIP packet in turn represents one command (instrument command or other).

Each transaction can be scheduled to be transmitted to the instrument at a particular point relative to the TIME TIC pulse.

Nominally there are two command transactions per 1PPS (TIME TIC) period. The first one is scheduled at 100ms after the TIME ONE, the second at 500ms. Each transaction allows for a maximum of 20 commands (including Time message) and has a maximum size of 2048 bytes.

If you don't specify any Command Transaction Timing this will be the default timing.

However, you can customize this with the [CmdTransactions] section.

For testing purposes it is useful to control the timing and grouping of commands into transactions. GSEOS allows you to specify one or more command transactions and lets you determine the time when the transaction is clocked out to the instrument as well as command and transaction size limits. To specify the command timing you define command transactions. Each command transaction has an execution time relative to the TIME TIC (1PPS) pulse and a maximum number of commands that can be grouped into this transaction. The time of the transaction is specified with the Time entry and specifies the offset to the TIME TIC in ms. You specify the maximum number of commands for the transaction with the MaxCmds entry. If you also schedule a Time Update Message this will count against the number of MaxCmds in this transaction. A transaction can contain one or more instrument commands. If the maximum number of commands is reached for a particular transaction and there are still commands on the command queue they will be

added to the next transaction. Keep in mind that the Time Update message takes up one slot if you schedule this message.

Besides limiting the transaction by the number of commands you can also limit the transaction size by the number of bytes. The MaxBytes entry determines how many bytes the entire transaction can have. The maximum size for the MaxBytes value is limited by the size of the FTF_OUT_CMD block. If you need more than roughly 32KB you will need to reconfigure this accordingly. If you need to transfer more command data per 1PPS period you can define multiple transactions with each one of them having a limit that meets the 32KB limit. Keep in mind that you can't violate the bandwidth limitation of the UART. If more data is transmitted to the SCE than the SCE can physically clock out to the instrument given the interface configuration ([UART configuration](#)) the SCE will drop commands and set an according error counter.

A transaction will never exceed the maximum number of bytes configured (or the default if not specified). If you place a command on the queue that violates this restriction the command will be rejected. Say you define only one command transaction with a maximum number of bytes of 200 you won't be able to issue commands that take up more than 200 bytes minus the IP/UDP/CIP/IDP headers required!

However, if you configure multiple transactions you will be able to place a command as large as will pass through the largest configured transaction.

Besides instrument commands a transaction can also contain a Time Update message. Nominally the Time Update message is transferred as the first message of the first transaction. However, if you customize command transactions you have to explicitly state where the Time Update message is placed. If you don't specify a TimeMsgPos entry no Time Update message is generated! You use the TimeMsgPos entry to specify the position of the Time Update message in a transaction. The TimeMsgPos is 0 based, so if you want the Time Update message to go out as the first message of the transactions set this to 0. For testing purposes you can insert multiple Time Update messages (one per transaction!). The MET gets updated every time the 1PPS is asserted, so even if you insert multiple Time Update messages into multiple transactions the MET reported will be the same for all messages.

You can also enable/disable the Time Update message during run-time using the [LLORRI_Bios API](#). The function LLORRI_Bios.EnableTimeMsg() lets you enable/disable the Time Update message from a Python script.

Some sample configurations (in ini style format):

```
[CmdTransactions]
Trans1 = Transaction
Trans2 = Transaction
```

```
[Trans1]
Time=200
MaxCmds    = 4
MaxBytes   = 998
TimeMsgPos = 0
```

```
[Trans2]
Time=900
```

```
MaxCmds   = 3
MaxBytes  = 200
```

```
[CmdTransactions]
Trans1 = Transaction
Trans2 = Transaction
Trans3 = Transaction
Trans4 = Transaction
```

```
[Trans1]
Time=0
MaxCmds = 5
TimeMsgPos = 0
```

```
[Trans2]
Time=50
MaxCmds = 5
TimeMsgPos = 1
```

```
[Trans3]
Time=920
MaxCmds = 10
TimeMsgPos = 0
```

```
[Trans4]
Time=990
MaxCmds = 8
```

Here is the nominal configurations (if you don't specify a [CmdTransactions] section):

```
[CmdTransactions]
Trans1 = Transaction
Trans2 = Transaction
```

```
[Trans1]
Time=100
TimeMsgPos = 0
MaxCmds   = 20
MaxBytes  = 2048
```

```
[Trans2]
Time=500
MaxCmds   = 20
MaxBytes  = 2048
```

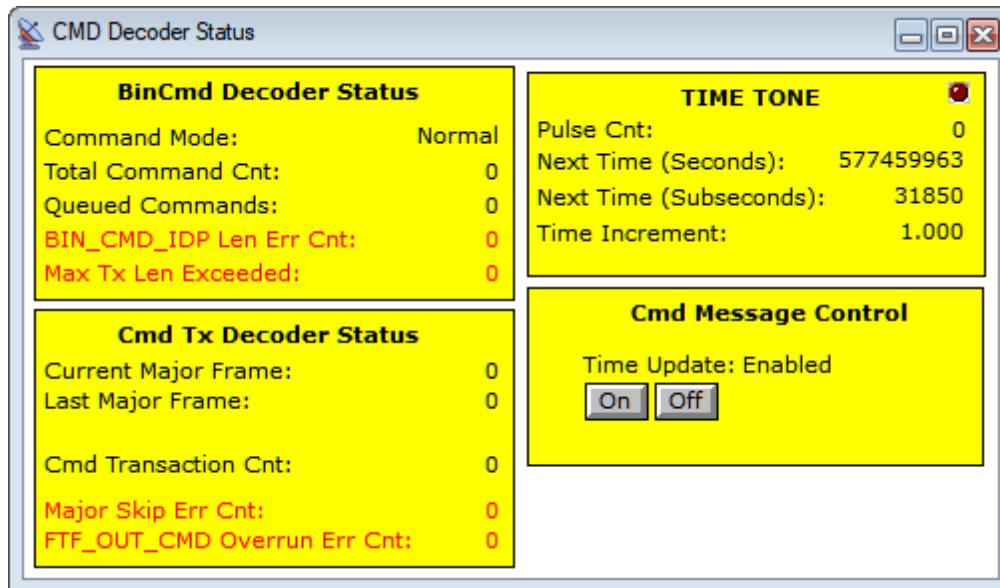

1.9 Command Processing

Once commands are defined (as described in the [previous chapter](#)) they can be issued using different mechanisms. You can execute them through the command dialog, assign them to buttons on screens, add them to a command menu or execute them from script (Python, STOL). The GSEOS command processor will convert the command from string format (command mnemonic) into a binary representation in the BinCmd block. The BinCmd block will be further decoded by the instrument specific InstrBinCmd Decoder to format the command as required into the BIN_CMD_IDP block. The BIN_CMD_IDP decoder then puts the BIN_CMD_IDP block onto the GSEOS command queue awaiting transmission.

Processing of the queue is triggered by the 1PPS pulse. Whenever the SCE generates the 1PPS pulse it sends a STATUS message to GSEOS which in turn transmits the next major frame worth of command data to the SCE. The commands contain timing information so the SCE can clock out the commands at predetermined times. The configuration of the command timing is detailed in the chapter on [Command Configuration](#).

When the STATUS block from the SCE triggers the next processing cycle the commands will be taken off the queue and bundled into command transactions to be transmitted to the SCE.

The CMD Decoder Status shows you some of the command processing status:

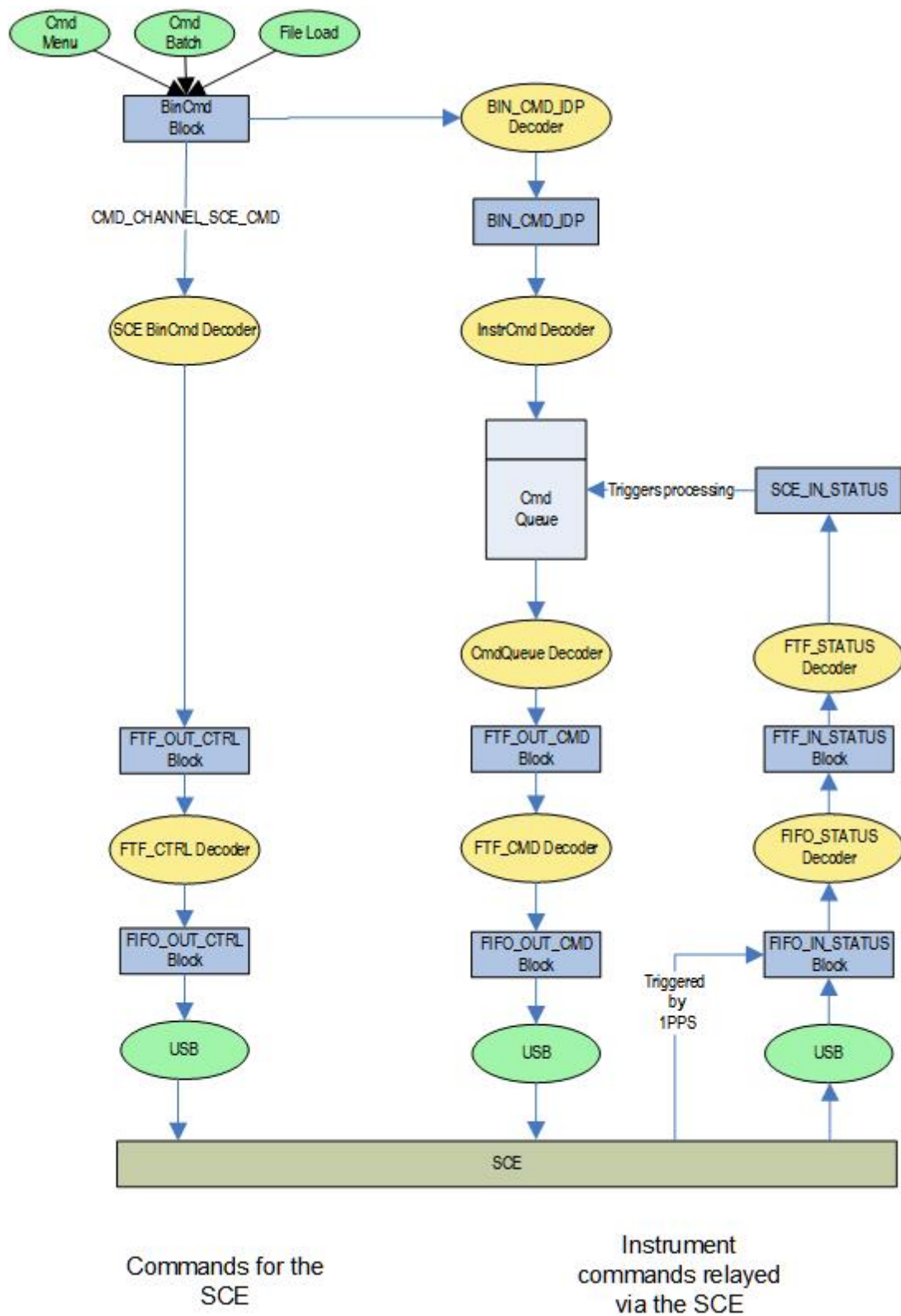


GSEOS will also add the Time Update message to the transaction as appropriate, the commands are then wrapped with the appropriate protocol layers (IP/UDP/CIP) into a command transaction.

The entire transaction will then be put into an SCE_OUT_CMD block. This block is handled by the FTF Command Decoder and transmitted to the SCE. The SCE will clock out the command data to the instrument as specified in the timing information in the SCE_OUT_CMD block.

The command queue serializes all commands, if you specify a file upload that puts many commands into the command queue it might take a while for these commands to be processed before any other commands will take effect. The L'LORRI CMD Decoder Status screen allows you to monitor the command queue and command status.

The diagram below shows the general command processing.



The BinCmd block has a Channel attribute that defines the command channel used.

Regular instrument commands use the CMD_CHANNEL_INST_CMD (0) command channel (right hand side in diagram). Besides commands destined to the instrument there are also commands destined to the SCE. These commands are transmitted on command channel CMD_CHANNEL_SCE_CMD (left hand side).

The command transactions are synchronized to the hardware TIME TIC pulse. The following paragraph describes how this synchronization is accomplished.

We define the concept of Major Frames and Minor Frames to handle these timing related issues. A Major Frame is defined as the time between two TIME TIC pulses. In our case that is 1s nominal. Each Major Frame is split into several Minor Frames with each Minor Frame being 20ms, so each nominal Major Frame will contain exactly 50 Minor Frames. The total Major Frame time can be configured. The LLORRI_Bios will schedule any outgoing data into a Minor Frame. It is the responsibility of the SCE to ensure the data is transmitted properly. If the commands happen to take more time than the 20ms allocated for the Minor Frame, any commands in the following frames will shift accordingly. If the SCE is not able to transmit the entire Major Frame before the TIME TIC pulse it will generate an error, continue to transmit the current Minor Frame and discard all remaining transactions for that major frame. It will then continue nominally with the next Major Frame.

If there is still data queued up for the old Major Frame once we crossed the TIME TIC boundary the SCE generates an error and discards any of the old Major Frame command data as described above.

GSEOS has to queue all command data that is received by user input or other sources and will transmit it to the SCE as one transaction. In order to do so GSEOS has to collect at least one full Major Frame period worth of commands, this means that the SCE will be at least one Major Frame 'behind' in sending out the data to the instrument. This means there will be a noticeable delay between the command data shown in GSEOS and the commands arriving at the instrument. The length of the 'SCE command queue' can be configured with the [SCE]/MaxCmdQueueLen and is 3 seconds by default.

The following example illustrates this process:

Let's assume the SCE has all the data for major frames N, and N+1, and is currently sending the data for major frame N out to the instrument. During this time GSEOS will collect all command data (time message, S/C data, instrument commands) on the command queue. Once it receives the STATUS message from the SCE indicating that it has completed Major Frame N and moved forward to N+1 GSEOS will package the collected command data and bundle it all up into an IP/UDP transaction and send it to the SCE with the major frame number N+2. The SCE will receive it during the time it is sending out data for N+1. The SCE will report with each STATUS message the currently executing Major Frame number as well as the last received Major Frame number. This will allow GSEOS to synchronize the sending of command data properly.

The above scheme will result in a minimum delay of ground commands of 2 seconds (if the command comes in right before the SCE reports completion of Major Frame N and the command is scheduled to go out right after assertion of the TIME TONE at N+1. The worst case will be 6 seconds (the command comes right after the SCE reports completion of Major Frame N, then GSEOS bundles it with the data for N+2, and the command goes out at the end of Major Frame N+2).

The above solution with GSEOS being one Major Frame ahead leaves a max. of 2s response time (including two round trips) for GSEOS.

This means any user code, including decoders, which runs in the foreground must not block processing for more than 2s or the synchronization between the SCE and GSEOS will be impacted and a command underflow will be reported by the SCE.

Raw Commanding

If raw command mode is enabled the commands will be transferred as-is without any additional processing. The individual raw commands will still be bundled into a transaction given the maximum number of commands per transaction and the maximum number of bytes per transaction just like regular commands. Raw commands are not subjected to the encoding into the L'LORRI protocol stack (IP/UDP/CIP). They can be used before these layers are implemented in the flight software or for injecting errors into the IP/UDP/CIP protocol layers (by setting these protocol headers by hand and injecting errors as appropriate). Raw commands are generated when the system is configured for raw commanding. Please check in the [configuration section](#) for more details.

1.10 Raw Commanding

[Raw Commanding](#)

Regular instrument commands are encoded with the proper L'LORRI protocol stack (IP/UDP/CIP). It is possible to issue raw commands which will not be subjected to this nominal encoding. The command bytes as defined in the command definition (and supplemented by the according command arguments) are issued as-is to the instrument. The system can be configured to be either in regular or in raw command mode. The default command mode is regular commanding.

To turn on raw command mode you have to set the following entry in your gseos.ini file:

```
[Bios]
RawCmdMode = Yes
```

In raw mode the Time Update message will not be generated. The raw commands will be bundled into 'transactions' according to the transaction definition with the limitations of number of commands per transaction and number of bytes per transaction. See the [transaction configuration](#) for more information. The command data will also be transmitted to the instrument at the time configured for the transaction.

Raw command mode is useful if your instrument doesn't have the L'LORRI protocol stack implemented and you just want to send a sequence of bytes to the instrument.

[Error Injection](#)

The other use for raw command mode is to generate error injection for instrument testing. You can create a binary command transaction that looks identical to a valid IP/UDP/CIP packet and inject errors into any header field you would like to test. If you then issue this binary sequence as a raw command you effectively generate a packet with targeted errors.

1.11 Error Injection

There are several different mechanisms for error injection:

[Command UART Hardware Errors](#)

Allows you to instruct the command UART to inject hardware errors like parity errors or frame errors.

[Raw Commanding](#)

In this mode you can construct your own command data as needed and have full control over all the data that is sent to the instrument.

1.11.1 Command UART Hardware Errors

Besides generating data level errors like command packet and raw commanding you can also generate UART errors on hardware level. These errors are generated by the FPGA and can be configured with the following command:

```
SCE_CTRL_INJECT_UART_ERROR()
```

The `SCE_CTRL_INJECT_UART_ERROR()` function takes a number of arguments and lets you inject the following errors:

- Parity Error
- Frame Error
- Bit Error

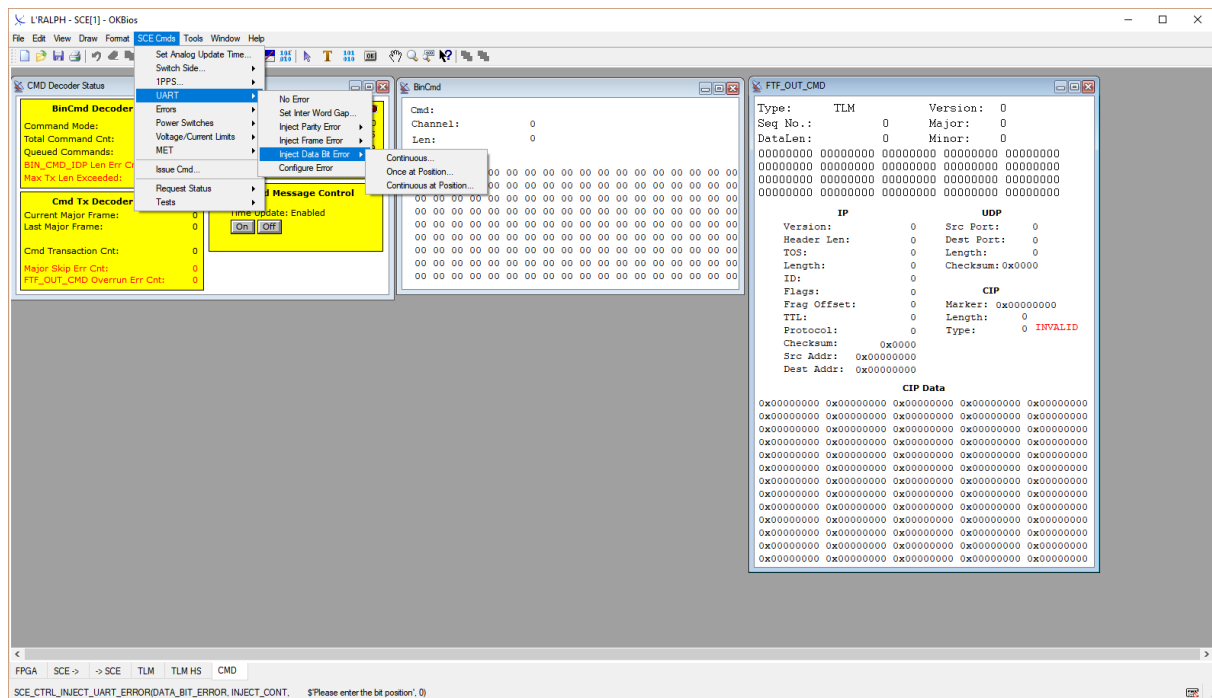
For all errors you can specify one of three modes:

Continuous, Once at specified position, or Continuous at specified position.

In Continuous mode an error is injected in each Byte in the given the bit position (the Parity and Frame errors don't use a bit position). The "Once at specified position" mode will inject the error exactly once in the specified Byte, and for the "Continuous at specified position" mode the error is injected continuously in the specified Byte (and bit if applicable) for each 1PPS.

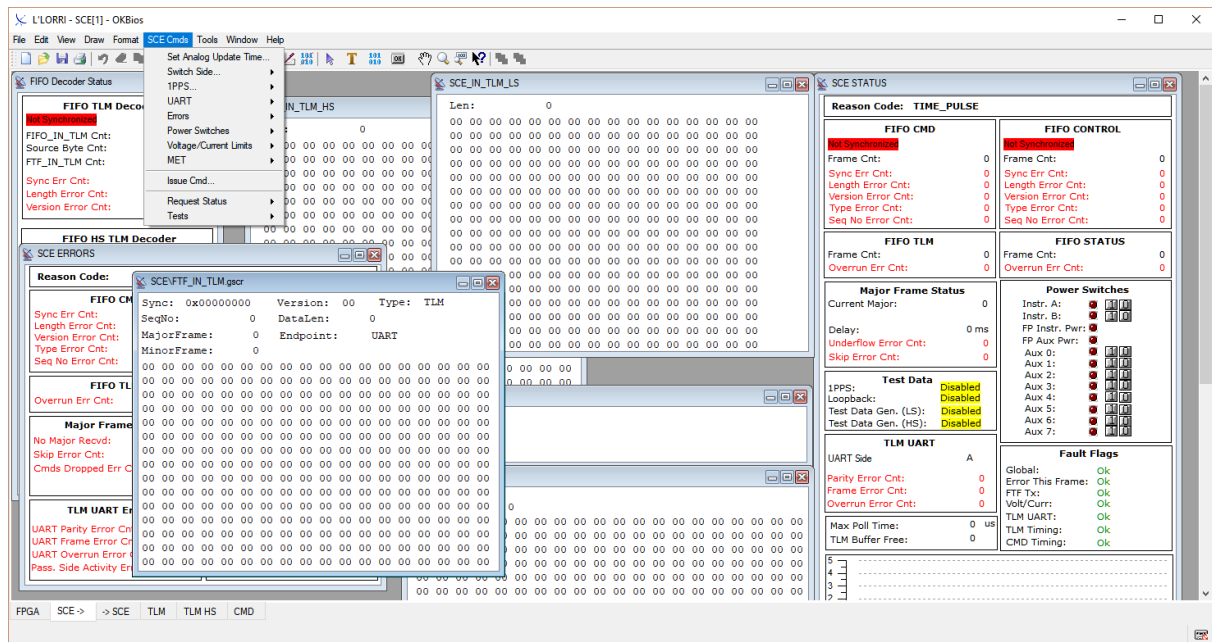
The Bit Errors allow you to also specify a Bit position. For the Bit Error injection you can specify bit 0 through 7 which corresponds to the eight data bits.

The SCE command menu gives you quick access to the error injection commands, the SCE Settings screen shows the current error injection settings:



1.12 SCE Commands

The SCE specific commands can be accessed from the 'SCE Cmds' menu.



These commands can also be issued from Python script or using the command dialog (F7). Find a list of all defined commands with a brief description below:

Command	Description
SCE_CONFIG_START_ACQUISITION	Starts the emulator data acquisition. This command is issued automatically on startup. You can override this behavior with the [SCE]/StartAcquisition switch. If you set this setting to Manual you have to issue the SCE_CONFIG_START_ACQUISITION command manually, otherwise the emulator will not collect TLM data.
SCE_CTRL_SET_POWER_SW	Set power switches.
ITCHES	
SCE_CTRL_ENABLE_INSTR_A_POWER	Enable instrument A power.
SCE_CTRL_DISABLE_INSTR_A_POWER	Disable instrument A power.
SCE_CTRL_ENABLE_INSTR_B_POWER	Enable instrument B power.
SCE_CTRL_DISABLE_INSTR_B_POWER	Disable instrument B power.
SCE_CTRL_ENABLE_AUX0_POWER	Enable aux 0 power.
SCE_CTRL_DISABLE_AUX0_POWER	Disable aux 0 power.
...	...
SCE_CTRL_ENABLE_AUX7_POWER	Enable aux 7 power.
SCE_CTRL_DISABLE_AUX7_POWER	Disable aux 7 power.
SCE_CTRL_SET_ANALOG_UPDATE_TIME	Sets the rate (in ms) at which the SCE will send analog data to GSEOS.
SCE_CTRL_SELF_TEST	Initiate SCE self test.
SCE_CTRL_SWITCH_SIDE	Switch to side A or B.

SCE_CTRL_SET_UART_INTE Set the gap between WORDS in uSeconds.
 R_WORD_GAP
 SCE_CTRL_INJECT_UART_ER Inject error into the CMD UART.
 ROR
 SCE_CTRL_ENABLE_LOOPBA Enable/disable CMD to TLM loopback mode.
 CK
 SCE_CTRL_RESET_FRONTPA Reset the frontpanel error LED and faults.
 NEL_ERROR
 SCE_CTRL_RESET_ERROR_C Reset the SCE error counters.
 COUNTERS
 SCE_CTRL_RESET_PEAK_VA Reset the SCE peak value counters.
 LUES
 SCE_CTRL_REQUEST_SETTI Request the SETTINGS message.
 NGS
 SCE_CTRL_REQUEST_ERROR Request the ERRORS message.
 S
 SCE_CTRL_SET_VOLTAGE_LI Set instrument/aux high/low voltage limits.
 MIT
 SCE_CTRL_SET_CURRENT_LI Set instrument/aux high/low current limits.
 MIT
 SCE_CTRL_ENABLE_TIME_M Enable/disable the Time Update message.
 SG
 SCE_CTRL_ENABLE_1PPS_N Enable/disable non gated 1PPS signal.
 ON_GATED
 SCE_CTRL_ARM_1PPS_GATE Arm the 1PPS gated 1 trigger. The emulator provides two 1PPS
 D1 trigger outputs. These are single shot triggers that can be used to
 synchronize external equipment to the 1PPS pulse. These triggers
 have to be armed with the SCE_CTRL_ARM_1PPS_GATED1() or
 SCE_CTRL_ARM_1PPS_GATED2() commands. Once you issue the
 command the SCE will assert the trigger line on the following 1PPS
 event.
 SCE_CTRL_ARM_1PPS_GATE Arm the 1PPS gated 2 trigger.
 D2
 GSEOS_CMD_SET_MET Sets the current spacecraft MET.
 GSEOS_CMD_SET_MET_INC Set the MET increment. The default is 1.
 REMENT

The "Run L'LORRI Bios Self Tests..." menu allows you to run the self test suites. For this to succeed you must disconnect the SCE before running the tests. The self tests do not perform end-to-end testing but test the internal LLORRI_Bios code. A test report will be printed to the Console window. Click F9 to display the console window. After running the tests you should shut down GSEOS and restart to initialize the system properly.

1.13 Python API

The LLORRI_Bios module exports a few functions that you can call via Python:

[EnableTimeMsg\(bEnable\)](#)

Enable or disable the Time Update message.

[RunSelfTests\(\)](#)

Runs the built-in unit tests for each module.

`Reset()`

Reset all decoders. Note, this does not reset the SCE.

`ResetErrorCounters()`

Resets all error counters.

1.14 FAQs

1.14.1 How do I customize my configuration?

Each instrument customizes its configuration in its own folder. See below of a directory listing of the L'LORRI GSEOS setup:

```

▼ [Icon] LLORRI.1.0.004
  [Icon] Doc
  > [Icon] Instruments
    [Icon] LLORRI
    [Icon] LLORRI_Common
  > [Icon] SCE
  > [Icon] System
  > [Icon] Test

```

All instrument customizations are located in the Instruments folder. Say you want to customize the L'LORRI instrument all your custom files would end up in the folder: `./Instruments/LLORRI`.

This layout helps to keep the configurations separate and makes software updates seamless.

There are a few default configuration files that have been pre-configured. The `gseos.ini` file is the main configuration file, the `.dt` file is the desktop file that contains the layout of the windows displayed.

The `.dt` file is updated automatically as you create and display new screens or add new tabs. The `gseos.ini` file is your main file for system customization. The following paragraphs outline the most important settings. For more information refer to the general GSEOS documentation.

```

#
*****
#
# * GSEOS.INI for L'LORRI
* #

```

```
# *
# * #
# * Copyright (C) 1998-2018, GSE Software, Inc.
# * #
# * Author: Thomas Hauck
# * #
# * #
# * #
# * #
# * History:
# * #
# * Feb-02-2018 R001 th 1st version
# * #
#
*****
#

__include__ LLORRI_Common/gseos.ini

[Project]
Name=L'LORRI
Title=L'LORRI
Splashbitmap=LLORRI_Common/LUCY.jpg

[Config]
__include__ LLORRI_Common/gseos.ini Config
Load = Instruments/LLORRI/LLORRI.dt

[Bios]
EnableTimeMsg=No

[Port]
Baudrate   = 115200
Parity     = e
Stopbits   = 1
Databits   = 8

[Command]
LogFile = Instruments/LLORRI/LLORRI_Cmd.log
Python  = Enable

[Console]
MaxFileSize=1024
FileName=Instruments/LLORRI/LLORRI.con

[Message]
FileName=Instruments/LLORRI/LLORRI.msg
```

The first entry imports all settings from the LLORRI_Common folder. This is responsible for loading system wide files and also helps keeping the installations independent. If a global change to the LLORRI_Common/gseos.ini file is made in a future release you will automatically inherit these settings.

In the [Project] section you should specify your instrument name and maybe configuration settings (EM, FM, Calibration, etc.). You can also provide a custom bitmap. Keep in mind that the bitmap should be about 450 pixels wide.

The [Config] section allows you to load your custom files using the Load entry. Again, you will inherit the general system settings from LLORRI_Common. The [Bios], [SCE], and [Port] sections let you customize the LLORRI_Bios according to your needs. Please check the [Configuration](#) section for more information.

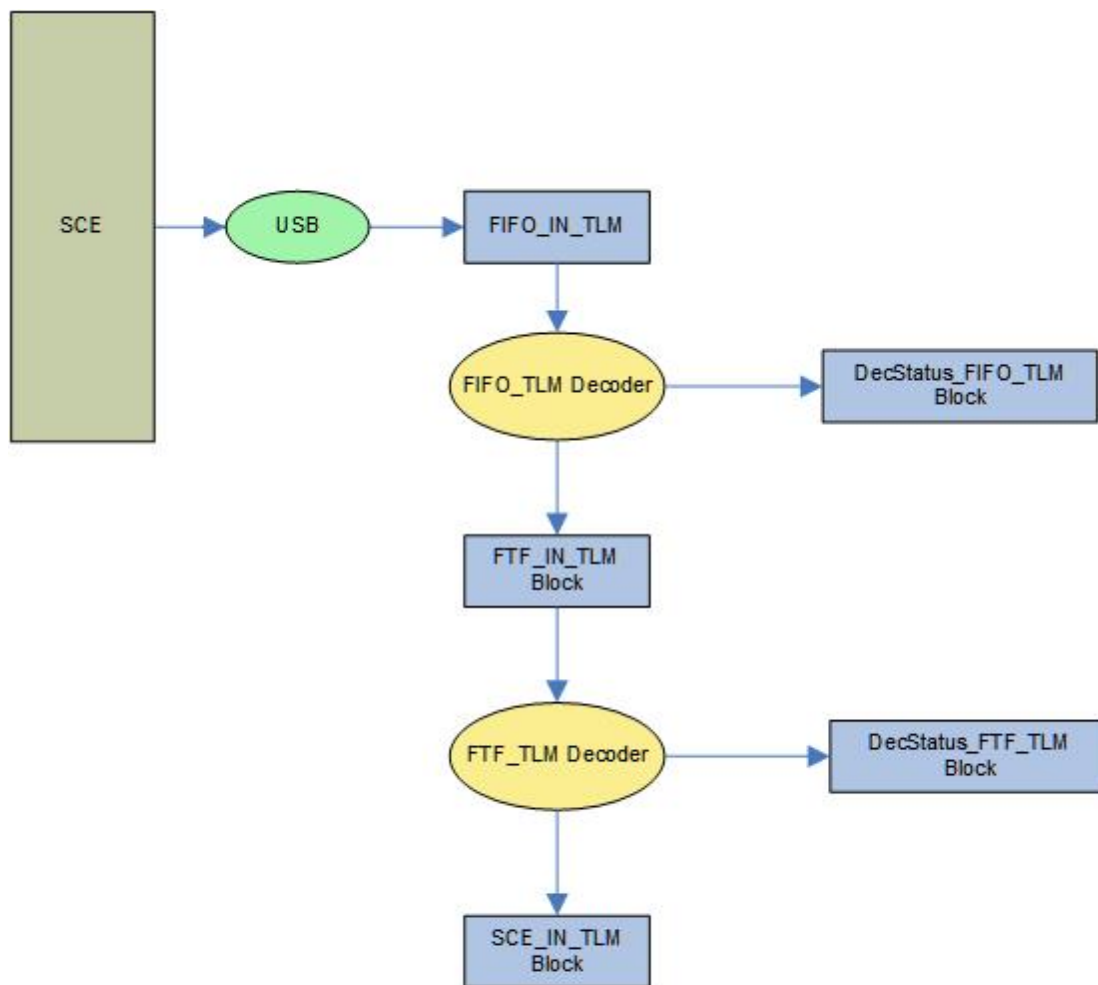
The [Command], [Console], and [Message] sections should point to files in your own folder so we keep them separate from the global files and also other instruments.

For more information about the various gseos.ini settings check the general GSEOS documentation.

1.14.2 What if I don't generate the S/C protocol stack for my TLM data?

The incoming telemetry data is expected to be formatted in valid IP/UDP/CIP packets containing IDP (Instrument Dependent Protocol) packets. During early instrument development the flight software to generate these headers might not be in place. Since GSEOS uses a layered approach you will still be able to access your raw data.

As you can see in the image below the raw SCE_IN_TLM_LS/HS data is processed by the SCE_IN_TLM decoder which will check for proper IP/UDP headers. Since your instrument doesn't generate IP/UDP data you can simply use the SCE_IN_TLM_LS/HS block which contains the raw data. The Len field of the SCE_IN_TLM_LS/HS block indicates how many bytes of data are available in the SCE_IN_TLM_LS/HS.Block array field.



SCE -> GSEOS Incoming TLM Data

1.14.3 How do I customize instrument specific commands?

When you define your instrument commands using a .cpd file as described in the chapter on [Command Handling](#) GSEOS translates your command into a binary sequence which will then be forwarded in the BinCmd block. If you have an instrument specific command protocol, say a specific checksum algorithm that is not supported by the native GSEOS command definition you have to annotate your command data. This chapter describes how to customize the command handling to accommodate your instrument dependent protocol.

The built-in instrument BinCmd decoder takes BinCmd blocks and copies blocks with channel CMD_CHANNEL_INSTR_CMD (0, which is the default) into BIN_CMD_IDP blocks. If you plan on customizing the command handling you can disable this decoder and replace with you own. You can disable the decoder programmatically with the following statement:

```
GseosDecoder.StopDecoder('BinCmd for Instrument Cmds Decoder')
```

The GSEOS Explorer will show the disabled decoder like this:

Decoder	Count	Total Time	Avg. Time	Last Time	Max. Time	Installed On	Description
1 BinCmd for Instrument Cmds Decode	0	0.000	0.000	0.000	0.000	BinCmd	
2 CIP Decoder	0	0.000	0.000	0.000	0.000	TLM_LS_IP	
3 Command Decoder	0	0.000	0.000	0.000	0.000	CmdString	GSEOS Command D
4 Command Logger	0	0.000	0.000	0.000	0.000	CmdString, BinCmd	
5 FIFO_STATUS Decoder	0	0.000	0.000	0.000	0.000	FIFO_IN_STATUS	Decode unsynchron
6 FIFO_TLM Decoder	0	0.000	0.000	0.000	0.000	FIFO_IN_TLM	Decode unsynchron
7 FIFO_TLM_HS Decoder	0	0.000	0.000	0.000	0.000	FIFO_IN_TLM_HS	Decode unsynchron
8 FTF_CMD Decoder	0	0.000	0.000	0.000	0.000	FTF_OUT_CMD	
9 FTF_CTRL Decoder	0	0.000	0.000	0.000	0.000	FTF_OUT_CTRL	
10 FTF_IN_TLM Decoder	0	0.000	0.000	0.000	0.000	FTF_IN_TLM	Decode FTF_IN_TL
11 FTF_IN_TLM_HS Decoder	0	0.000	0.000	0.000	0.000	FTF_IN_TLM_HS	Decode FTF_IN_TL
12 FTF_STATUS Decoder	0	0.000	0.000	0.000	0.000	FTF_IN_STATUS	Decode FTF_IN_ST
13 IP Decoder	0	0.000	0.000	0.000	0.000	SCE_IN_TLM_LS	
14 LLORRI BIN_CMD_IDP Decoder	0	0.000	0.000	0.000	0.000	BIN_CMD_IDP	
15 LLORRI Cmd Tx Decoder	0	0.000	0.000	0.000	0.000	SCE_IN_STATUS	
16 SCE BinCmd Decoder	0	0.000	0.000	0.000	0.000	BinCmd	
17 Time Tone Decoder	0	0.000	0.000	0.000	0.000	FTF_OUT_CMD	

You then have to install your own custom decoder. You can use the template below as a starting point. The template makes a mere copy of the BinCmd contents. You would want to modify the data depending on your needs. You should then load the decoder from your gseos.ini [PyStartup] section.

MyInstrBinCmdDecoder.py:

```
# ***** #
# * MyInstrBinCmdDecoder.py * #
# * #
# * This module implements the instrument specific command handling for... * #
# * #
# * The decoder handles regular commands that are sent on channel * #
# * CMD_CHANNEL_INSTR_CMD and decodes them into BIN_CMD_IDP instrument * #
# * specific blocks. * #
# ***** #

# ----- #
# - Imports - #
# ----- #
import GseosBlocks
import GseosDecoder

import LLORRI_Bios

# ===== #
# = Exceptions. = #
# ===== #
class TMyInstrBinCmdDecoderError(Exception): pass

# ===== #
# = Data. = #
# ===== #
```

```

# ***** #
# *           B i n C m d   D e c o d e r           * #
# *                                                                 * #
# * Intercept GSEOS commands on command channel CMD_CHANNEL_INSTR_CMD and * #
# * perform the proper command. * #
# ***** #
class TBinCmdDec:
# ***** #
# * ctor() * #
# * * #
# * Initialize all decoder state and register the decoder with the * #
# * data source requested. * #
# * * #
# * Parameters: dwCmdChannel:      The instrument command channel. * #
# ***** #
def __init__(self, dwCmdChannel):
# ----- #
# - Initialize member data. - #
# ----- #
self.dwCmdChannel = dwCmdChannel

# ----- #
# - Make sure our blocks are defined properly. - #
# ----- #
try:
    self.oBlkSrc      = GseosBlocks.Blocks['BinCmd']
    self.oBlkDestCmd = GseosBlocks.Blocks['BIN_CMD_IDP']

except KeyError, oX:
    raise TMyInstrBinCmdError("GSEOS block %s is not defined." % oX)

# ----- #
# - Create the decoder object and install on our source block. - #
# ----- #
oDec = GseosDecoder.TDecoder('BinCmd Decoder for XXX', self.fDecoder,
                             [self.oBlkDestCmd])

# ----- #
# - Hook the decoder to our source block. - #
# ----- #
self.oBlkSrc.Decoders.append(oDec)

# ***** #
# * fMyCmdIDP() * #
# * * #
# * Perform whatever instrument specific conversion is necessary. * #
# * * #
# * Parameters: ctData:      The raw command data. * #
# * Return:      ctIDPData: The data formatted in IDP format. * #
# ***** #
def fMyCmdIDP(self, ctData):
# ----- #
# - Simple checksum, we replace the last two bytes with the sum over - #
# - all command bytes. - #
# ----- #
wChecksum = sum(ctData[:-2])
wChecksum &= 0xFFFF

```

```

ctIDPData = ctData[:-2] + [wChecksum >> 8, wChecksum & 0xFF]
return ctIDPData

# ***** #
# * fDecoder() * #
# * * * #
# * Copy the command to the BIN_CMD_IDP block if is on our instrument * #
# * command channel. * #
# * * #
# * Parameters: oBlkSrc: BinCmd block. * #
# ***** #
def fDecoder(self, oBlkSrc):
# ----- #
# - CMD_CHANNEL_INSTR_CMD - #
# ----- #
if oBlkSrc.Channel == self.dwCmdChannel:
# ----- #
# - Convert our command data depending on IDP. - #
# ----- #
ctData = self.oBlkSrc.CmdData[:oBlkSrc.Len]
ctIDPData = self.fMyCmdIDP(ctData)

# ----- #
# - Generate BIN_CMD_IDP block, insert your custom data here. - #
# ----- #
self.oBlkDestCmd.Len = len(ctIDPData)
self.oBlkDestCmd.Data[:] = ctIDPData
self.oBlkDestCmd.SendBlock(bCopy=True)

# ----- #
# - Create the BinCmd decoder. - #
# ----- #
_oBinCmdDec = TBinCmdDec(LLORRI_Bios.CMD_CHANNEL_INSTR_CMD)

# ----- #
# - Stop system decoder. - #
# ----- #
GseosDecoder.StopDecoder('BinCmd for Instrument Cmds Decoder')

```

1.14.4 How do I use command channels?

Command channels are used by GSEOS to route commands to different destinations. There are currently three command channels defined:

LLORRI_Bios.CMD_CHANNEL_GSEOS_CMD

This channel is used for L'LORRI GSEOS specific commands like setting the MET or time increment.

LLORRI_Bios.CMD_CHANNEL_INSTR_CMD

This is the default command channel (0). If you don't specify a "Channel" attribute in your

command definition the command will have a channel of 0. All commands on channel 0 are handled as instrument commands.

LLORRI_Bios.CMD_CHANNEL_SCE_CMD

Commands on this channel are directed to the SCE. These commands allow to modify SCE settings.

If you need to use custom command channels please use channels in the range: 512-1024. This will allow to extend the common L'LORRI command channels in the future without clashing with instrument specific command channels.

Why would I use a custom command channel?

If you have custom software within your L'LORRI extensions and you want to use the GSEOS command definitions (as opposed to Python functions). You could define a command channel and issue commands to your custom module, i.e. configuration options for a simulation.

1.14.5 How do I inject errors using raw command mode?

To inject custom errors into the L'LORRI protocol headers (IP/UDP/CIP) or your own command data you can use [raw commands](#).

To put the system in raw command mode set the following entry in your gseos.ini file:

```
[Bios]
RawCmdMode = Yes
```

You have to restart GSEOS for this change to take effect.

First you have to generate the proper protocol headers yourself, inserting errors where necessary. Let's say we want to introduce a UDP checksum error.

We first have to construct the 20 bytes of IP header, 8 bytes of UDP header, and 8 bytes of CIP header and finally add our command data. Lets say we simply insert a counting pattern of 20 bytes. The binary command data would look something like this:

```
0x45, 0x00, 0x00, 0x2C, 0x00, 0x00, 0x40, 0x00, 0xFF, 0x11, 0x59, 0x53,
0x64, 0x66, 0x2C, 0xD1, 0x64, 0x66, 0x2C, 0xD0, 0x0D, 0x05, 0x0D, 0x05,
0x00, 0x18, 0xC1, 0x46, 0x00, 0x00, 0x00, 0xF0, 0x00, 0x10, 0x01, 0x00,
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B,
0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F
```

The UDP checksum is 0xc146, we will change this to 0x46c1. This results in the following pattern:

```
0x45, 0x00, 0x00, 0x2C, 0x00, 0x00, 0x40, 0x00, 0xFF, 0x11, 0x59, 0x53,
0x64, 0x66, 0x2C, 0xD1, 0x64, 0x66, 0x2C, 0xD0, 0x0D, 0x05, 0x0D, 0x05,
0x00, 0x18, 0x46, 0xC1, 0x00, 0x00, 0x00, 0xF0, 0x00, 0x10, 0x01, 0x00,
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B,
0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F
```

We will now generate a binary command block with this error injected data. You can either

type the following in the GSEOS console window or save it to a Python script which you then load using the File/Open menu and select: Python Files (.py) as the file type:
(if you run from a script you have to import the BinCmd block into the script namespace using: `from __main__ import BinCmd`)

```
ctData = [0x45, 0x00, 0x00, 0x2C, 0x00, 0x00, 0x40, 0x00, 0xFF, 0x11, 0x59,
0x53, 0x64, 0x66, 0x2C, 0xD1, 0x64, 0x66, 0x2C, 0xD0, 0x0D, 0x05, 0x0D,
0x05, 0x00, 0x18, 0x46, 0xC1, 0x00, 0x00, 0x00, 0xF0, 0x00, 0x10, 0x01,
0x00, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A,
0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16,
0x17, 0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F]
```

```
BinCmd.Len          = len(ctData)
BinCmd.CmdData[:] = ctData
BinCmd.SendBlock()
```

You can of course corrupt the data in any way you think is useful for testing. You can also add multiple commands to this transaction or even construct multiple transactions. Keep in mind that the command length must be a multiple of four, otherwise you will get a Length error from the command decoder and your command will not get queued. Also, make sure to not exceed the maximum transaction length.

Once you are done with your testing don't forget to set the RawCmdMode entry to No:

```
[Bios]
RawCmdMode = No
```

1.14.6 How do I implement File Uploads?

L'LORRI uses a bent-pipe scheme for commanding. That is each instrument is free to define its own command definitions and memory or file upload mechanism. GSEOS provides a framework to implement a custom file upload that is specific to each payloads needs. There are a wide variety of input formats from simple binary files to record based (SFDU, etc.) files. On the output side each instrument has a different command or set of commands to accommodate a memory or file upload. The general mechanism of the file upload consists of multiple steps:

- 1) Select the upload file and specify any parameters (like start address) needed for the upload.
- 2) Issue any setup commands to prepare for the file load.
- 2) Loop over the file extracting chunks and issuing the proper memory upload command(s).
- 3) Issue any cleanup commands to finalize the file upload.

The file type to be uploaded can be configured by file extension. Any parameters that need to be supplied can be configured also. When the user initiates the file upload he chooses the proper file type and a file selection dialog let's him choose the desired file. Then a dialog for the configuration parameters is presented. The configured FileUploadStartHandler() function is called an these parameters is passed along.

Then the FileUploadTimerHandler() function is called everytime a configured timer expires

and the user code can read the next chunk of file data and format it into commands or BinCmd blocks as required.

The following section gives an overview of the various gseos.ini settings to configure the File Upload:

Configuration Settings

The file upload is configured in the gseos.ini configuration file. You can specify multiple file upload types. Each file upload type is configured with the [FileUploads] section. See the example below that specifies three different file types:

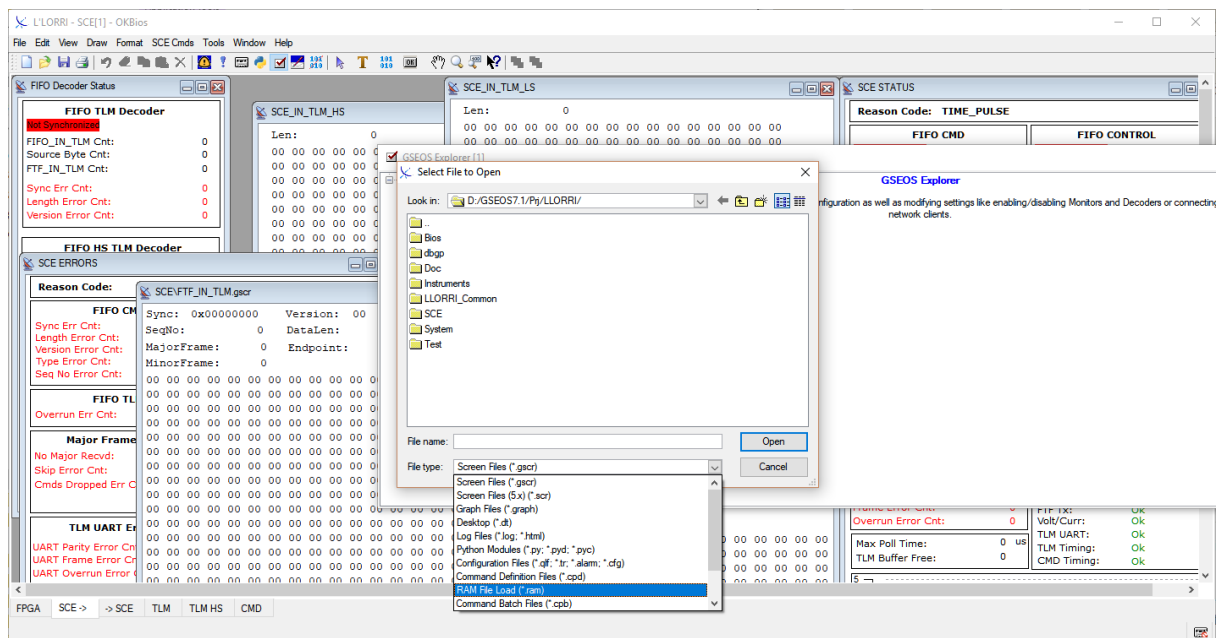
```
[FileUploads]
RAMLoad=FileUpload
EEPROMLoad=FileUpload
MemLoad=FileUpload
```

The name of each file upload type (RAMLoad, EEPROMLoad, MemLoad) is the name of a configuration section that further specifies the details of this file upload type:

```
[RAMLoad]
Description=RAM File Load
Extension=ram
Timer=100
FileUploadStartHandler=MyModule.fOnRAMLoadStart
FileUploadTimerHandler=MyModule.fOnRAMLoadProcess
```

```
Arg0 = StartAddr: 0x2000, 0x3FFF
Arg1 = Mode:      SWAP, BLANK, XOR
Arg2 = Verify:    ON, OFF
```

The above example configures the RAMLoad file upload. The description will show up in the GSEOS File Open dialog to describe the file upload type, the extension is used as the file filter. The user initiates a file load by simply selecting the desired file. The screenshot below shows the open File Select Dialog with the RAM File Load file type selected.



The Timer setting specifies the time out time of the upload timer. The FileUploadStartHandler and FileUploadTimerHandler settings specify your callback handler routines where you will fill in your code to process the file and issue the commands. The ArgX settings configure any additional arguments you want the user to specify when loading the file like start address etc..

The following paragraph describes the configuration settings in detail:

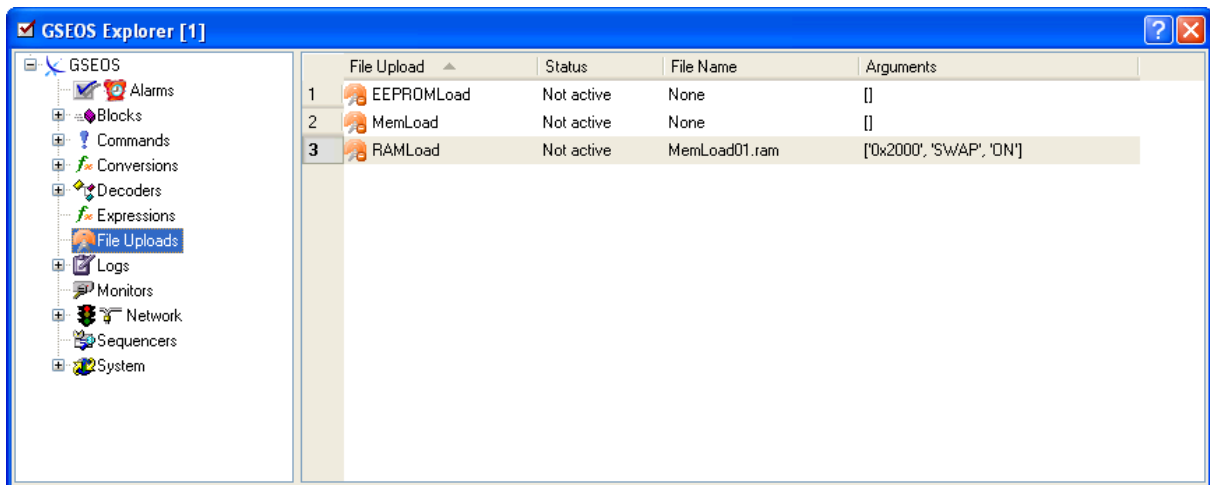
Key	Assignment
Description	This should be a brief description of the file upload type. This will show up in the File/Open dialog File Type setting.
Extension	The file extension, this should be a three character extension that doesn't conflict with any of the other GSEOS common extensions. The '.' (dot) must not be included in the file extension.
Timer	The time out interval in ms. The FileUploadTimerHandler routine (see below) is called whenever this timer expires.
FileUploadStartHandler	This entry must specify your callback function that is called when the user starts a file upload. This function must take two arguments, the first one is the absolute file name of the file the user has selected, the second is a list of arguments the user has selected. Even if there is only one argument it will be delivered as a list with one argument. If the user specifies arguments from a select list they will be provided as strings, numeric arguments will be checked against the proper range and be forwarded as integer. You can specify a module and package name in typical Python namespace syntax. You have to make sure that this name is accessible (i.e. the module is loaded). If this function returns False the file upload is aborted, if True the upload timer is started and the FileUploadTimerHandler function called. If it returns a string the string is interpreted as an error string and the argument query dialog is popped up again displaying the error

and let the user correct the argument entry.

FileUploadTimerHandler This entry sets the timer callback. In this function you should service the file, read the next chunk of data and prepare and issue the command(s) as required. This function doesn't take any parameters, it is your responsibility to keep references to the open file, and any other data you need to process the file. Your function should return True if there is more data to be processed, it should return False when you are done with the file. Once this function returns False the file upload is finished and your routine won't be called until the user uploads another file.

Arg0...Arg4 You can configure up to five arguments the user can specify when choosing a file to upload. Each Argument takes the prompt followed by a colon and either a select list or a range. A select list is a comma delimited list. A range is indicated by two elipses (n..m). If neither a list nor a range is specified the argument will simply prompt for any string. If given a range the user input will be verified against the range (both ends are inclusive), if a select list is provided the user can choose one of the settings specified. If a range is specified only decimal integer values can be entered by the user. If you require hex values you can simply use a string argument and interpret the value in your handler. The user entries will be passed as a list as the second argument to the FileUploadStartHandler callback function you supplied.

You can verify the File Upload progress using the GSEOS Explorer. The File Upload node displays all configured file uploads and the current status of the uploads.



Programmatic Access

You can use the general Gseos.FileOpen() function to start your file upload from a script. Keep in mind that on startup you are prompted for the input parameters which effectively blocks the function. This is not allowed in GSEOS decoders or monitors.

Simple File Upload Sample

In the following example we are going to set up a simple binary file upload. We make the following assumptions:

- a) The upload file is a simple binary file, that is we can take chunks of the file and upload the data as is.
- b) Each file upload command that we issue as part of the file upload is self-contained and has the upload opcode (one byte) and the 16-bit start address for the data following. The next parameter is the memory type: FLASH, RAM, EEPROM, the last parameter is the number of bytes to upload. The data portion can be up to 256 bytes.
- c) The start address must be in the range 0x4000..0x7FFF. We want the user to be able to enter a hexadecimal start address.

First we have to start with the gseos.ini configuration of the file upload we want to set up:

```
[FileUploads]
BinLoad=FileUpload

[BinLoad]
Description=Binary File Load
Extension=bin
Timer=250
FileUploadStartHandler=BinFileLoad.fOnStart
FileUploadTimerHandler=BinFileLoad.fOnTimer

Arg0 = StartAddr:
Arg1 = Memory: FLASH, RAM, EEPROM
```

We will put all the file upload logic into the BinFileLoad module (which we will discuss below). So the Start and Timer handlers get assigned the according functions. The BinFileLoad module needs to be accessible so we will load it from the [PyStartup] gseos.ini section:

```
[PyStartup]
__include__ L'LORRI_Common/gseos.ini PyStartup
Import = BinFileLoad
```

Keep in mind that we import the [PyStartup] section from the L'LORRI_Common gseos.ini file. So in order to reuse the common settings we simply import all the common [PyStartup] settings using the __include__ directive.

We could also use a batch file to load the BinFileLoad module. However, if we use the [Config]/Load section to load the module it will not be 'imported' into the __main__ namespace and would therefore not be found.

Since we want the user to be able to enter hex start addresses we simply use a string type argument for the start address (not the range since that only allows decimal integers). We will parse the string ourselves in the Start handler. The Memory type can be specified using a select list, we will convert the selected string into the proper code in the Start handler as well.

After having configured the file upload we now have to write our actual command handling. The BinFileLoad module pretty much provides the two handlers for the file upload start and timer notifications. Besides these two routines we will keep some global state:

```
#
```

```

*****
#
# * BinFileUpload.py
# * #
# *
# * #
# * Simple binary file upload sample.
# * #
#
*****
#

#
-----
#
# - Imports
# - #
#
-----
#
import struct
import GseosBlocks

#
=====
#
# = Defines.
# = #
#
=====
#
OPCODE_UPLOAD = 0x5F

#
=====
#
# = Data
# = #
#
=====
#
oFile      = None
wCurrAddr  = 0
wMemType   = 0
BinCmd     = GseosBlocks.Blocks['BinCmd']

#
*****
#
# * fOnStart()
# * #
# *

```

```

* #
# * File upload start handler, get's called when the user opens a file of
* #
# * type: *.bin.
* #
# * We open the file and keep a reference to the open file for further
* #
# * processing. We also check the user provided arguments and convert them
* #
# * accordingly.
* #
# * We don't have to check for exceptions since they will be handled by the
* #
# * file upload mechanism and abort the file upload.
* #
# *
* #
# * Parameters: strFile: The file name of the file upload.
* #
# *           ctArgs: A list with user selected arguments.
* #
# *           According to our configuration the first item is
* #
# *           the start address (as a string) and the second
* #
# *           is the memory type.
* #
# * Returns:   oRet: True, if everything is ok and we need to continue
* #
# *           the file load.
* #
# *           False, if the file load can be completed here
* #
# *           (that is the entire file is less than 256 bytes).
* #
# *           An error string if there is a problem with the
* #
# *           arguments and we want to prompt the user to
* #
# *           correct his selection.
* #
#
*****
#
def fOnStart(strFile, ctArgs):
    global oFile
    global wCurrAddr
    global wMemType

    #
    ----- #
    # - Check the arguments.
    - #

```



```

#
----- #
wCurrAddr = eval(ctArgs[0])

if not 0x4000 <= wCurrAddr <= 0x7FFF:
    return 'The start address must be in range: 0x4000..0x7FFF'

strMemType = ctArgs[1].strip()
wMemType   = {'FLASH': 0, 'RAM': 1, 'EEPROM': 2}[strMemType]

#
----- #
# - Open the file and read the first chunk of data.
- #
#
----- #
oFile = file(strFile, 'rb')

strData = oFile.read(256)
wLen    = len(strData)

#
----- #
# - Generate the command data. We pack the data using big endian.
- #
#
----- #
strCmdData      = struct.pack('>BHHH%ds' % wLen,
                               OPCODE_UPLOAD,
                               wCurrAddr,
                               wMemType,
                               wLen,
                               strData)

BinCmd.Len      = len(strCmdData)
BinCmd.Channel  = 0
BinCmd.CmdData[:] = strCmdData
BinCmd.SendBlock()

#
----- #
# - Update our current upload address.
- #
#
----- #
wCurrAddr += wLen

#
----- #
# - We are at the end of the file, terminate the file upload.
- #

```

```

#
----- #
if wLen < 256:
    oFile.close()
    return False

#
----- #
# - Continue file upload with next timer expiration.
- #
#
----- #
return True

#
*****
#
# * fOnTimer()
# *
# *
# *
# * We continue uploading data in chunks of 256 bytes. If we run out of
# *
# * data we terminate the upload.
# *
# *
# *
# * Parameters: -
# *
# * Returns:      bRet:      True, if everything is ok and we need to continue
# *
# *                  the file load.
# *
# *                  False, if we are done with the upload.
# *
#
*****
#
def fOnTimer():
    global oFile
    global wCurrAddr

#
----- #
# - Read the next chunk of data.
- #
#
----- #
strData = oFile.read(256)
wLen    = len(strData)

#

```

```

----- #
# - Generate the command data. We pack the data using big endian.
- #
#
----- #
strCmdData      = struct.pack('>BHHH%ds' % wLen,
                               OPCODE_UPLOAD,
                               wCurrAddr,
                               wMemType,
                               wLen,
                               strData)

BinCmd.Len      = len(strCmdData)
BinCmd.Channel  = 0
BinCmd.CmdData[:] = strCmdData
BinCmd.SendBlock()

#
----- #
# - Update our current upload address.
- #
#
----- #
wCurrAddr += wLen

#
----- #
# - We are at the end of the file, terminate the file upload.
- #
#
----- #
if wLen < 256:
    oFile.close()
    return False

#
----- #
# - Continue file upload with next timer expiration.
- #
#
----- #
return True

```

You have to save the module as BinFileUpload.py in a directory that is on the Python search path.

The code is generously commented. For simplicity we duplicated much of the code in the fOnStart() and fOnTimer() routines, the command generation could be split off it it's own routine. Any exceptions in the script will be handled by terminating the upload and flagging the error in the GSEOS message window.

We first check the arguments provided by the user, since we want to allow hex arguments

we have to convert the string ourselves. If we had used a range type argument the limit check would have been handled automatically but this limits us to decimal values only. After we have checked our settings we open the file and read the maximum amount of data allowed. If we are at the end of the file we are done with the file upload and return False, otherwise we return True and the upload will continue with the timeout value as specified in the configuration. The fOnTimer() routine then handles clocking out the remaining data. We have to keep the current address as a global since we have to feed this into every file upload command and increment accordingly.

The sample application generates BinCmd blocks which contain binary command data, effectively going around your command definitions. If you do define command mnemonics for your upload commands you might want to consider CmdString blocks containing your command mnemonics. This all depends on the specific instrument command structure.

1.14.7 How do I configure multiple emulators?

It is possible to run multiple instances of GSEOS on one machine and control multiple emulators side by side. However, to have a specific instance address a specific emulator you need to specify the affinity of the GSEOS instance to the particular emulator you want to control.

If you don't care which emulator you use for each instance you don't need to specify the affinity. Say you launch Fields1 you'll get one of the two connected emulators (most likely it's going to be the same every time but that might depend of the USB connection order, etc.).

However, once instrument hardware is connected this is probably not a desirable solution. We can tie a particular emulator to a specific instance of GSEOS by giving the emulator a device ID. You can do this using the FrontPanel application. Say you give the Fields1 emulator the ID: FIELDS1 and the second emulator the ID: FIELDS2. In order to associate the emulator with a specific GSEOS instance you have to put the following entry into the respective gseos.ini file:

For Fields1:

```
[XEM1]
DeviceID = FIELDS1
```

For Fields2:

```
[XEM1]
DeviceID = FIELDS2
```

Now you will always get the correct emulator with the according instance of GSEOS. However, keep in mind that now we have to find an exact match on the device ID. So if you were to connect, say a SWEAP emulator and try to run GSEOS with the Fields configuration you'll get an error since the FIELDS1 emulator can't be found.

The device ID (if configured) is reported in the message Window.

1.14.8 How do I defer the emulator startup?

When implementing scripts in GSEOS you should make sure the scripts are not running for more than a few hundred milliseconds since otherwise the foreground processing will be blocked by your script. This is usually not a problem with common decoders/monitors etc. However, if you need to do some extensive processing on system startup like reading and processing input files or other time consuming tasks this can be done with some precautions.

You can defer the emulator data acquisition which in turn will allow you to block the foreground processing by running a script. To do so you have to set the

```
[SCE]  
StartAcquisition = Manual
```

entry. Once you do so the `SCE_CONFIG_START_ACQUISITION()` command is not sent automatically on startup. You then can issue the command manually once your processing has completed.

1.15 Troubleshooting Guide

If you are running into problems with the emulator there are a few steps you can take to troubleshoot these issues:

a) Linux 64-bit

If you have problems running GSEOS on a Linux 64-bit platform please make sure to install the 32-bit libraries. There are instructions in the [installation section](#) on how to do this.

b) The Error LED on the emulator lights up and the Global Fault Flag on the GSEOS screen is lit.

This can have a number of reasons, most likely this is related to a communication problem between GSEOS and the emulator. The Global Fault is always flagged as soon as any fault is encountered, this flag stays latched and needs to be reset, even if the cause of the error was transient.

The most common cause of this error is a TLM or CMD Timing error. In the case of a TLM error there is most likely no instrument hardware attached or the instrument hardware doesn't generate telemetry.

If the CMD Timing error is lit most likely GSEOS wasn't able to get the command data through to the emulator in a timely fashion. Even if you don't issue any commands and/or there is no Time Update message configured GSEOS sends a message to the emulator periodically. This can have several causes, the most common excessive CPU load like system backups, system updates, virus scanners, etc. Please make sure that these kind of services don't interfere with regular operations.

The other reason might be user scripts. Any Python script that is being run in the foreground in GSEOS will block all other execution until it completes. As a general rule of thumb no script should run for more than a few hundred milliseconds. In general decoders and monitors are very efficient and short lived. If you do need to run long scripts first check if there is a need to do so, most tasks can be broken down into small, short pieces of code. If you need to control GSEOS 'top-down' a STOL script might be the better approach. An alternative is to run your Python scripts in threads. Please make sure to use

proper thread synchronization for global or shared resources. The GSEOS Sequencer is another alternative that allows you to run sequences which don't block foreground processing.

The default configuration allows for a 1s latency, you can change this with the:

[SCE]

MaxCmdQueueLen = NNN

setting. Where NNN is the number of major frames outstanding between GSEOS and the emulator (default is 3). Your latency will be NNN-1 seconds. So if you set NNN to 5 you will have a latency of 4 seconds. Please keep in mind that all your commands will be delayed by that amount as well!

TLM UART fault flags usually show up if there are hardware errors in the underlying bus. Please make sure your grounding scheme between instrument and emulator is sound. We have seen bit-errors on the UART when the instrument is not on common ground with the emulator.

Back Cover