Software Requirements Specification (SRS) and Software Design Document (SDD) for the Core Flight Software Generic Command Ingest / Telemetry Output Applications

Engineering Directorate
Software, Robotics, and Simulation Division

Availability:
JSC & JSC contractor employees and other NASA & NASA contractor employees as required

February, 2016
Revision 1.4

National Aeronautics and
Space Administration
Lyndon B. Johnson Space Center

Verify this is the correct version before use

For correct version go to: *state electronic pathname*

Software Requirements Specifications and Software Design Document for the Core Flight Software Generic Command Ingest/Telemetry Output Applications

Prepared by:

_____

Guy de Carufel, Odyssey Space Research, LLC Date

_____

D. Alan Asp, Odyssey Space Research, LLC Date

Approved by:

_____

SQA Representative's Name *Date*

SQA Representative, *Mail Code*

_____

Lorraine Prokop, Ph.D., Project Manager, Core Flight Software AES Project *Date*

Project Manager, *Mail Code*

_____

Division/Project Approval Authority Name *Date*

Title, Mail Code

Your division chief may choose to delegate signature authority for this document to the Project Manager, per EA-WI-023 Table 7.2.5-1

Verify that this is the correct version before using.

Change Record

Rev.	Date	Originator	Description
1.0	*03/09/2015*	Guy de Carufel Alan D. Asp	Draft release
1.1	*05/06/2015*	Guy de Carufel Alan D. Asp	First Revision
1.2	09/29/2015	Guy de Carufel Alan D. Asp	Revision following design review
1.3	11/13/2015	Guy de Carufel	Addition of requirements to reflect design
1.4	02/09/2016	Guy de Carufel	Application revision pre-release

Table of Content

List of Figures

List of Tables

Verify that this is the correct version before using.

1 Introduction

The software requirements specification (SRS) and software design document (SDD) documents the Computer Software Units (i.e., the CSUs), including their identities, attributes, static relationships, dynamic interactions and design requirements. The SRS/SDD includes enough structural and behavioral information to specify the internal and external behavior of the primitive software components and to be translated straightforwardly into the chosen programming language(s). The SRS/SDD includes information sufficient to enable sustaining engineering of the software by programmers other than the developers. This includes the software structure, module definitions and functionality, high-level interface descriptions, threads of control, major data structures, and important algorithms. Design requirements and the traceability matrix of requirements to CSUs are also presented.

This SRS/SDD defines the requirements and design of the software for the Core Flight Software Generic Command Ingest / Telemetry Output Applications.

1.1 Scope

This Software Design Requirements (SRS) and Software Design Document (SDD) documents the requirements and design of the flight software and firmware for the Core Flight System Generic Command Ingest / Telemetry Output Applications Government Furnished Equipment (GFE) project. This document includes the selected design of the software, including its decomposition into software components, their relationships, and the design of interfaces.

1.2 Responsibility and Change Authority

This document is prepared in accordance with EA-WI-025, “GFE Flight Project Software and Firmware Development”. The responsibility for the development of this document lies with the Software, Robotics, and Simulation Division of the Johnson Space Center. Change authority is the Software, Robotics and Simulation Division of the Johnson Space Center.

2 Related Documentation

The following documents are referenced in this specification. Unless otherwise specified, the exact issue shown is the applicable version.

2.1 Applicable Documents

The following documents, of the exact issue and revision shown, form a part of this SDD to the extent specified herein.

Document Number	Revision/ Release Date	Document Title
EA-WI-025	Rev. D / 09/18/15	GFE Flight Project Software and Firmware Development
NPR 7150.2	09/27/04	NASA Software Engineering Requirements
JSC-61918	Rev. B/ Jan 2014	Core Flight Software (CFS) Advanced Exploration Systems (AES) Project Plan [CFS AES]

2.2 Reference Documents

The following documents are referenced within this SDD. These documents do not form a part of this SDD and are not controlled by their reference herein.

Document Number	Revision/ Release Date	Document Title
GSFC 582-2007-001	Rel. 5.4 / 07/09/14	cFE Application Developers Guide
GSFC 582-2008-012	Rel. 3.0 / 09/01/14	Core Flight System (CFS) Deployment Guide
GSFC Code 582	Rel. 4.1 / 01/17/14	OS Abstraction Layer Library, v4.1
GSFC 582-2007-00	Rel. 4.1 / 01/31/14	Operating System Abstraction Layer (OSAL) Configuration Guide
GSFC 582-2009-xxx	Rel. 1.0 / 09/10/09	Flight Software Unit Test Tool User's Guide
MPCV 70022-05	Baseline / 12/12/12	Orion Multi-Purpose Crew Vehicle (MPCV) Program: Command, Control, Communication, and Information (C3I) Interoperability Standards Book, Volume 5: Data Exchange Specification
CCSDS 232.1-B-2	Issue 2 / 09/10	Recommended Standard for Communications Operation Procedure-1
CCSDS 132.0-b-1	Issue 1 / 09/03	CCSDS Recommendation for TM Space Data Link Protocol
CCSDS 130.0-G-3	Issue 3 / 07/14	Overview of Space Communication Protocols Informational Report
CCSDS 130.1-G-2	Issue 2 / 10/12	TM Synchronization and channel coding – Summary of concept and rationale
CCSDS 230.1-G-2	Issue 2 / 10/12	TC Synchronization and channel coding – Summary of concept and rationale
CCSDS 131.1-B-2	Issue 2/ 09/10	TM Synchronization and channel coding – Recommended Standard
CCSDS 231.1-B-2	Issue 2/ 09/10	TC Synchronization and channel coding – Recommended Standard
CF User's Guide V1.0	August 26, 2011	CFDP (CF) Flight Software User's Guide

2.3 Order of Precedence

In the event of a conflict between the text of this SDD and an applicable document cited herein, the text of this SDD takes precedence.

3 CSCI-Wide Design Decisions

3.1 Background and Context

3.1.1 Core Flight Software

The Core Flight Software (CFS) System as developed by Goddard Spaceflight Center is a layered framework for reuse (Figure 1). It was developed and tested over several years and first deployed for the Lunar Reconnaissance Orbiter (LRO) spacecraft mission in 2009. Since then it has been successfully reused for several other satellite missions, and its reuse has proliferated across other NASA centers and projects. A key feature of this framework is that its layered architecture has proven robust and resilient in adapting to additional software and hardware platforms without change.

The Command Ingest (CI) and Telemetry Output (TO) have thus far been treated as mission specific CFS applications (in yellow). Their purpose is to deal with all data flow in and out of the CFS framework, especially over the Space-To-Ground link. They have been deemed mission specific mainly because of the many permutations possible and the mission specific protocols used to interact with ground control for command inputs, and telemetry outputs. The new AES sponsored effort presented here aims to productize a generic version of these two applications, so that they may be used as a starting point for all missions requiring CI and TO capabilities and become a CFS reusable app (in blue). The selected design to be presented allows for customization for mission specific needs, within a generic pattern. One specific implementation also presented deals with the Multi-Purpose Crew Vehicle (MPCV) Orion mission based Command, Control, Communication, and Information (C3I) data protocols.

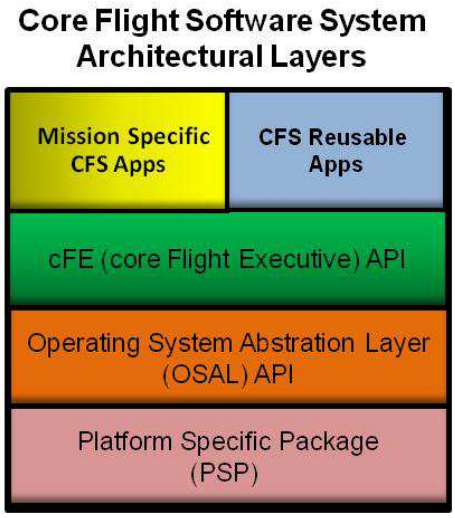

Figure 1: Core Flight Software Framework, Architectural Layers

3.1.2 CCSDS Space Communication Protocols

The CFS was developed with The Consultative Committee for Space Data Systems (CCSDS) Space Communication Protocols in mind. Specifically the Space Packet Protocol (SPP) and the CCSDS File Delivery Protocol (CFDP). In cFE, the Software Bus (SB) module is an implementation of the CCSDS SPP services, as all messages sent over the SB use the SPP for message routing. At the CCSDS Space Communication Protocol application and transport layers, the CFS CF application, a CFS reusable application, is a direct implementation of the CCSDS File Delivery Protocol (CFDP). As such, CFS covers the application, transport and network layers of the CCSDS Space Communication Protocols.

As this document treats the development of applications concerned with the data movement in and out of the CFS, it is important to understand which layer it represents in the context of the CCSDS Space Communication Protocol layered architecture, as shown in Figure 2. In this context, TO and CI provides Data link layer services to the network layer (CFS-Software Bus) as well as some Application Specific Protocol services, such as services for the Orion C3I protocol. CCSDS Telecommand Space Data Link Protocol (TC-SDLP) response services (namely the receiving segment of the Communications Operation Procedure-1 or COP-1) are available as optional CI services, while Telemetry Space Data Link Protocols (TM-SDLP) services are available to the TO application. These services are available in the form of protocol libraries.

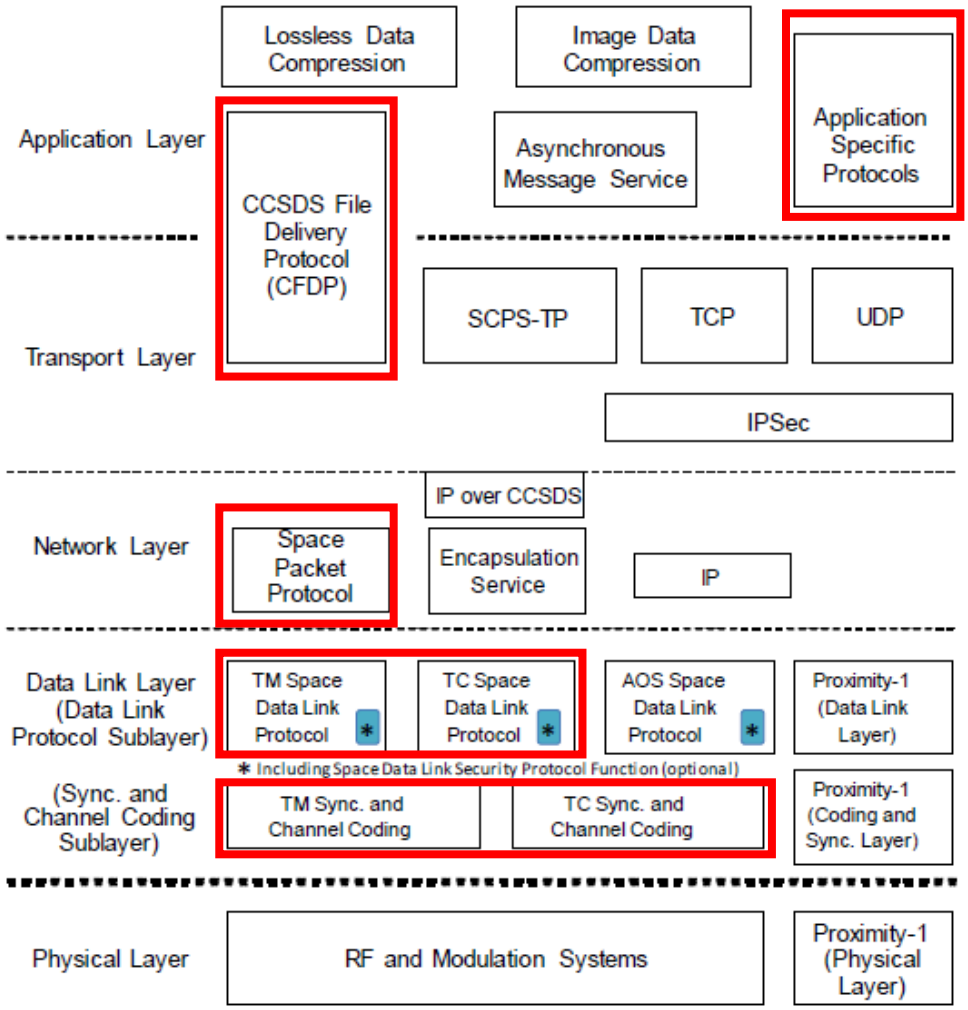

Figure 2-1: Space Communications Protocols Reference Model

Figure 2: The Space Communication Protocols Reference Model [CCSDS 130.0 G-3]

Verify that this is the correct version before using.

3.1.3 Space Data Link Protocols

CCSDS defines two sublayers in the Data Link Layer of the OSI Model. The Data Link Protocol Sublayer is the higher sublayer. This sublayer specifies methods of transferring data units provided by the network layer over a point-to-point space link using Protocol Data Units (PDUs) known as Transfer Frames (TFs) [CCSDS 130.0-G-3]. Two protocols within this sublayer are included in this design effort, namely the TM-SDLP and the TC-SDLP (COP-1 services).

The TM-SDLP is primarily used to send telemetry from a spacecraft to a ground station. The TC-SDLP is primarily used to send commands from a ground station to a spacecraft. Both the TM and TC SDLPs provide three link identifiers used to identify data streams. These identifiers include:

- Transfer Frame Version Number (TFVN), used to distinguish TFs,
- Spacecraft Identifier (SCID), and
- Virtual Channel Identifier (VCID).

The TM-SDLP performs several functions including: generating control information to perform data identification, loss detection, and error detection; transferring variable-length service data units in fixed-length PDUs; multiplexing and demultiplexing of data; and generating and removing idle data to transfer PDUs at a constant rate [CCSDS 132.0-B-1].

The TC-SDLP performs two major functions including: segmentation and blocking of service data units; and transmission control of service data units. Segmenting and blocking provides options for more efficient transfers of service data units. Controlling the transmission of service data units by performing retransmissions as needed ensures the delivery of data units in the proper sequence and without gaps or duplication [CCSDS 232.0-B-2]. The automatic retransmission control mechanism is called the Communications Operation Procedure-1 (COP-1). The receiver reports back to the sender with a Communications Link Control Word (CLCW) describing the status of acceptance of Transfer Frames. The CI application supports COP-1 by implementing the CLCW reporting service. The ground control is responsible for implementing the uplink portion of the COP-1 protocol. For example, the Integrated Test and Operating System (ITOS) implements COP-1 for sending commands.

Both TM and TC SDLP utilize the concept of “Virtual Channels” (VCs). Virtual Channels permit the use of one Physical Channel to relay data among multiple higher-layer data streams.

The second sub-layer is the Synchronization and channel coding sub-layer. Both TC and TM synchronization services are provided in the form of libraries (TC_SYNC, TM_SYNC). The receiving end of the TC_SYNC services is included while the sending

end of TM_SYNC services is included. All channel coding functionality is left as user specific implementation.

The receiving end of the Synchronization allows for encapsulation detection of data units (transfer frames) to detect the start and end of data units, and allows the resolution of data ambiguity through de-randomization of the data [CCSDS 230.1-G-2].

Similarly, the sending end of TM Synchronization allows for encapsulation of transfer frames and pseudo-randomization of its data for ambiguity improvement [CCSDS 130.1-G-2].

3.1.4 MPCV Orion Protocol

The Data Exchange Message (DEM) is a C3I protocol designed to support the exchange of data among NASA's Exploration Systems, e.g. MPCV Orion. DEMs have fields to provide functional context and time, and data integrity. The data carried by each message is defined by message metadata. The DEM protocol is used at the application layer of the OSI model.

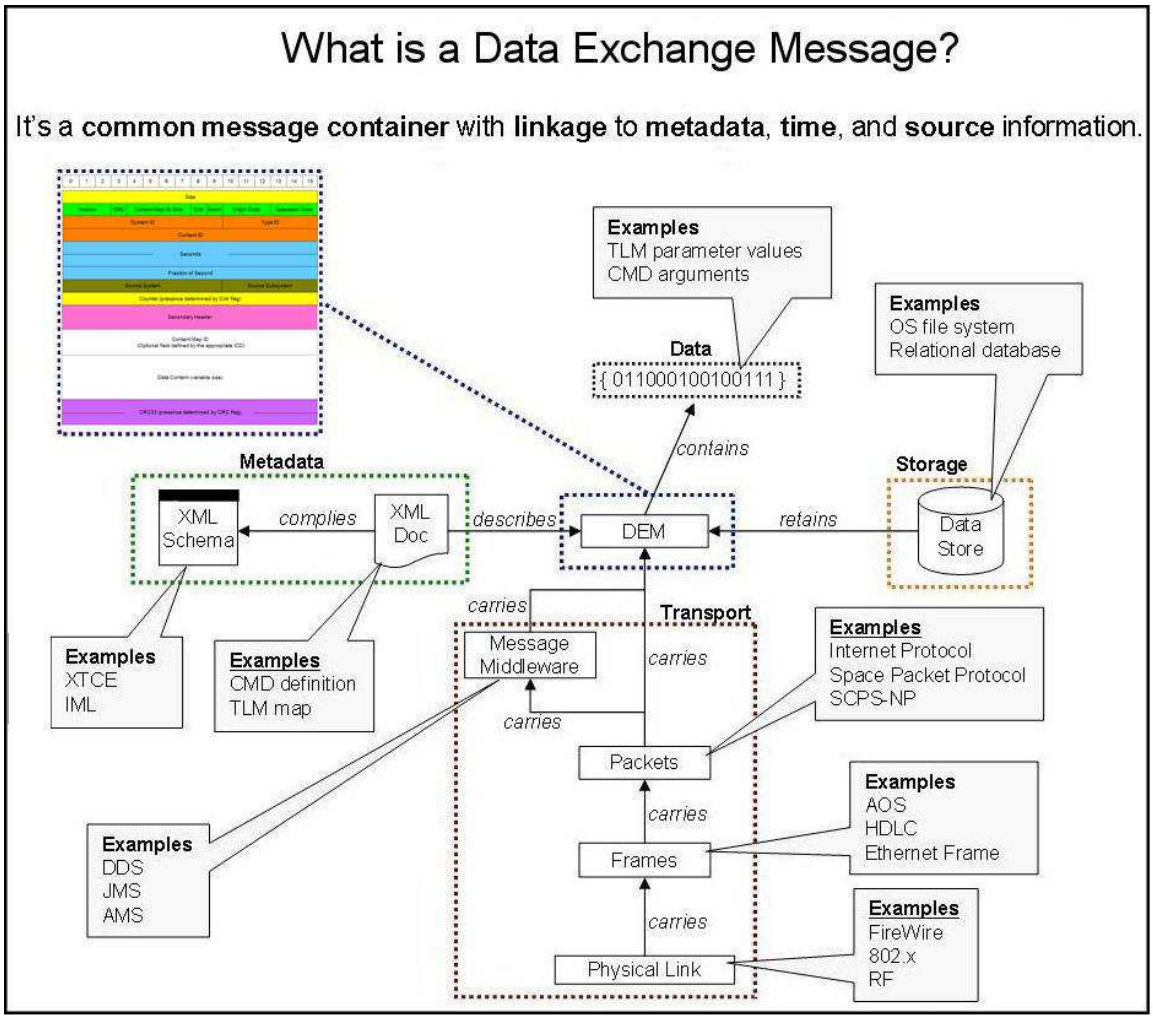

Figure 3: What is a Data Exchange Message? [MPCV 70022-05]

3.1.5 Previous CI & TO Implementations

The Command Ingest (CI) and Telemetry Output (TO) have typically been mission specific applications tailored for the needs of the mission. As the names imply, the CI application is responsible for receiving uplink commands, while TO is responsible for sending telemetry in the downlink. Here, uplink refers to the communication link from a ground station to a space platform operating CFS.

Lab versions of these applications are available, which offer simple UDP interfaces to external sockets. For the JSC Morpheus project, implementations of the CI & TO were developed to support a serial RS-422 interface and support some CCSDS Data Link framing services. Table 1 below describes the different applications available, which are used as reference code for the implementation of the new generic CI and TO applications.

Table 1: Existing CI / TO implementations

Application	Description
CI_lab	"Laboratory" version of the CI application, such that commands are received over a non-blocking UDP socket. Every packet is assumed to be in the CCSDS SPP format.
TO_lab	Similarly to CI_lab, it sends CCSDS – SPP packets over a UDP socket. A convenient feature that is also included is the delivery of a bit packed sample message so that the platform bit order can be assessed.
CI_M	The Morpheus Project version of the CI application. It can receive messages over both UDP sockets and an RS-422 serial interface. It includes support of the COP-1 data link protocol (Type-B only).
TO_M	The Morpheus Project version of the TO application. It can send messages over both UDP sockets and an RS-422 serial interface. It includes provisions for the CCSDS TM-SDLP framing.

3.2 Supported Protocols

For this implementation of the generic CI and TO application, several of the mentioned protocol services and formats are supported, as well as generic transport protocols used for communication over different transport mediums (socket, serial). Note that these "transport protocols" do not correspond to the CCSDS Reference model transport protocol layer, as they provide a transport layer to the data link layer.

The following format application protocol is provided as a library, available to both the CI and TO applications (and any other applications which require DEM format):

- C3I-DEM protocol application format (Used by CI and TO)

The following services are provided in the form of libraries available to both the CI and TO applications:

- TC-SDLP data link service with the COP-1 procedure (used by CI)
- TM-SDLP multiplexed framing data link service (used by TO)

The following transport protocols are available to both CI and TO:

- UDP Socket interface
- RS-422 Serial interface

3.3 Reference Model

For the CI and TO applications, the following reference model represents the different services and application protocols supported. The protocols in dark blue are services and formats that the CI and TO applications access through what is called the IO libraries, while the CFDP services are supported by the CF application. The network layer utilized by the CI and TO applications is the CFS SB represented in green.

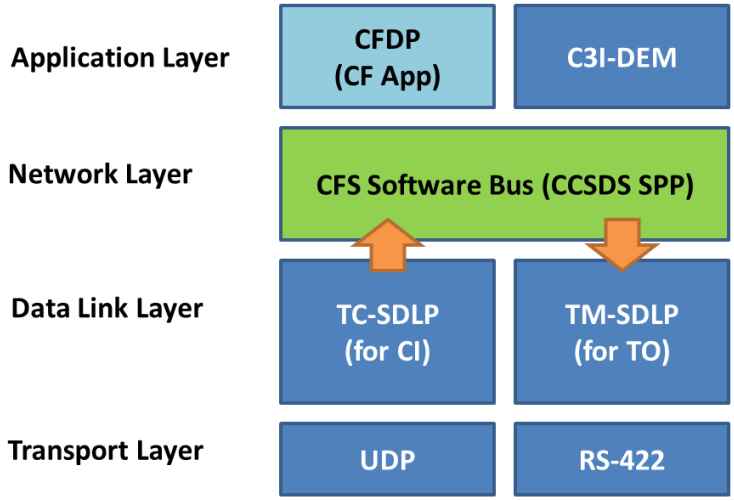

Figure 4: CI / TO Reference Model

4 CSCI Architectural Design

4.1 CSCI Components

The general design of the CI & TO applications is structured such that mission specific customization is confined to one dedicated file (ci_custom.c and to_custom.c), whereas all protocols are provided through a CFS libraries, called the IO_LIB and C3I_LIB. The libraries are divided into two components, services and formats. IO_LIB Services include protocol services accessed by CI/TO, such as TM/TC SYNC, TC-SDLP COP-1 and transport services (UDP, RS422 and Select). C3I_LIB services includes the DEM to CCSDS translation service. The format component includes header manipulation APIs such as the C3I Orion protocols for the C3I_LIB, and the transfer frame protocols (TCTF, TMTF) for the IO_LIB. All components are located in four new directories within the /apps/ CFS mission workspace directory: "/apps/ci", "/apps/to", "/apps/io_lib" and "/apps/c3i_lib". The standard "apps/inc" directory holds public header files for the CI and TO apps (Mission_xx_types.h) including the message structures and command structures. The mission specific CI and TO performance ids, message ids, command ids and command codes are to be added to the respective mission header file in the apps/inc folder.

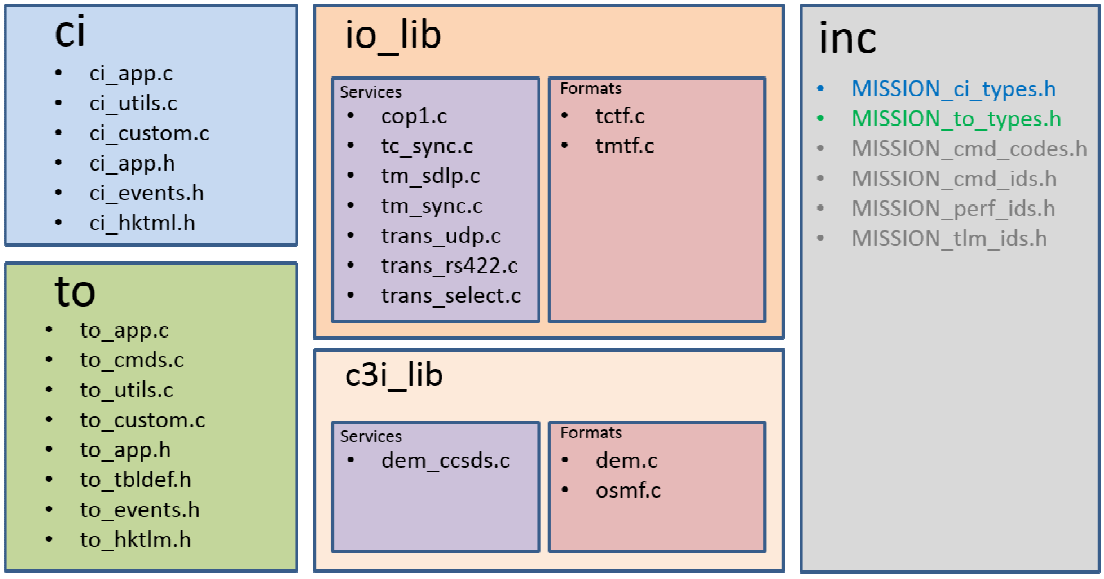

Figure 4: CI/TO and IO_LIB file structure

This approach allows the flexibility required to tailor the application for mission specific requirements while giving access to common reusable libraries. This also easily permits

the addition of new libraries without affecting the architecture. The C3I_LIB is included as a separate library from IO_LIB as it contains proprietary protocols.

The new applications (CI/TO) have the following files in their xx/fsw/src/ directory:

- xx_app.c: This file holds the generic source code of the applications, which make calls to the custom functions defined in xx_custom.c. This file implements the generic CFS application pattern for generic interactions with the cFE core services such as the Software bus and table services, and implements non-mission specific features.
- xx_utils.c: This file holds the definition of utility functions used by other functions. These utility functions are also intended to be available to the custom layer. The prototypes of these functions are defined in the xx_app.h file.
- xx_custom.c: This file defines all mission specific code and implements a generic API called by the xx_app.c functions. These custom functions make calls to the appropriate library functions from the libraries, according to the mission's needs. Any custom functions, and custom structures local to the custom file are declared and defined within this file. Several example implementations are provided within the example directory as described below.
- xx_app.h: This header file holds the app's data structure, as well as public generic function prototypes. This also includes the required custom functions which are called from the xx_app.c file and defined in xx_custom.c. Default definitions of required configuration macros are also included.
- xx_events.h: This file defines the event numbers for the application. An information and error event number is reserved for custom events.
- xx_hktlm.h: This file presents a standard housekeeping packet that is included through the MISSION_xx_types.h file if the default structure is desired. If a custom Housekeeping packet is desired, the #include is omitted and the custom structure is defined in the MISSION_xx_types.h file.

The TO application have these additional files:

- to_cmds.c: This file holds the definition and local prototypes of the response to generic TO app commands. Any custom commands are handled through functions in the TO_custom.c file.
- to_tbldefs.h: This file defines the TO configuration table for message subscription and routing.
- to_cmds.h: This file describes the prototype and message structures of the standard TO commands. Any additional custom commands are defined in the MISSION_to_types.h.

Two additional files are included in the CI/TO apps directory (xx/fsw) for mission and platform configurations:

- platform_inc/xx_platform_cfg.h: This file holds all configuration definitions such as transport protocol configurations, buffer sizes, limits and pipe depths.
- mission_inc/xx_mission_cfg.h: This file simply includes all mission specific configuration files defined in the apps/inc (see below).

In the apps/inc folder, several files include structures and macro definitions used by the CI and TO applications for a specific mission (e.g. "MISSION"). The include files are:

- MISSION_xx_types.h: These files contains the definition of custom command and telemetry message structures used by the applications (xx = ci or to). It is customizable based on the mission and the messages expected by the custom layer implementations of the CI and TO applications.
- MISSION_cmd_ids.h: This file includes the definition of all command message ids used by all applications in the mission, including CI and TO. All command message IDs must be unique.
- MISSION_cmd_codes.h: This file includes the definition of all command codes ids used by all applications in the mission, including CI and TO. Command codes are specific to an application, and can be reused between applications. Mission specific custom command codes for CI and TO are added here.
- MISSION_tlm_ids.h: This file includes the definition of all telemetry message ids used by all applications in the mission, including CI and TO. All telemetry message IDs must be unique.
- MISSION_perf_ids.h: This file includes the definition of all performance monitoring ids used by all applications in the mission, including CI and TO. All performance monitoring IDs must be unique. Additional IDs may be added here to provide profiling granularity.

In the apps/{ci or to}/fsw/examples directory, several implementation examples are provided as sub-directories, including:

- udp: An example implementation of a single UDP channel with support of CCSDS headers, with similar capabilities as the ci_lab and to_lab applications. Makes use of the trans_udp service.
- rs422: An example implementation of CCSDS packets received and transmitted over a duplex rs422 serial port. Makes use of the trans_rs422 service.

- multi: An example implementation of CCSDS packets transmitted over two transport channels used: RS422 serial port and UDP sockets. The transport services used include trans_udp, trans_rs422 and trans_select.
- multi_tf: This example implements two channels (UDP and RS422) with transfer frames as the data unit. CI make use of the TC SYNC and COP-1 services while the TO make use of the TM SYNC and TM SDLP services. The transport services used include trans_udp, trans_rs422 and trans_select.
- udp_dem: This example makes use of the DEM-CCSDS translation service library to translate C3I formatting protocols to the CCSDS SPP protocol over two UDP sockets. A third UDP route is included for normal CCSDS messages (without DEM translation) to demonstrate support of multiple formats. This example also demonstrates the concept of TO message routing for multiple output channels. Note that this example requires the C3I_LIB library.
- demo: This example was used as a demonstration example with 3 routes: CCSDS over udp, DEM over udp, and CCSDS over rs422.

For each example, three files are included:

1. xx_custom.c: The example's application custom implementation. To use in a build, copy the file to the apps/xx/fsw/src directory.
2. MISSION_xx_types.h: The example's custom message structures. It includes the xx_hktlm.h file by default, making use of the default Housekeeping packet. To use in build, copy this file to the apps/inc directory.
3. xx_platform_cfg.h: The examples configuration macro definitions. To use in a build, copy the file to the xx/fsw/platform_inc directory.

A convenient bash script has been included for both CI and TO apps to copy or link the desired example files to the appropriate locations. This script is located in the xx/fsw/examples directory. To use the script, execute the following bash command:

./setup.sh {example}

Use the –l option to create symbolic links instead of making copies.

4.2 Concept of Execution

4.2.1 Command Ingest

The Command Ingest (CI) Application is responsible for receiving commands from an external source (such as a ground station) over a transport channel, and to forward the command to the appropriate application over the cFE Software Bus (SB).

4.2.1.1 CI Gate command

The CI application makes use of a special command called the CI Gate command. So that the CI application may respond promptly to custom critical commands meant for I/O control, a CI Gate command (with the message id: CI_GATE_CMD_MID) is to be captured and processed immediately by CI when received, and is not sent to the software bus. This also provides the ability to capture critical commands which are not to be passed to the other applications, such as critical arming commands. The specific command codes and responses to the Gate command are mission specific and implemented within the custom layer of the CI application.

4.2.1.2 Message control

To ensure that messages are always received, CI does not have generic provisions to disable particular messages from being received.

4.2.1.3 Uplink data flow

The general data flow for uplink through the CI application follows the sequence:

1. The ci custom task pends on the receipt of a new packet through API calls to the IO_LIB service transport protocols (UDP, RS422 or Select).
2. Once received, the ci_custom task calls the necessary service APIs (IO_LIB, C3I_LIB) to manipulate the received data as required.
3. On reception of the CI_GATE_CMD_MID message, the appropriate command is executed based on the command code in the message.
4. Any other messages are forwarded to the software bus for eventual use by other applications.

4.2.2 Telemetry Output

The Telemetry Output (TO) Application is responsible for transmitting telemetry to external destination(s) (such as a ground station) over transport devices(s). A configuration table determines the routing of messages such that route specific telemetry pipes are subscribed to specified messages, and such messages are processed through route specific protocol stacks when received from the software bus.

4.2.2.1 TO Message Routing

The TO application makes use of a configuration table in order to route messages. A route is defined as a particular set of operations to execute on specified data before being transmitted over a specific transport channel, also described as a “protocol stack”. In practice this can equate to a sequence of IO_LIB format and service calls and/or custom data manipulation.

A route is a sequence of operations which a packet is subjected to according to its message id, as specified in TO's configuration table. The specific operations related to a route are implemented by the user within the custom segment of the TO application. The application configuration table maps specific message ids to route(s) through the usRouteMask parameter.

It is up to the mission specific implementation of the custom segment of the TO application to determine the proper behavior, according to the route. The TO configuration table is reconfigurable through the standard cFE Table services and through several available TO commands.

Functionality is also included to set certain message IDs as critical, such that the configuration table is only valid if it contains all messages that are deemed critical by the user.

Each route owns a separate CFE SB telemetry pipe for memory isolation between routes. When a route is specified as present, a dedicated pipe is created during initialization and is subscribed to all messages that are routed to that pipe according to the configuration table.

Functionality is also included to set the processing rate of specific routes to a wakeup period with respect to the overall TO application processing rate. This allows different routes to process their associated pipe messages at different rates.

Routing was not implemented for the CI application, as the protocols associated with messages received over specific transport channels are already known, and so no additional routing strategy is required beyond the segregation already provided by the distinct input channels.

4.2.2.2 Message control

TO can enable/disable specific messages from being transmitted through its message ID or through the use of a parameter called the uiGroupData, which allows multiple table entries to be enabled/disabled through a single command.

The uiGroupData parameter allows a message to belong to a specific group, or a multi-group, according to the TO configuration table. With the uiGroupData, a command can then enable/disable several table entries or reconfigure the usRouteMask of these entries.

4.2.2.3 TO downlink data flow

The general data flow for downlink through the TO application follows the sequence:

1. Each telemetry pipe of existing routes receives messages it is subscribed to according to the configuration table.
2. The TO task is awakened by the scheduler through the WAKEUP message or through a timeout, at which point it reads its telemetry pipes.

3. For each pipe of enabled routes, all new messages present for which the message id is enabled is then processed at the custom layer according to the associated route. The custom function performs the necessary data manipulation for eventual downlink.
4. The message is then sent over the appropriate IO transport channel for downlink.

4.3 Interface Design

Figure 5 illustrates the relationship between the CI/TO applications, the library services, the CFS Software Bus and any other applications. The CI application is composed of two tasks: the main task which processes CI commands and sends the housekeeping report, and the child task that processes incoming uplink commands. The custom layer of the CI application (ci_custom.c) is responsible for the CI child task, while the application layer (ci_app.c) is responsible for the scheduling of the main task.

The TO application only has one task and is scheduled by a wakeup message from the software bus (or through a timeout). The application layer of the TO application (to_app.c) is responsible for interacting with CFE services (table configuration, software bus, event registration) while the custom layer of the TO application (to_custom.c) is responsible for processing route specific messages to downlink. Several command responses are included in the TO application, processed within the application layer (to_cmds.c).

For both application, the concept of the “application layer” and the “custom layer” is simply a distinction between files which are fully implemented and considered static (xx_app.c, xx_utils.c, xx_cmds.c), and the mission specific implementation custom file (xx_custom.c).

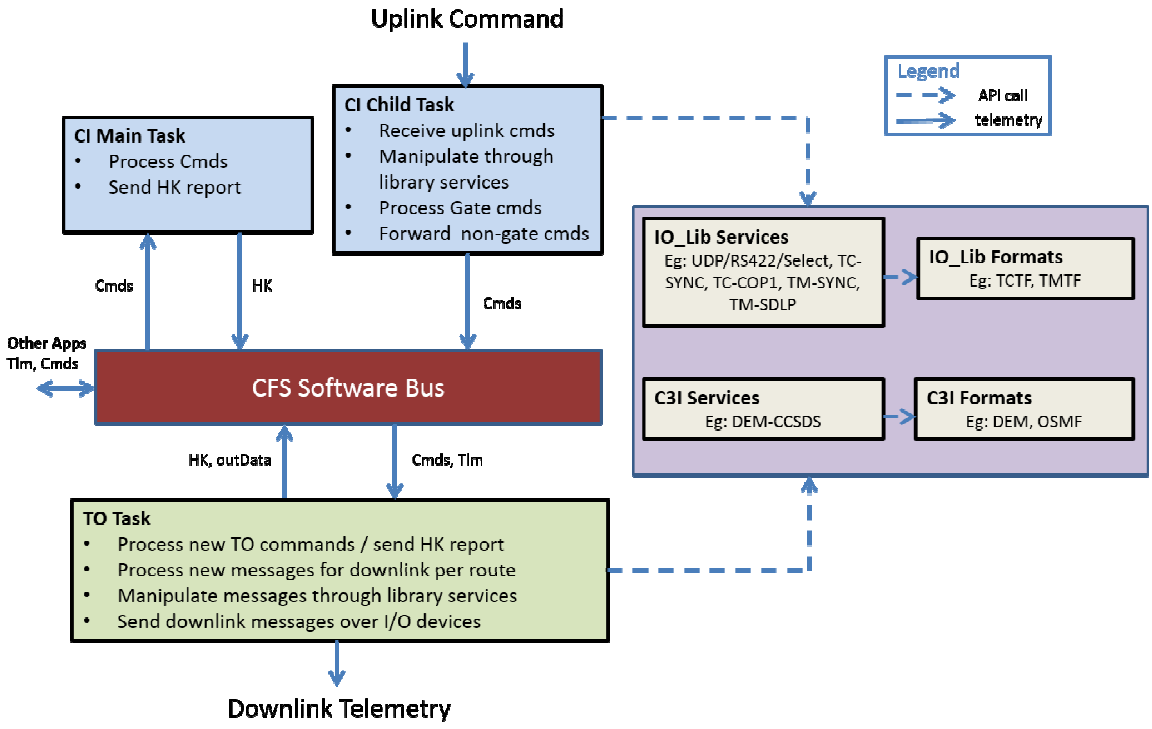

Figure 5: CI/TO data flow and execution architecture

5 CSCI Detailed Design

5.1 Command Ingest (CI)

The CI application is divided into two layers, the Application layer and the Custom layer. Each layer is responsible for specific interactions. The implementation of both layers is divided into two main source files: ci_app.c and ci_custom.c. The application layer controls the main application task, while a child task is created for the custom layer. This division is necessary because the application layer operates synchronously with the SCH application, while the custom layer operates synchronously with incoming commands.

5.1.1 CI Application Layer

The CI Application layer is responsible for interacting with the cFE services, including the cFE Software Bus (SB), and Event Services (EVS). Its interaction with the custom layer is through the CI_CustomInit, the CI_CustomEnableTO and the CI_CustomAppCmds functions. This layer is implemented within the ci_app.c source file. Its main function is to initialize the application, send the housekeeping telemetry and forward custom commands to the custom layer.

5.1.1.1 Application Layer Functions

The functions for the CI app layer include the following:

Table 2: CI Application Layer Functions

Function	Description
CI_AppInit	Calls functions: CI_InitEvent, CI_InitPipe, CI_InitData, CI_InitTable, followed by CI_CustomInit, which is implemented in the custom layer.
CI_InitEvent	Initialize the CI events with the cFE Event services. This includes initializing the event filter table and registering with Event services.
CI_InitPipe	Initialize the CI pipes (Schedule, Command) and subscribe the respective pipes with the appropriate messages. The Schedule pipe is subscribed to the Wakeup sent from the SCH application. The command pipe is subscribed to the CI_APP_CMD_MID and the CI_SEND_HK_MID.
CI_InitData	Initialize the housekeeping packet.

CI_RcvMsg	Blocks on the schedule pipe, processes new application commands (CI_ProcessNewCmds), and sends out data (CI_SendOutData) on Wakeup, or on timeout.
CI_ProcessNewCmds	Polls the command pipe for any new application command and executes CI_ProcessNewAppCmds. Executes CI_ReportHousekeeping on a CI_SEND_HK_MID message.
CI_ProcessNewAppCmds	Executes the appropriate command response based on the command code received. Calls the CI_CustomEnableTO on the CI_ENABLE_TO_CC. For any other custom commands, calls the CI_CustomAppCmds function.
CI_SendOutData	Sends out data to the software bus. Out data is populated in the custom layer.
CI_ReportHousekeeping	Sends the Housekeeping packet to the software bus.
CI_AppMain	Calls CI_AppInit, then loops on CI_RcvMsg. Calls CI_CustomCleanup before termination.

5.1.1.2 Application Commands

The following commands are supported. The command codes correspond to the application command message: CI_APP_CMD_MID.

Table 3: CI Application Commands

Command Code	Description
CI_ NOOP_CC	No-operation command. Increment command counter.
CI_ RESET_CC	Reset the housekeeping packet.
CI_ ENABLE_TO_CC	Call the CI_CustomEnableTO (custom layer) command to enable the TO application.

5.1.1.3 Application Default Settings

The following default values are defined (with #ifndef condition) in the ci_app.h file. These may be overwritten in the ci_platform_cfg.h file through #define.

Table 4: CI Application Default Settings

Macro	Default	Description
CI_WAKEUP_TIMEOUT	1000 (ms)	The wakeup timeout value. Set this to schedule the ci application without CI_WAKEUP_MID

CI_SCH_PIPE_DEPTH	10	The pipe depth of the scheduler pipe before messages are dropped.
CI_CMD_PIPE_DEPTH	10	The pipe depth of the command pipe before commands are dropped
CI_CUSTOM_TASK_STACK_PTR	NULL	Fourth input to CFE_ES_CreateChildTask()
CI_CUSTOM_TASK_STACK_SIZE	0x4000	Fifth input to CFE_ES_CreateChildTask()
CI_CUSTOM_TASK_PRIO	118	Sixth input to CFE_ES_CreateChildTask()

5.1.1.4 Main Task Scheduling

The scheduling of the main task for the CI application is performed by the SCH application by sending the CI_WAKEUP_MID. A wakeup timeout value can also be set (CI_WAKEUP_TIMEOUT), such that the CI application may be executed at a constant rate, without receiving the CI_WAKEUP_MID. Note that a SEND_HK_MID must still be sent by the SCH application in order for the CI application to report the HK message. The CI_SEND_HK_MID should be sent at an equal or lower rate compared to the CI_WAKEUP_MID, or the wakeup timeout rate.

5.1.2 CI Custom Layer

This layer is responsible for creating and managing the ci child task, and pend on incoming uplink commands over mission specific IO transport services (UDP, RS-422). It must perform all necessary framing (eg: TC-SDLP, TM-SDLP) and do all necessary format translation on received uplink commands such that commands sent to the SB are CCSDS SPP packets. It is also responsible for responding to custom commands received over the application command pipe, executed by the main task. The custom layer is implemented in ci_custom.c. Several example implementations are provided for interactions with various libraries.

5.1.2.1 Custom Layer Required Functions

The following functions must be implemented in ci_custom.c. Table 5 are functions which are executed on the main task and are called from the application layer (ci_app.c). Table 6 are required functions which are executed on the child task to process uplink commands.

Table 5: Required CI Custom Layer Functions (Executed by Main Task)

Function	Description
CI_CustomInit	Initializes the transport protocols and other library specific initializations, and creates the custom main child task through the CFE_ES_CreateChildTask function.
CI_CustomEnableTO	This function is responsible for sending the enable command to the TO application, with the necessary information, based on the specific transport protocol. This function is called by the application layer in function CI_ProcessNewAppCmds on receipt of the CI_ENABLE_TO_CC.
CI_CustomAppCmds	This function is responsible for dealing with custom app commands (if any) and is called by the CI_ProcessNewAppCmds for any non-generic command codes with message id CI_APP_CMD_MID. Returns CI_ERROR if the command code is not recognized.
CI_CustomCleanup	Cleanup functions such as closing transport I/O devices.

Table 6: Required CI Custom Layer Functions (Executed by Child Task)

Function	Description
CI_CustomMain	This is the main implementation of the custom layer, where data received is processed to be sent over the SB. See next section for example implementation.
CI_CustomGateCmds	This function is responsible for dealing with any gate commands (with message id CI_GATE_CMD_MID), after reception from external I/O channel, not destined for the SB.

5.1.2.2 Custom Commands

The command code of any mission specific custom commands can be added in the MISSION_cmd_codes.h file and have the response function declared and defined in ci_custom.c. The command message structure is to be defined in the MISSION_ci_types.c file. Custom commands can have one of two message ids:

1. CI_APP_CMD_MID custom commands are received over the application command pipe and are sent to the CI_CustomAppCmds via the CI_ProcessNewAppCmds. These commands are executed by the CI app main application task, and so are meant for commands sent by other CFS applications. The execution of these commands is in sync with the scheduler WAKEUP message.

2. CI_GATE_CMD_MID custom commands are identified at "the gate", meaning that they are immediately processed after receiving them over the I/O transport device. On receipt of these commands, the CI_CustomGateCmds function is called, and the appropriate custom local functions are then executed. These commands are to be executed by the custom layer child task.

5.1.2.3 Custom Layer Configuration

The following are defined in the app/inc folder mission header files.

- MISSION_cmd_codes: Definition of custom command codes for the CI_APP_CMD_MID and CI_GATE_CMD_MID.
- MISSION_ci_types: Definition of custom command message structures (including for the CI_CustomEnableTO and all other custom commands). Structure definition of output data (CI_OutData_t) and HK telemetry (CI_HkTlm_t).

The following are defined in the ci_platform_cfg.h file:

- Overwrite of default macro settings
- Custom macro definitions for custom layer managed settings

The following are implemented in the ci_custom.c file:

1. Allocation of local message buffers
2. Declaration and definition of local custom functions.
3. Definition of all required custom functions (as in Table 14).

5.1.2.4 CI_CustomInit Pattern

1. Initialize the IO_LIB transport protocol(s) to use, e.g. IO_TransUdpInit
2. Create the custom child task with CI_CustomMain as the entry point.

5.1.2.5 CI_CustomMain Pattern

The CI_CustomMain should follow this general pattern:

1. Block on transport protocol (UDP socket, serial port or select)
2. Call any other IO_LIB data link protocols or framing service calls, such as TC-SDLP COP-1 (if required).
3. Call any custom command authentication library (if applicable/implemented).
4. Call appropriate format translation services (eg: dem-ccsds.c) to translate the received Message header to an SPP Message header (if required).

5. Validate the CCSDS checksum of the command through CFE_SB_ValidateChecksum.
6. See if the message id is the CI_GATE_CMD_MID, and if so, call the CI_CustomGateCmds function with the received message.
7. Send any other messages to the software bus.

5.1.2.6 Multi-Channel Pattern

If CI is to handle multiple input channels, such as messages from multiple serial ports and/or sockets, the easiest approach is to use the same pattern as above with the provided IO_LIB trans_select.c API. This API allows pending on multiple sockets/serial-ports at once with a specified timeout.

Alternatively, the pattern can be extended by adding an additional child task for each transport channel. Additional multi-threaded protection may be required.

5.1.3 CI Mutex

As both tasks must have access to the housekeeping packet and output data, a mutex called the ciMutex is used to protect memory due to the multi-threaded nature of CI. Both tasks (main and child) must lock/unlock this mutex whenever the housekeeping packet and the output data packet are accessed. A utility function is included to manipulate HK counters in a thread-safe manner.

5.1.4 CI Utilities

Through the ci_utils.c file, a few utilities are provided, which can be used by both the application and custom layers. All utilities are presented in Table 7.

Table 7: CI Utilities

Function	Description
CI_IncrHkCounter	Increment a provided housekeeping counter. Memory protection through the ciMutex is performed.
CI_VerifyCmdLength	Verify the length of a received command compared to the expected length of the command. This function must be called on reception of all commands. Issues an error event and increments the housekeeping usCmdErrCnt if the cmd length verification fails.

5.2 Telemetry Output (TO)

As in CI, the TO application is divided into two conceptual layers: application and custom. Unlike CI, both layers operate on a single cFE task, as all functions must be

synchronized with the SCH application, or set to a constant rate through a wakeup timeout.

5.2.1 TO Application Layer

This layer is responsible for interacting with the cFE services, including the cFE Software Bus (SB), the Event Services (EVS) and the Table Services (TBL). Its interaction with the custom layer is through the TO_CustomInit, TO_CustomProcessTlm, and through TO_CustomAppCmds. The application layer is implemented within the to_app.c source file along with the to_cmds.c for application command responses and utilities from to_utils.c.

5.2.1.1 Application Layer Functions

The functions for the TO app layer include:

Table 8: TO Application Layer Functions

Function	Description
TO_AppInit	Calls functions TO_InitEvent, TO_InitPipe, TO_InitData, TO_InitTable, followed by TO_CustomInit, which is implemented in the custom layer.
TO_InitEvent	Initializes the TO events with the cFE Event services. This includes initializing the event filter table and registering with Event services.
TO_InitPipe	Initializes the TO pipes (Schedule, Command, Telemetry) and subscribes the respective pipes with the appropriate messages. See below for more details on telemetry pipes. The Schedule pipe is subscribed to the Wakeup to be sent from the SCH application. The command pipe is subscribed to the TO_APP_CMD_MID and TO_SEND_HK_MID.
TO_InitData	Initializes the housekeeping packet.
TO_InitTable	Registers the configuration table with Table services.
TO_ValidateTable	Validate the TO configuration table.
TO_RcvMsg	Blocks on the schedule pipe, processes new application commands (TO_ProcessNewCmds) and processes telemetry (TO_ProcessTlmPipes) on a Wakeup message.
TO_ProcessNewCmds	Polls the command pipe for any new application command and calls TO_ProcessNewAppCmds, implemented in to_cmds.c. Calls TO_ReportHousekeeping on a SEND_HK

	message.
TO_ProcessTlmPipes	Loop over every existing telemetry pipe (one per existing route) and call TO_ProcessNewData if the current wakeup count corresponds to the route's wake period.
TO_ProcessNewData	Polls the telemetry pipe for any new telemetry and calls TO_CustomProcessData (if output is enable and active), which is implemented in to_custom.c.
TO_ReportHousekeeping	Sends the Housekeeping packet to the software bus.
TO_AppMain	Calls TO_AppInit, and then loops on TO_RcvMsg. Calls TO_CustomCleanup before termination.

5.2.1.2 Application Commands

The following commands are processed through the TO_ProcessNewAppCmds function defined in the to_cmds.c file. The command codes correspond to the application command message: TO_APP_CMD_MID. Any other command code received is assumed to be a custom command, handled by the TO_CustomAppCmds function.

Table 9: TO Application Commands

Command Code	Description
TO_APP_NOOP_CC	No-operation command. Increment command counter.
TO_APP_RESET_CC	Reset the housekeeping packet.
TO_ENABLE_OUTPUT_CC	Call the TO_CustomEnableOutputCmd to configure and enable routes. Sets usOutputEnabled and usOutputActive to true.
TO_DISABLE_OUTPUT_CC	Call the TO_CustomDisableOutputCmd to disable output of specific route(s). Sets usOutputEnabled to false.
TO_ACTIVATE_ROUTES_CC	Enable all routes by supplied usRouteMask. A route must be configured to be enabled.
TO_DEACTIVATE_ROUTES_CC	Disable all routes by supplied usRouteMask.
TO_PAUSE_OUTPUT_CC	Pause transmission of all downlink (set usOutputActive to false)
TO_RESUME_OUTPUT_CC	Resume transmission of all downlink (set usOutputActive to true)
TO_ADD_TBL_ENTRY_CC	Add an entry to the config table
TO_REMOVE_TBL_ENTRY_CC	Remove a table entry by message id.
TO_ENABLE_ MSG_CC	Enable a table entry in the config table by MID
TO_DISABLE_MSG_CC	Disable a table entry in the config table by MID
TO_ENABLE_GROUP_CC	Enable a group in the config table
TO_DISABLE_GROUP_CC	Disable a group in the config table
TO_ENABLE_ALL_CC	Enable all used table entries
TO_DISABLE_ALL_CC	Disable all used table entries
TO_SET_ROUTE_BY_MID_CC	Set the usRouteMask of a table entry by MID
TO_SET_ROUTE_BY_GROUP_CC	Set the usRouteMask of multi entries by uiGroupData
TO_MANAGE_TABLE_CC	Hook for cFE Table services
TO_SET_ROUTE_PERIOD_CC	Set the wakePeriod value for routes. Value must be a factor of TO_MAX_WAKEUP_COUNT.

TO_SET_WAKEUP_TIMEOUT_CC	Set the TO wakeup timeout value. Must be larger or equal to TO_MIN_WAKEUP_TIMEOUT.
Other	Handled by TO_CustomAppCmds

5.2.1.3 Application Default Settings

The following default values are defined (with #ifndef condition) in the to_app.h file. These may be overwritten in the to_platform_cfg.h file through #define.

Table 10: CI Application Default Settings

Macro	Default	Description
TO_WAKEUP_TIMEOUT	500 (ms)	The wakeup timeout value. Set this to schedule the TO application without TO_WAKEUP_MID
TO_MIN_WAKEUP_TIMEOUT	500 (ms)	The lowest value that the wakeup timeout can be set to.
TO_MAX_WAKEUP_COUNT	20000	The value at which the wakeup count rolls back to 0. This value will affect valid wakePeriod values that a route may be set to.
TO_SCH_PIPE_DEPTH	10	The pipe depth of the scheduler pipe before messages are dropped.
TO_CMD_PIPE_DEPTH	10	The pipe depth of the command pipe before commands are dropped
TO_NUM_CRITICAL_MID	0	The number of critical message IDs that must be present in the configuration table.
TO_GROUP_NUMBER_MASK	0xFF000000	The bytes dedicated to the group number within the GroupData.
TO_MULTI_GROUP_MASK	0x00FFFFFF	The bytes dedicated to the multi-group within the

		GroupData.
TO_CONFIG_TABLENAME	"to_config"	The name of the config table
TO_CONFIG_FILENAME	"/cf/apps/ to_config.tbl"	The name of the config file of the configuration table.
TO_NUM_CF_CHANNELS	1	Number of CF channels
TO_CF_THROTTLE_SEM_NAME	"CFTOSemId"	The name of the CF counting semaphore of CF channel 0.

5.2.1.4 Application Scheduling

The scheduling of the main task for the TO application is performed by the SCH application by sending the TO_WAKEUP_MID. A wakeup timeout value can also be set (TO_WAKEUP_TIMEOUT), such that the TO application may be executed at a constant rate, without receiving the TO_WAKEUP_MID. Note that a SEND_HK_MID must still be sent by the SCH application in order for the TO application to report the HK message. The TO_SEND_HK_MID should be sent at an equal or lower rate compared to the TO_WAKEUP_MID, or the wakeup timeout rate. The wakeup timeout value can be changed at run time through the TO_SET_WAKEUP_TIMEOUT_CC command.

Each route may be set as a period of TO application wakeups through the use of the route's usWakePeriod (See 5.2.1.7).

5.2.1.5 Telemetry Output Configuration

Through the cFE Table Services, the following table is available to assist the custom implementation of TO. Modifying this table can be accomplished through the cFE Table services and through several TO commands.

Table 11: TO Message Configuration Table

Type	Name	Description	Example
CFE_SB_MsgId_t	usMsgId	Message ID	0x0880
CFE_SB_Qos_t	qos	Quality of service	{0,0}
uint16	usMsgLimit	Maximum number of message instance in telemetry pipe	1
uint16	usRouteMask	The routing of this message. ex: 0x0003 => Routes: 0 and 1	0x0003
uint32	uiGroupData	Group data for multi table entry commanding (see below)	0x01000000

uint16	usFlag	Custom flag (implementation specific)	0
uint16	usState	Current state (working parameter)	1

The message configuration table defines the subscription parameters (usMsgId, qos, usMsgLimit), the route path(s) this message is subject to according to the usRouteMask, and whether it is enabled by the usState flag. The uiGroupData allows multiple table entries to be modified at once through commands. The usFlag parameter can be used by the custom layer, and can have any custom meaning.

A maximum of 16 routes can be used, each one of which corresponds to a bit in usRouteMask. A message can use multiple routes.

For example usRouteMask = 00000000 00000101 indicates that this message will be processed by both routes 0 and 2, according to the index of bits with 1.

The uiGroupData follows the same pattern as used in the SCH application GroupData strategy, with the Most-Significant Byte representing the Group ID, and the 3 remaining bytes representing the multi-group mask, as shown in Table 12.

Table 12: uiGroupData 32-bits

Byte:	1 (MSB)	2	3	4
	Group ID	Multi-Group Mask		

A maximum of 256 group ID (2^8) can be defined, and a maximum of 24 (3x8bits) multi-groups can be defined. A table entry can be set to one group ID and a maximum of 24 multi-groups through its uiGroupData parameter. To set a uiGroupData to a group ID and multi-groups, the OR operator (|) can be used.

Example: uiGroupData = TO_GROUP_ONE | TO_MGROUP_1 | TO_MGROUP_2

5.2.1.6 Table Validation

The function TO_ValidateTable is called whenever a new configuration table is loaded through the CFE TBL services. A table is valid if it passes the following conditions:

1. All used table entries are continuous without gaps (No entries with TO_UNUSED_ENTRY before a used entry).
2. No message ids are repeating.
3. All Critical Messages IDs are present in the table (see section 5.2.3)

5.2.1.7 Route Configuration

An array of 16 routes is stored in the TO application data (TO_AppData_t) defined in to_app.h. Each route is of type TO_Route_t, as described below:

Table 13: TO Route Configuration (Type TO_Route_t)

Type	Name	Description	Default
uint16	usWakePeriod	A multiple of the TO rate when the route should be processed. Process the route if this is true: usWakePeriod % TO Wakeup Count = 0	1
int16	sCfChnlIdx	The index of the CF Channel array used by this route. Must be unique. (see 5.2.1.9)	-1
uint16	usExists	If this route exists / is present. This is set within TO_CustomInit() for all routes that can be used.	0
uint16	usIsConfig	If this route is configured (eg: I/O device opened). Use utility function to set: TO_SetRouteAsConfigured(<index>)	0
uint16	usIsEnabled	If this route is enabled and processing output data. Should only be set via commands (TO_ENABLE_OUTPUT_CC and TO_ACTIVATE_ROUTES_CC)	0

5.2.1.8 Telemetry Pipes

For each route that has usExists = 1 (set in TO_CustomInit()), a new telemetry pipe will be created within TO_InitPipe(). Each of these new pipes will be subscribed to all messages within the configuration table (Table 11) which has a usRouteMask which includes the route associated with this pipe. The following pipe parameters may be set in TO_CustomInit():

- usTlmPipeDepth: Set the depth of the pipe, corresponding to the maximum number of messages that this pipe can hold before additional messages are dropped. All pipe's depth are left to the default value of TO_TLM_PIPE_DEPTH if not explicitly set in custom layer.
- cTlmPipeName: The custom implementation may set the name of each pipe to describe the type of message it should listen to. Otherwise, the default value is "TO_TLM_PIPE_[0-15]". This name must be unique.

The pipes are processed in ascending order, from 0 to 15, when present.

5.2.1.9 CF Channels

The CFDP application (CF), which implements the CFDP CCSDS protocol (see 3.1.2), must be configured with the TO application for message throttling. For each CF channel, a counting semaphore is created within TO so that CF does not overrun the associated TO telemetry pipe. A route is thus associated with a specific CF channel through the route's sCfChnlIdx, which must be unique. The number of CF Channels to consider within TO is set with the TO_NUM_CF_CHANNELS macro, which can be defined in to_platform_cfg.h. By default, only one CF channel is configured (named: "CFTOSemId"). For each additional CF channel, the parameter cCFCntSemName must be set in the TO_CustomInit(), and must correspond to a channel entry in the CF configuration table within the CF application [See CF User's Guide]. This name must be unique.

5.2.2 TO Custom Layer

The TO custom layer is responsible for interacting with the mission specific IO_LIB services in order to transmit downlink telemetry. It is implemented in to_custom.c. This layer consists mostly of the definition of custom functions called from the application layer.

5.2.2.1 Custom Layer Required Functions

The following functions must be implemented in to_custom.c. Functions in Table 15 must be defined if framing is enabled by defining TO_FRAMING_ENABLED in to_platform_cfg.h.

Table 14: TO Custom Layer Functions

Function	Description
TO_CustomInit	Initialize the transport protocol(s), and configure the critical MIDs, route, pipe and cf channels.
TO_CustomAppCmds	Process custom commands (called by TO_ProcessNewAppCmds in to_cmds.c)
TO_CustomProcessData	Process all telemetry, and apply operations according to the route that message is subject to. Operations include IO_LIB services, custom data manipulation and finally transmission over the I/O transport protocol if sending each packet individually.
TO_CustomCleanup	Process cleanup functions such as closing transport devices.
TO_CustomEnableOutputCmd	This is the custom implementation called by

	TO_ProcessNewAppCmds on the command TO_ENABLE_OUTPUT_CC. It configures the transport protocol based on the input parameters for specified route(s).
TO_CustomDisableOutputCmd	This command implements the disabling of specific routes or all output on a TO_DISABLE_OUTPUT_CC command.

Table 15: TO Custom Layer Framing Functions

Function	Description
TO_CustomFrameStart	This function is called once per route per wakeupPeriod from TO_ProcessNewData() prior to processing messages on the telemetry pipe. It may be used to prepare and start a new frame before individual packets are added through TO_CustomProcessData()
TO_CustomFrameSend	This function is called once per route per wakeupPeriod from TO_ProcessNewData() after all messages have been processed from the telemetry pipe. It should complete the frame and send it over the route specific I/O device.

5.2.2.2 Custom Commands

Any custom command codes can also be added in MISSION_cmd_codes.h file and have the response function declared and defined in to_custom.c. The TO_APP_CMD_MID must be used for these commands. The TO_CustomAppCmds is responsible for calling the appropriate response function on receipt of such custom command codes. The structure of these new commands should be defined in the MISSION_to_types.h file.

5.2.2.3 Custom Layer Configuration

The following are defined in the app/inc folder mission header files.

- MISSION_cmd_codes: Definition of custom command codes for the TO_APP_CMD_MID.
- MISSION_to_types: Definition of custom command message structures, output data (TO_OutData_t) and HK telemetry message structure (TO_HkTlm_t).

The following are defined in the to_platform_cfg.h file:

- Configuration parameters of transport devices through macro definitions

- Non-default application settings through macro definitions (see 5.2.1.3).

The following are implemented in the to_custom.c file:

1. Allocation of local message buffers
2. Declaration and definition of local custom functions
3. Configuration of criticalMIDs, route, telemetry pipes and CF channels
4. Definition of required custom functions (Table 14).

5.2.2.4 TO_CustomInit Pattern

The TO_CustomInit function should follow this general pattern:

1. Initialize the IO_LIB transport protocol(s) to use, (e.g. IO_TransUdpCreateSocket)
2. Set the criticalMid array for all critical messages
3. Set all routes to be used as usExist = 1
4. Associated the CF channels to the corresponding routes
5. Configure telemetry pipe depth and name (if default not appropriate)
6. Configure CF channel names (if channels beyond first default channel is used)
7. Configure any translation libraries (eg. dem_ccsds translation table)

5.2.2.5 TO_CustomProcessData Pattern

When packet framing is not used, the TO_CustomProcessData should follow this general pattern based on the route ID supplied:

1. Perform any required format translation (eg. dem-ccsds)
2. Send the message over the route specific transport channel (UDP, RS422, SB)
3. Verify that the sent message size is accurate, or report an “incomplete message sent” error otherwise.

Note that the cFE Software Bus (SB) may act as “transport protocol” if another application is responsible for transmission over another medium.

5.2.2.6 TO_CustomEnableOutputData Pattern

The TO_CustomEnableOutputData should follow this general pattern for each used route:

1. Type Cast the input command message as TO_EnableOutputCmd_t *
2. Check the input usRouteMask argument and proceed for the appropriate configuration of the applicable routes

3. Configure the associated transport protocol, making appropriate library API calls (eg. IO_TransUdpSetDestAddr), making use of parameters included in the message
4. (Optional) Activate the route if configuration was successful and update the usActiveRoutes Housekeeping Packet.
5. Return the route mask.

Note that the exact implementation may deviate from this pattern, for example:

- The command may accept a specific Route ID (0-15) instead of a Route mask (0x0001-0xffff), such that only one route may be enabled at a time.
- The route may be left un-activated after successfully completing this command, such that a separate command (TO_ACTIVATE_ROUTES_CC) must be called to activate after being enabled here.
- The command may be set to enable all routes, without a routes parameter included.
- If the implementation has only a single route, the usRouteMask parameter may be omitted, such that transmission is solely controlled by the route-agnostic usDownlinkEnabled and usTransferEnabled parameters.

The route mask returned is used in the calling function (TO_EnableOutputCmd) to enable the associated routes and update the usRouteEnabled Housekeeping parameter. It is also responsible to enable both the global TO downlink (usDownlinkEnabled) and transmission (usOutputActive) flags.

5.2.2.7 Packet framing pattern

When packet framing is enabled when TO_FRAMING_ENABLED is defined, the following pattern should be used:

TO_CustomFrameStart() (Called once per route wake period):

- Initiate a new frame, which may include copying packets from an overflow buffer and setting appropriate frame headers.

TO_CustomProcessData() (Called on every message on telemetry pipe):

- Perform message format translation (if appropriate)
- Add the packet to the frame buffer, or an overflow buffer if frame is full

TO_CustomFrameSend() (Called once per route wake period):

- Complete the frame, by setting frame headers as appropriate
- Perform frame synchronization / encoding

- Send the frame to over the route's I/O device

The "multi_tf" example makes use of this pattern with the use of the TM-SYNC and TM-SDLP libraries.

5.2.3 TO Critical MIDs

Critical MIDs are used to ensure that certain MIDs are always included in the configuration table. This prevents inadvertently updating the configuration table without these critical MID. Within platform config, the TO_NUM_CRITICAL_MIDS must be defined, and each entry in g_TO_AppData.criticalMid array must be defined in TO_CustomInit().

5.2.4 TO Utilities

Through the to_utils.c file, utilities are provided, which can be used by both application and custom layers. Table manipulation utilities are typically only used by the to_cmds.c functions but may also be used by the custom layer. All utilities are presented in Table 16.

Table 16: TO Utilities

Function	Description
TO_FindEmptyTableIndex	Find an empty table index in the configuration table.
TO_FindTableIndex	Find a table index by message id.
TO_SetStateByGroup	Set the state (enable / disable) by group / multi-group.
TO_SetRouteByGroup	Set the usRouteMask by group / multi-group.
TO_SetAllEntryState	Set the state (enable / disable) of all used entries.
TO_SetRouteAsConfigured	Set a route as configured and update HK usConfigRoutes.
TO_SetRouteAsUnconfigured	Set a route as unconfigured and update HK usConfigRoutes.
TO_DisableRoute	Disable route and update HK usEnabledRoutes parameter.
TO_ValidateRouteMask	Confirm that all routes included in routeMask exists.
TO_GetRouteMask	Get the routeMask by table index.
TO_GetMessageID	Get the message ID by table index.
TO_VerifyCmdLength	Verify the length of a received command compared to the expected length of the command. This function must

	be called on reception of all commands. Issues an error event and increments the housekeeping usCmdErrCnt if the cmd length verification fails.
TO_SubscribeAllMsgs	Subscribe all messages to all appropriate telemetry pipes according to existing routes and the configuration table.
TO_SubscribeMsg	Subscribe a specific message to all appropriate telemetry pipes according to existing routes and the configuration table.
TO_UnsubscribeAllMsgs	Unsubscribe all messages to all appropriate telemetry pipes according to existing routes and the configuration table.
TO_UnsubscribeMsg	Unsubscribe a specific message to all appropriate telemetry pipes according to existing routes and the configuration table.

5.2.5 TO Housekeeping Packet

The following is the default Housekeeping packet for the TO application. Note that it is normally included (#include "../to/fsw/src/to_hktlm.h") in the MISSION_to_types.h. If not included, the TO_HkTlm_t may be defined in MISSION_to_types.h such that the default structure be extended with other custom parameters. Note that a custom implementation of the TO_HkTlm_t structure must include the following parameters.

Table 17: TO Housekeeping Telemetry

Function	Description
usCmdCnt	A command counter incremented on the receipt of any valid commands
usCmdErrCnt	A counter incremented on the any command errors
usMsgSubCnt	The number of subscribed messages to the telemetry pipe corresponding to the number of table entries in the configuration table.
usMsgSubErrCnt	Counter to any errors related to message subscription during a table update.
usTblUpdateCnt	Counter indicating number of full table updates
usTblErrCnt	Counter of errors associated with table updates
usConfigRoutes	A 16-bit Hex value indicating which routes are configured
usEnabledRoutes	A 16-bit Hex value indicating which routes are enabled

5.2.6 Configuration Example

Here is an example of initialized configuration of routes, pipes and CF channels, and a partial representation of a TO configuration table. This state would represent the application data parameter values once initialization is complete.

Route Structure [Array of 16]

Set Through TO_CustomInit()			Set in to_custom.c	Set through CMDs
usWakeupPeriod	**sCfChnlIdx**	**usExists**	**usIsConfig**	**usIsEnabled**
1	0	1	0	0
1	1	1	0	0

Telemetry Pipe Structure [Array of 16]

Set by CFE	Can overwrite in TO_CustomInit() [Default: TO_TLM_PIPE_<IDX>]	
cfePipeId	**usTlmPipeDepth**	**cTlmPipeName**
23	20	"Critical Msg Pipe"
24	10	"TO_TLM_PIPE_1"

CF Channel Structure [Array of TO_NUM_CF_CHANNELS]

Enabled by default	Set by CFE	Overwrite in TO_CustomInit() [Default: "CFTOSemId"]
usIsEnabled	**uiCfCntSemId**	**cCfCntSemName**
1	12	"CFTOSemId"
2	13	"CFTOSemId2"

Config Table – Set through CFE Table and TO Commands

usMsgId	Qos	usMsgLimit	usRouteMask	uiGroupData	usFlag	usState
0x0880	{0,0}	1	0x0003	0x01000000	0	1
0x0881	{0,0}	1	0x0001	0x01000001	0	1

…

Figure 6. TO Configuration example

In the above example, TO_CustomInit() would look like this:

```
g_TO_AppData.routes[0].usExists = 1;
g_TO_AppData.routes[1].usExists = 1;
g_TO_AppData.routes[0].sCfChnlIdx = 0;
g_TO_AppData.routes[1].sCfChnlIdx = 1;
```

```
g_TO_AppData.tlmPipes[0].usTlmPipeDepth = 20;
strncpy(g_TO_AppData.tlmPipes[0].cTlmPipeName, "Critical Msg Pipe",
        OS_MAX_API_NAME);

strncpy(g_TO_AppData.cfChnls[1].cCfCntSemName, "CFTOSemId2",
        OS_MAX_API_NAME);
```

The configuration would result in two telemetry pipes being created in TO_InitPipe(): "Critical Msg Pipe" and "TO_TLM_PIPE_1" with a depth of 20 and 10 respectively. Route 0 would be associated with CF Channel index 0 with a counting semaphore "CFTOSemId" and route 1 would be associated with CF Channel index 1 with a counting semaphore "CFTOSemId2".

Based on the configuration table, both telemetry pipes would be subscribed to message 0x0880 based on the usRouteMask of that table entry, while only the first pipe would be subscribed to 0x0881. Both messages would be processed once telemetry output is enabled (through TO_ENABLE_OUTPUT_CC) as their state is initialized as enabled (usState = 1).

5.2.7 TO Potential Enhancements

Some enhancements have been discussed and could be implemented in the future

1. Move common operations that are frequently performed in the various custom layer functions into a library-like set of functions to reduce code repetition. The maintenance benefit will have to be weighed against the possible obfuscation that it will add to the user. This also applies to the CI application.
2. Add support for multiple configuration tables such that mission specific configuration tables could also be registered with the CFE Table services through the TO application. Each table could then be loaded and managed by the application layer, and take advantage of the "Parameter" variable in the CFE_TBL_NotifyByMessage API to receive a notification message when any of the registered table is updated. The added complexity of this feature should be weighed against the likelihood of this feature being used.

5.3 I/O Library (IO_LIB)

The Input/Output Library is a collection of protocol libraries to be called by the CI/TO application custom implementations. These libraries are divided into two main groups, consistent with the general OSI Model designations:

- Formats:
  - Space Data Link Protocol Transfer Frames (TCTF, TMTF)
- Services:
  - Transport Protocols (UDP, RS-422, Select)
  - Telecommand Space Data Link Protocol services (TC COP-1)
  - Telecommand Synchronization services (TC-SYNC)
  - Telemetry Space Data Link Protocol services (TM-SDLP)
  - Telemetry Synchronization services (TM-SYNC)

5.3.1 Transport Protocol Services

5.3.1.1 UDP Socket transport protocol

The UDP transport protocol services permits the opening of connectionless blocking sockets. The following functions are available:

Table 18: UDP Transport Protocol Services

Function	Description
IO_TransUdpInit	Create a socket, and set a timeout value. Set the socket IP and port, and bind the socket to the port.
IO_TransUdpCreateSocket	Create a UDP Datagram (DGRAM) socket.
IO_TransUdpConfigSocket	Configure the UDP Socket
IO_TransBindSocket	Bind the socket for incoming messages
IO_TransUdpCloseSocket	Close the socket
IO_TransUdpSetDestAddr	Set the destination address for outgoing
IO_TransUdpRcvTimeout	Receive messages over a socket with a

	timeout through the use of Select
IO_TransUdpRcv	Receive messages over a blocking socket
IO_TransUdpSnd	Send messages over a socket to a specified destination address.

The IO_TransUdpRcvSelect function should be used within ci_custom.c if it is the only channel used. The timeout value can be set to block forever by passing timeout=-1. When several channels are used, the IO_TransSelect library should be used instead with a call to IO_TransUdpRcv once the IO_TransSelect function indicates that a message is pending.

The library makes use of the POSIX API and is compatible with the Linux and VxWorks operating systems.

Refer to Appendix C for a specific example in the use of a UDP socket API.

5.3.1.2 RS-422 Transport Protocol

The RS-422 transport protocol services allow duplex access to serial ports. The following public functions are available:

Table 19: RS-422 Transport Protocol Services

Function	Description
IO_TransRS422Init	Open a serial port and set options (baudrate, non-canonical controls)
IO_TransRS422Close	Close the specified file descriptor
IO_TransRS422ReadTimeout	Through select, wait on port for new data with timeout. Calls IO_TransRS422Read on new data available.
IO_TransRS422Read	Read the data on a serial port immediately.
IO_TransRS422Write	Write data to serial port by specified file descriptor

Similarly to the UDP library detailed above, the IO_TransRS422ReadTimeout should be used for a CI implementation with a single incoming channel. A multiple input channel implementation should use IO_TransRS422Read along with the IO_TransSelect library.

The library is compatible with both the Linix and VxWorks operating systems.

Refer to Appendix C for examples of a single and multi-channel implementation.

5.3.1.3 Select Transport Protocol

The Select transport protocol is a wrapper service around both the UDP socket and RS-422 serial port APIs, such that one can block with a timeout on multiple devices at once. This service is only useful for the CI application. The following public functions are available:

Table 20: Select Transport Protocol Services

Function	Description
IO_TransSelectClear	Clear all devices from the select set
IO_TransSelectAddFd	Add a device / file descriptor to the set
IO_TransSelectRemoveFd	Remove a specific file descriptor from set
IO_TransSelectFdInFull	Check if File descriptor is in set
IO_TransSelectFdInActive	Check if File descriptor is the currently active set
IO_TransSelectInput	Call the select library for incoming devices
IO_TransSelectOutput	Call the select library for outgoing devices

The IO_TransSelect is used primarily by the CI application to pend on multiple channels. The sequence of use is as followed:

- Initialize All devices through IO_TransUdpInit and IO_TransRS422Init
- Add all device file descriptors to the Select set through IO_TransSelectAddFd
- At run time, pend on all devices through IO_SelectInput. The library will set whichever device has ready data as active.
- Call IO_TransSelectFdInActive for each device and perform the appropriate receive function (IO_TransUdpRcv, IO_TransRS422Read) when the specific file descriptor is found to be active.

For outgoing, the Select library may be used if a message should be sent to the first available device (as opposed to all devices). The sequence of use is as followed:

- Initialize All outgoing devices to obtain their file descriptors
- Add all device file descriptors to the Select set through IO_TransSelectAddFd
- At run time, pend on all devices through IO_SelectOutput. The library will set whichever device is ready to transmit as active.
- Call IO_TransSelectFdInActive for each device and perform the appropriate send function (IO_TransUdpSend, IO_TransRS422Write) when the specific file descriptor is found to be active.

Refer to Appendix C for an example of a CI implementation with RS422 for incoming data.

5.3.2 Data Link Protocols

5.3.2.1 CCSDS TC-SDLP (TCTF)

To support implementation of the COP-1 procedure, a set of functionality was developed to handle incoming TC-SDLP transfer frames. The low level capability to build transfer frames from incoming codeblocks is provided as a service of the TC Synchronization and Channel Coding sublayer protocol (TC-SYNC). The implementation of the transfer sublayer of the data link protocol sublayer provides accessors for the identification of the Master Channel ID and the Virtual Channel ID as well as the COP-1 functionality. The implementation of the segmentation sublayer of the data link protocol sublayer provides accessors for the Multiplexer Access Point (MAP) ID and the sequence flags. The relationship between the CCSDS protocols, sublayers and channels is illustrated in Figure 7.

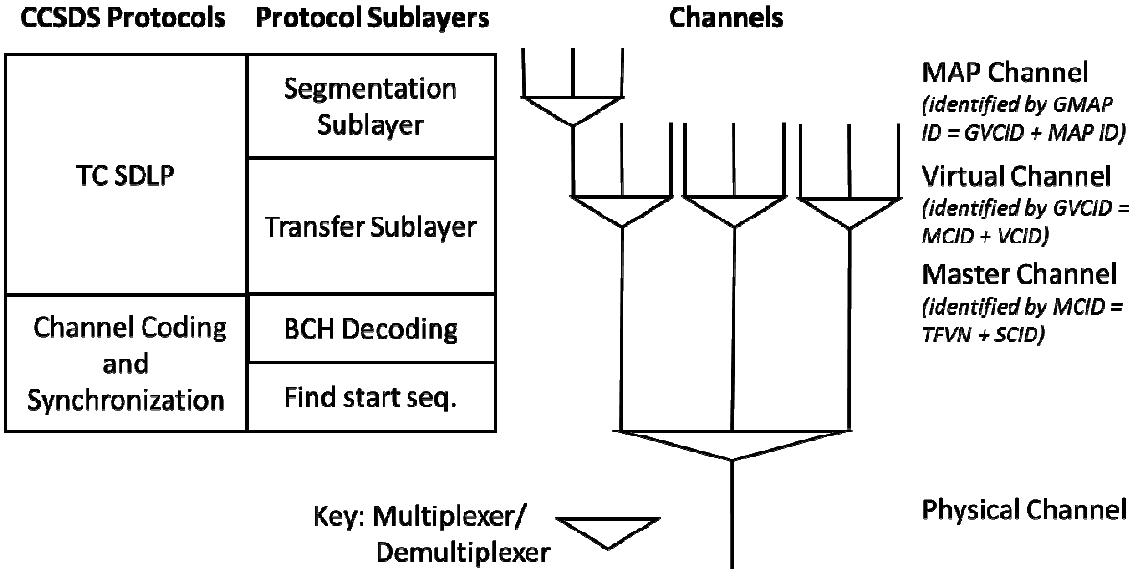

Figure 7: Relationship between CCSDS Protocols, sublayers, and channels

The TCTF code leverages code from Morpheus and the Lunar Reconnaissance Orbiter (LRO). However, the design of the API is significantly different. For the Morpheus/LRO design, successful processing of a protocol sublayer resulted in a call to a function embedded within that function to process the next higher protocol sublayer. The calling frame stack was twelve levels deep by the time a CCSDS command was put on the SB.

By contrast, this design is modular. Each function performs its designated operations and returns to the calling function for subsequent processing. With the implementation of the COP-1 functionality, a typical user will not need to call any of these functions directly. Instead, the COP1 main entry point, COP-1_ProcessFrame, will access the TCTF provided functionality as needed.

The TF API hides the details of the message structure. Implementations of CI should not attempt to access a TF field directly, as the structure is subject to change.

For each function in the API, a pointer to the TF is passed as an argument. All pointers are checked if NULL. As the receiving end of the TF, CI does not need to set any TF values, so only accessor functions are provided. A NULL pointer results in a return value of zero, or a value indicative of no data.

The TF consists of a primary header, an optional segmentation header, the TF data unit, and an optional Frame Error Control Field. The use of the segmentation header is determined prior to the mission, and, if used, must be used on all TFs carried on a Virtual Channel. The use of the Frame Error Control Field is determined prior to the mission, and, if used, must be used on all TFs carried on the same physical channel throughout a Mission Phase. The decoding of the Frame Error Control Field has not been implemented as part of this development effort.

Internally, bit manipulations are implemented as macros for efficiency, but the macros are not part of the TCTF API. For copying the payload data from the TF, the TCTF_CopyData function uses the CFE_PSP_MemCpy function. No other CFS functionality is used within the API.

The TC transfer frame API is presented in Table 21.

Table 21: TC Transfer Frame (TCTF) API

CSU	Description
TCTF_GetVersion	Returns the TF version number
TCTF_GetBypassFlag	Returns the TF Bypass flag
TCTF_GetCtlCmdFlag	Returns the TF Control Command flag
TCTF_GetScId	Returns the TF spacecraft ID
TCTF_GetVcId	Returns the TF Virtual Channel ID
TCTF_GetLength	Returns the TF length in number of octets
TCTF_GetSeqNum	Returns the TF Frame Sequence number
TCTF_GetSegHdrSeqFlags	Returns the TF Sequence flags from the segment header, if included
TCTF_GetSeqHdrMapId	Returns the MAP ID from the segment header, if included
TCTF_GetPayloadLength	Returns the TF payload length in number of octets
TCTF_CopyData	Copies the TF data field to a specified buffer

To provide functionality to handle variations in frame information, a structure is provided for the user to specify information on the TC service type, spacecraft ID, virtual channel ID, MAP ID, and inclusion of the frame error control field. For each type of channel and service type, the user shall create an instance of this structure and populate the fields appropriately. Several TCTF functions utilize this structure as an argument. The COP1_ProcessFrame function also takes a pointer to an instance of the structure as an argument.

5.3.2.2 CCSDS TC-SDLP (COP-1)

The Communications Operation Procedure-1 (COP-1) is a close-loop procedure consisting of a pair of synchronized procedures for each Virtual Channel. The sending entity executes a Frame Operation Procedure-1 (FOP-1), while the receiving entity executes the Frame Acceptance and Reporting Mechanism-1 (FARM-1). As the receiving entity, CI has access to the the FARM-1 capability through the COP1 library. This capability leverages the code used by Morpheus and the Lunar Reconnaissance Orbiter (LRO).

The COP1_ProcessFrame function is the entry point for FARM-1 processing of each Transfer Frame (TF). This processing consists of:

1. verifying the virtual channel ID matches the expected ID
2. checking the value of the TF's 'Bypass' Flag;
3. checking the value of the TF's 'Control Command' Flag;
4. checking the value of the frame sequence number for Type-AD TF;
5. Updating the Communication Link Control Word.

The 'Bypass' Flag controls whether the acceptance check is performed (Type-AD TFs). If the 'Bypass' Flag is set, and the 'Control Command' Flag is set (Type-BC TF), the TF carries control commands for configuring COP-1 ('Unlock' and 'Set V(R)'). If the TF is a Type-BC TF, the FARM-1 state is not 'Lockout', and the 'Unlock' command is sent, the FARM-1 state is moved from the 'Lockout' state. If the TF is a Type-BC TF, and the 'Set V(R)' command is sent, the expected sequence number is set to the specified value.

When the 'Bypass' Flag is not set, the TF is a Type-AD TF. In this case, an acceptance check is performed on the TF, namely checking the TF Frame Sequence Number against the expected sequence number.

- If the sequence numbers agree the frame is incorporated in the service data unit buffer, and the Communications Link Control Word (CLCW) is updated indicating the acceptance of the TF.

- If the sequence numbers disagree, and the difference is outside the sliding window width, the frame is discarded and the CLCW lockout flag is set, i.e. the FARM-1 state is 'Lockout'. In this case, CI no longer accepts additional Type-AD TFs until the receipt of an 'Unlock' Control Command.
- If the sequence numbers disagree, the difference is inside the sliding window width, and the TF sequence number is greater than the expected sequence number, then the frame is discarded and the CLCW Retransmit Flag is set indicating a TF has been lost in transmission.
- If the sequence numbers disagree, the difference is inside the sliding window width, and the TF sequence number is less than the expected sequence number, then the frame is discarded and the CLCW is not updated The TF has already been received and processed previously.

In order to provide a scheme for flow control between FARM-1 and the Higher Procedures, i.e. the function which calls the COP1_ProcessFrame function, the destination buffer pointer is used to signal the availability of buffer space. When the calling function lacks enough buffer space to handle the maximum expected TF data unit, a NULL pointer shall be passed as the destination buffer argument. During the TF processing, if a Type-AD TF arrives with the expected sequence number and the destination buffer is NULL, the TF will be discarded and the 'Wait' flag will be set in the CLCW. For a Type-BD TF, the data will also be discarded, as there is no destination to copy the data to, but the 'Wait' flag will not be set. All other TF processing proceeds nominally.

After processing the transfer frame, if the CLCW was updated, it is sent to TO for downlink as the Operational Control Field (OCF) of the Telemetry TF.

COP1_InitClcw function uses the CFE_PSP_MemSet function. No other CFS functionality is used within the API.

The COP-1 function APIs are presented in Table 22.

Table 22: COP-1 API

CSU	Description
COP1_InitClcw	Initializes the CLCW by zeroing all octets, then setting the COP in Effect and virtual channel ID fields
COP1_GetClcwCtrlWordType	Returns the control word type of the CLCW
COP1_GetClcwVersion	Returns the version number for the CLCW
COP1_SetClcwStatus	Sets the status for the CLCW
COP1_GetClcwStatus	Returns the status for the CLCW
COP1_GetClcwCopEffect	Returns the COP in Effect value for the CLCW
COP1_GetClcwVcId	Returns the virtual channel identifier for the

CSU	Description
	CLCW
COP1_SetClcwNoRf	Sets the 'no radio frequency available' flag for the CLCW
COP1_GetClcwNoRf	Returns the 'no radio frequency available' flag for the CLCW
COP1_SetClcwNoBitlock	Sets the 'no bitlock' flag for the CLCW
COP1_GetClcwNoBitlock	Returns the 'no bitlock' flag for the CLCW
COP1_GetClcwLockout	Returns the lockout flag for the CLCW
COP1_GetClcwWait	Returns the wait flag for the CLCW
COP1_GetClcwRetransmit	Returns the retransmit flag for the CLCW
COP1_GetClcwFarmbCtr	Returns the FARM-B counter for the CLCW
COP1_GetClcwReport	Returns the Report Value for the CLCW
COP1_ProcessFrame	Entry point for performing the frame acceptance and reporting for a transfer frame.

5.3.2.3 CCSDS TC-SYNC

A set of function is provided through the TC-SYNC library to perform the synchronization operations as specified in CCSDS 231.0-B-2. This sending end is responsible for breaking frames into equal sized modified Bose-Chaudhuri-Hocquenghem (BCH) codeblocks and encapsulating each into a Communication Link Transmission Unit (CLTU) data structure, as shown in Figure 8. The specification also includes the option for the code block to be randomized.

The TC-SYNC library provided implements the receiving end de-encapsulation and de-randomization services, such that full TC transfer frames can be extracted from a CLTU, which can then then processed through the TC-COP1 service.

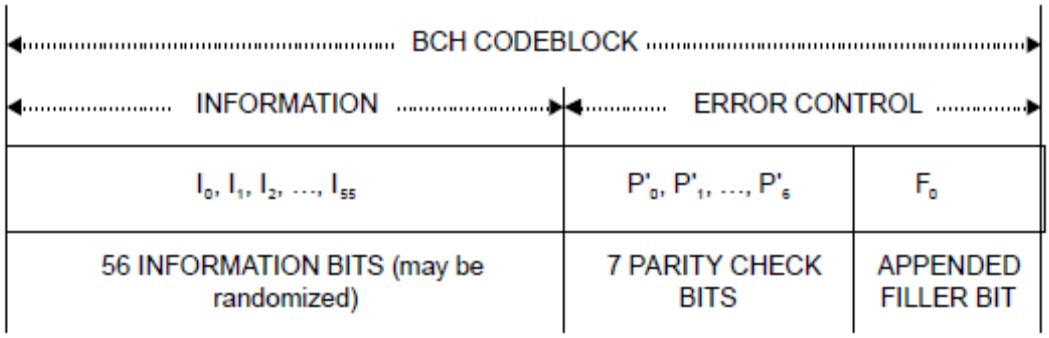

Figure 3-1: BCH Codeblock Format

Verify that this is the correct version before using.

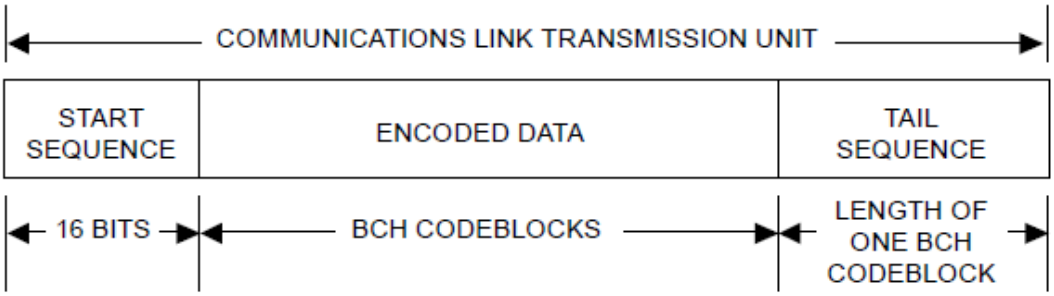

Figure 4-1: Components of the CLTU

Figure 8: TC-SYNC Data Unit

The TC-SYNC function APIs are presented in Table 23.

Table 23: TC-SYNC API

CSU	Description
TC_SYNC_LibInit	Initializes the static pseudo-random sequence
TC_SYNC_GetTransferFrame	Extract a TC transfer frame from a CLTU
TC_SYNC_CheckStartSeq	Verify the start sequence of a CLTU
TC_SYNC_GetCodeBlockData	Get TF data from a CLTU code block.
TC_SYNC_DeRandomizeFrame	Perform the standard CCSDS channel de-randomization of the code block.

When a full CLTU can be processed at once, as when it is received over a UDP connection, the TC_SYNC_GetTransferFrame function should be called. When the CLTU is received incrementally over a serial connection, the piecewise functions should be called as followed:

1. Call TC_SYNC_CheckStartSeq to confirm the reception of a CLTU
2. Call TC_SYNC_GetCodeBlockData for every codeblock received until a tail sequence code block is found
3. Call TC_SYNC_DeRandomizeFrame on the fully formed frame to derandomize the frame. This should only be called if the frame was randomized by the sending end.

5.3.2.4 CCSDS TM-SDLP (TMTF)

The Telemetry Space Data Link Protocol (TM-SDLP) functionality leverages code from Morpheus/LRO. Outgoing data is packaged into a TM Transfer Frame (TF). TFs are of a fixed size per virtual channel or master channel. Depending on the settings enabled for a

mission, data from multiple small packets may be blocked together in a TF, packets too large to fit in a single TF may be segmented, or each TF may be limited to a single packet of data.

The TM-SDLP is the higher sublayer protocol in the CCSDS recommendation for the Data Link Layer protocol. The lower level sublayer, the TM Synchronization and Channel Coding protocol included in the TM-SYNC library provides additional functionality when processing a TM TF. These functions are:

- Error-control coding/decoding, including frame validation
- TF delimiting/synchronization
- Pseudo-randomizing

The functionality included in the Morpheus code was limited to the TF delimiting/synchronization with the addition of the Attached Sync Marker (ASM) to the beginning of the TF. The transmitted codeblock included octets for the Reed-Solomon check symbols, but the Reed-Solomon error-control coding was not performed. For this design effort, the library includes a full set of functions for the TM-SDLP higher sublayer protocol. For the TM Synchronization and Channel Coding sublayer protocol, the ASM is added to the beginning of each TF and frame randomization services is also provided.

The TF consists of a primary header, an optional secondary header, the TF data field, an optional Operational Control Field and an optional Frame Error Control Field.

The TMTF API hides the details of the message structure. Implementations of TO should not attempt to access a TF field directly, as the structure is subject to change.

The TMTF function APIs are listed in Table 24.

Table 24: TMTF API

CSU	Description
TMTF_LibInit	Initialize the static TMTF CRC Table
TMTF_SetVersion	Set the version number for the transfer frame
TMTF_SetScId	Sets the spacecraft identifier
TMTF_SetVcId	Sets the virtual channel identifier
TMTF_SetOcfFlag	Sets the flag indicating the presence or absence of the operational control field
TMTF_GetMcId	Get the Master Channel ID
TMTF_GetGlobalVcId	Get the Global Virtual Channel ID (MCID + VCID)
TMTF_SetMcFrameCount	Set Master channel frame count in TF
TMTF_SetVcFrameCount	Set Virtual channel frame count in TF
TMTF_IncrVcFrameCount	Increment Virtual channel frame count in TF

CSU	Description
TMTF_SetSecHdrFlag	Sets the flag indicating the presence or absence of the secondary header
TMTF_SetSyncFlag	Sets the flag indicating the type of data inserted in the TF Data Field
TMTF_SetPacketOrderFlag	Sets the Packet Order Flag.
TMTF_SetSegLengthId	Sets the Segment Length Identifier
TMTF_SetFirstHdrPtr	Sets the position of the first octet of the first Packet that starts in the TF.
TMTF_SetSecHdrLength	Sets the length in octets of the secondary header, if present
TMTF_SetSecHdrData	Sets the data field of the secondary header, if present
TMTF_SetOcf	Sets the value of the Operational Control Field, if present
TMTF_UpdateErrCtrField	Calculates the value for the Frame Error Control field and sets the field, if present

5.3.2.5 CCSDS TM-SDLP (Service)

The Service library TM_SDLP includes all required end-points for the TO application to form TM Transfer frames. The service make use of the TMTF API presented above to manipulate the transfer frame headers.

The general implementation of the TM-SDLP makes use of managed channel configuration inputs to initialize each virtual channel and stores the state of the frame in a status structure available by the TO application. The TO application is responsible for providing pointers to memory buffers that it allocates in to_custom.c. A sliding window overflow buffer is implemented such that packets are stored in a separate buffer when a transfer frame is full.

The TM-SDLP function APIs are listed in *Table 25*.

Table 25: TM-SDLP API

CSU	Description
TM_SDLP_InitIdlePacket	Initializes an Idle Data Buffer with a repeating pattern sequence.
TM_SDLP_InitChannel	Populate the channel and overflow info structures based on provided configuration data and buffer pointers.
TM_SDLP_FrameHasData	Returns whether the frame has data or not

CSU	Description
TM_SDLP_AddPacket	Add a CCSDS SPP packet to a provided transfer frame buffer
TM_SDLP_AddIdlePacket	Add an Idle CCSDS SPP packet to fill the provided partial transfer frame
TM_SDLP_AddVcaData	Copies a Virtual Channel Access (VCA) PDU to the TF data field at the next free octet.
TM_SDLP_StartFrame	Start a new TF by copying any data from the overflow buffer into the empty frame buffer. This readies the frame to accept new data.
TM_SDLP_SetOidFrame	Set a Frame with Only Idle Data (OID)
TM_SDLP_CompleteFrame	Complete a Frame to ready for transmisison

5.3.2.6 CCSDS TM-SYNC

A set of function is provided through the TM-SYNC library to perform the synchronization operations as specified in CCSDS 131.0-B-2. The provided library performs the sending end services, which includes adding a fixed size header to the TM Transfer frames and performing pseudo-randomization of the transfer frame, if required.

The TM-SYNC function APIs are presented in Table 26.

Table 26: TM-SYNC API

CSU	Description
TC_SYNC_LibInit	Initializes the static pseudo-random sequence
TC_SYNC_Synchronize	Build a Channel Access Data Unit (CADU) by appending the AMS header to the TF
TC_SYNC_PseudoRandomize	Perform TM Frame pseudo randomization

5.3.3 Addition of New Libraries to IO_LIB

Additional custom libraries can easily be added to the IO_LIB framework, to be called by the appropriate application (CI or TO). Examples of such libraries may be:

- Authentication library for incoming commands
- Additional formats or services

Although every library is specific to the format or services it is implementing, the following general guideline should be used:

- Add the public header file to the io_lib/fsw/public_inc file
- Add an event ID to the IO_LIB_Events_t defined in public_inc/io_lib_envents.h

- Implement the library in the io_lib/fsw/src/formats or io_lib/fsw/src/services directory. A format deals with header/wrapper manipulations (eg: TCTF, TMTF). A service will provide an API to interact with other OSI reference model layers, such as the COP-1 and the IO_Trans libraries.
- The resulting object (eg. my_lib.o) file must be added to the OBJS environment variable in /io_lib/fsw/for_build/Makefile
- Any unit tests for the library should use the ut-assert framework under a new library specific directory within the /io_lib/fsw/unit_tests/
- The library source file should follow this general pattern:

```
#include "my_lib.h"
int32 IO_MyLibFunction(...)
{
        ...
        if (error == True)
        {
                CFE_EVS_SendEvent(IO_MY_LIB_EID, CFE_EVS_ERROR,
                                        "My Lib Error: Error message Here.");
        }
        ...
        return Size_or_Status;
}
```

- Your header file should include the io_lib.h file (#include "io_lib.h")
- The return code of API functions is generally of type int32 with negative numbers representing an error as defined in your public header file.

5.4 C3I Library (C3I_LIB)

A set of libraries has been developed to support the C3I protocol used by the MPCV Orion vehicle. As in the IO_LIB, these libraries are separated as Format and Service libraries.

- Formats:
  - DEM
  - OSMF
- Services:
  - DEM translation service (DEM_CCSDS)

The OSMF protocol is a proprietary protocol, and so is not discussed in this document.

5.4.1 DEM Format Protocol

A DEM is a message format consisting of a series of bytes representing a header, variable length data, and an optional trailer. Applications create DEM messages by allocating sufficient memory, calling the DEM API to initialize the contents of the DEM, and then storing any appropriate data into the structure.

The DEM API hides the details of the message structure, providing routines such as DEM_GetMsgTime and DEM_TimeStampMsg in order to get and set a message time. “Setter” and “getter” functions are available for all the fields in a DEM. Applications should not attempt to access or modify a DEM field directly, as the message structure is subject to change.

For each function in the API, a pointer to the DEM message is passed as an argument. All pointers are checked if NULL. For setter functions, a NULL pointer results in a return without any message processing taking place. For getter functions, a NULL pointer results in a return value of zero or a value indicative of no data.

The DEM header consists of a primary header and possibly a secondary header. The size of the primary header is determined by whether the optional Counter field is included. The existence of a secondary header is specified by the Secondary Header Flag in the primary header. The size of the secondary header is determined by the Secondary Header Size field in the secondary header. Helper functions have been implemented that are not part of the API in order to calculate the offset to fields in the DEM whose locations vary between DEMs.

In order to correctly calculate offsets in the message, the fields of both the primary header and the secondary header, if included, must be set to appropriate values. If constructing a

DEM message, the following primary header fields must be set prior to any other processing:

- Message size
- CRC flag
- Content Map ID size
- Counter Flag
- Secondary header flag

After setting the specified primary header fields, the following secondary header fields may be set, provided the secondary header flag is set:

- Secondary header type
- Secondary header size

Once those values are set, the remaining header fields and the payload data may be set as needed prior to sending the message to its intended receiver.

Internally, bit manipulations are implemented as macros for efficiency, but the macros are not part of the DEM API. For fields that do not require bit manipulations, direct assignment of values to fields, or vice-versa, is the preferred implementation. For fields which do not have an easily assignable variable, e.g. the variable sized Content Map ID, the CFE_PSP_MemCpy function is used. The CFE_TIME_GetTime function is used when setting the time stamp on a message. No other CFS functionality is used within the DEM API.

All bit and byte fields defined in a DEM use big endian ordering. When accessing the message payload, the application must handle endian issues for the payload. Header and trailer endian ordering is handled by the API.

Table 27: C3I DEM API

CSU	Description
DEM_InitMsg	Initializes a DEM with a set in input values, zeroing the message first, if desired
DEM_GetMsgSize	Returns the total size in octets of the DEM
DEM_GetVersion	Returns the protocol version of the DEM
DEM_SetCrcFlag	Sets the flag indicating whether the DEM has a CRC field
DEM_GetCrcFlag	Returns the flag indicating whether the DEM has a CRC field
DEM_SetContentMapId_Size	Sets the size of the content map ID
DEM_GetContentMapId_Size	Returns the size of the content map ID

CSU	Description
DEM_SetCounterFlag	Sets the flag indicating whether the DEM has a counter field
DEM_GetCounterFlag	Returns the flag indicating whether the DEM has a counter field
DEM_GetSecHdrFlag	Returns the flag indicating whether the DEM has a secondary header
DEM_SetOriginCode	Sets the origin code
DEM_GetOriginCode	Returns the origin code
DEM_SetOperationCode	Sets the operation code
DEM_GetOperationCode	Returns the operation code
DEM_SetStreamId	Sets the combined value for the system ID and type ID
DEM_SetSystemId	Sets the system ID
DEM_GetSystemId	Returns the system ID
DEM_SetTypeId	Sets the type ID
DEM_GetTypeId	Returns the type ID
DEM_SetContentId	Sets the content ID
DEM_GetContentId	Returns the content ID
DEM_TimeStampMsg	Sets the current time in the time fields, both seconds and fractions of a second
DEM_GetMsgTime	Returns the seconds and fractions of a second from the time fields
DEM_SetSourceSystem	Sets the source system
DEM_GetSourceSystem	Returns the source system
DEM_SetCounter	Sets the counter, if the counter flag is set
DEM_GetCounter	Returns the counter, or zero if the counter flag is not set
DEM_SetSecHdrEvtTrgId	Sets the event trigger ID in the secondary header, if the secondary header type is Event Trigger
DEM_GetSecHdrEvtTrgId	Returns the event trigger ID in the secondary header, if the secondary header type is Event Trigger
DEM_SetSecHdrTime	Sets the seconds and fractions of a second in the secondary header time fields to the values of the time argument, if the secondary header type is Time Trigger
DEM_GetSecHdrTime	Return the seconds and fractions of a second from the secondary header time fields, if the secondary header type is Time Trigger
DEM_SetContentMapId	Sets the content map ID, if the content map ID size is greater than zero
DEM_GetContentMapId	Returns the content map ID, if the content map ID size is greater than zero
DEM_GetDataLength	Returns the length in octets of the user data

CSU	Description
DEM_SetUserData	Sets the user data
DEM_GetUserData	Returns a pointer to the user data
DEM_SetCrc	Sets the CRC32 value, if the CRC flag is set
DEM_GetCrc	Returns the CRC32 value, or zero if the CRC flag is not set
DEM_GetCalculatedCrc	Returns the calculated CRC32 value of the DEM, or zero if the CRC flag is not set
DEM_UpdateCrc	Calculates the CRC32 and sets that value in the CRC field
DEM_IsValidCrc	Returns a true/false value indicating whether the message's CRC32 value matches the calculated value

5.4.2 Design Decisions on CFS Message Formats for C3I-DEM

CFS is designed to move messages between applications on the SB using the Space Packet Protocol to provide routing information. However, the interface was designed so that new protocols could be added to the existing API, allowing different protocols to be used without impacting the user applications. One design option was to add the C3I DEM message format to the existing SB code, allowing the user to choose which protocol would be used to move messages on the SB at compile time by use of compiler macros. This approach would simplify message processing on ingress and output as no additional message processing would be required to convert C3I DEM messages to SPP.

This option was not pursued for a few reasons. First, it involved updating the CFS core functionality for a protocol that was likely to be used by only one program, MPCV Orion. The effort required to push through changes to the core functionality along with the attendant recertification efforts was deemed prohibitive for the value. Second, the structure of the two protocols is different enough to cause issues. Both protocols have fields representing similar functionality, but not all the fields are represented by the same number of bits. This meant the range of values that can be represented for similar fields is different between the two protocols. In addition, the C3I DEM format contains significantly more fields. These structural differences were considered to be significant enough that existing applications would very likely be impacted by the switch to a different protocol, including the need to:

- Verify the range of values used for different fields remained valid,
- Determine the impacts to the message sizes allocated during initialization, and
- Populate the additional C3I DEM fields with usable values.

A second design option was to continue using the CFS SPP to move messages on the SB, but to translate incoming C3I DEM messages to SPP in CI, and to translate outgoing SPP

messages to C3I DEM in TO. This option has the similar issues as the above discussed option, namely:

- How to translate from one protocol to another where many fields are represented by a different number of bits
- How to populate the additional C3I DEM fields that are absent in SPP

This option would also have a CPU utilization impact due to the need to translate between protocols and copy data between the message structures.

Due to the noted issues with the first two options, a third option was developed and chosen for implementation. Messages continue to be moved on the SB using SPP. For missions that utilize C3I DEM messages, an SPP header is added to all incoming messages before being sent to the software bus. In TO, the SPP header is then stripped on outgoing messages, leaving only the DEM header. Through CI, the DEM header is stripped if the message is destined for CFS product line applications. Through TO, a DEM header is added to any messages originating from non-DEM applications and CFS services.

A set of "getter" and "setter" functions are provided for accessing and setting any of the DEM fields. As in the SPP payload, access to the DEM and its contents is via the same pointer returned by the CFS SB function, CFE_SB_GetUserData. This pointer is an argument to all the DEM processing functions, and as such, provides a simple mechanism for the user to manipulate the contents of the DEM message.

5.4.3 Translation of DEM Messages (DEM-CCSDS Service)

In order to work with both DEM and SPP headers, a translation of the header is required. This translation is performed through DEM-CCSDS translation tables as defined in and , defined in the dem_ccsds.h file in the io_lib/fsw/pubic_inc folder.

Table 28: DEM to CCSDS translation table

Type	Bits	Name
uint16	16	demSystemId
uint16	16	demSwDestId
uint16	16	demSwPortId
uint16	16	demCmdExecId
uint16	16	ccsdsStreamId
uint16	16	ccsdsCmdCode
uint8	8	keepDemHdr
uint8	8	hasDemCfsHdr

Table 29: CCSDS to DEM translation table

Type	Bits	Name
uint16	16	ccsdsStreamId
uint16	16	demSystemId
uint16	16	demSwDestId
uint16	16	demSwPortId
uint16	16	demSource
uint8	8	demCrcFlag
uint8	8	hasDemHdr

The mission specific implementation of these tables is performed in the ci_custom.c and to_custom.c. Two API methods are available to perform the translation based on the supplied message and translation table:

DEM_CCSDS_TranslateDEM(): Translate an incoming DEM message to a CCSDS message – Used by the CI application.

DEM_CCSDS_TranslateCCSDS(): Translate an incoming CCSDS message to a DEM message – Used by the TO application.

demonstrates the translation strategy implemented. Since the CFS SB is designed to use CCSDS (SPP) to move messages between applications, CI processes C3I DEM messages by prepending a CCSDS header to the messages on arrival prior to sending them to the software bus, based on the DEM message header and the translation table supplied. If the destination application expects a DEM header (specified through isDemMsg parameter in the DEM to CCSDS translation table), then the DEM header is kept as the payload header. If the destination application is non-DEM (as in all reusable CFS application and cFE services), the isDemMsg is false, the DEM header is stripped and only the message with a CCSDS header is sent over the SB.

For downlink over TO, the reverse is performed. If the message originates from a DEM application, the CCSDS header is simply stripped, leaving the downlink message of DEM + payload. If the message originates from a non-DEM application, the CCSDS header is translated to a DEM header through the DEM-CCSDS translation API. The final downlinked message is then DEM + payload only.

With this strategy, applications that require DEM information have access to the DEM header and can access specific DEM parameters through the C3I DEM library.

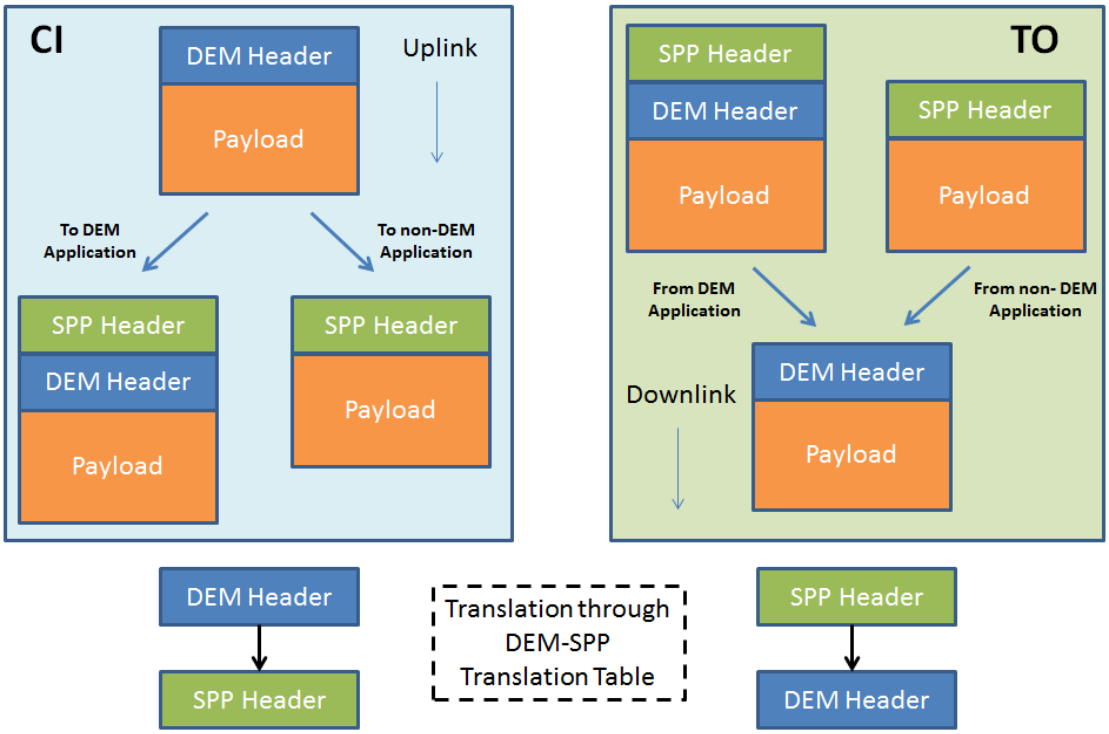

Figure 9: DEM to SPP Translations in CI and TO

6 Requirements

6.1 CI Application Requirements

Table 30 lists the CI application requirements, their rationale, on how they are satisfied in this design.

Table 30: CI Application Requirements

REQ	Description	Design Traceability
CI-01	The CFS CI application shall initialize from configurable parameters. *Rationale: Configurable parameters should include format protocol configuration, channel configuration (UDP, 422), and data link settings.*	All configurations are performed by the user in the custom layer of the CI application through local macros defined in ci_platform_cfg.h.. These macros are then used at initialization in IO_LIB function calls in the CI_CustomInit() function. Sections: **5.1.2.3**
CI-02	The CFS CI application shall receive commands from a UDP socket and publish commands to the software bus. *Rationale: Common I/O interface for ground testing.*	UDP is supported through a dedicated transport protocol library in the IO_LIB services. Sections: Error! Reference source not found.**, 5.3.1.1**
CI-03	The CFS CI application shall receive commands from an RS-422 interface and publish commands to the software bus. *Rationale: Common I/O interface for space link.*	RS-422 is supported through a dedicated transport protocol library in the IO_LIB services. Sections: Error! Reference source not found.**, 0**
CI-04	The CFS CI application shall, if configured for COP-1 protocol handling, process commands using the COP-1 protocol. *Rationale: This is the CCSDS protocol for command handling over an RF link*	COP-1 is supported through a dedicated data link protocol library in the IO_LIB services. Sections: **0, 5.3.2.2**
CI-05	The CFS CI application shall, if configured for C3I input, receive C3I DEM formatted commands and publish to the software bus. *Rationale: Protocol compatibility with Orion.*	C3I DEM is supported through a dedicated DEM format library in the IO_LIB formats. Sections: **0, 1.1.1**
CI-06	The CFS CI application shall forward commands upon receipt. *Rationale: This is a data driven app, runs only when it receives an uplink command.*	The design calls for a dedicated child task for reception of commands in the custom layer of CI. Its main function is CI_CustomMain() which loops when it receives an uplink commands.

REQ	Description	Design Traceability
		Sections: **4.2.1.3, 5.1.2.5**
CI-07	The CFS CI application shall output an HK (housekeeping) message when it receives a SEND_HK message. *Rationale: The HK message should contain any CI application status that an operator might need to know to understand if CI is executing properly.*	The main task of the CI application follows the general CFS application pattern, and sends the housekeeping message on reception of the SEND_HK message. Sections: **5.1.1.1**
CI-08	The CFS CI application shall be thread safe. *Rationale: If CI is multi-threaded, a thread safe design is required for memory protection.*	CI makes use of two tasks, and makes use of a mutex (ciMutex) to protect memory which is accessed by both tasks, such as the HK message and the outData message. Sections: **5.1.3**
CI-09	The CFS CI application shall read CI application commands from the software bus on the receipt of a WAKEUP message. Rationale: To follows the general CFS App design pattern.	The CI main thread which processes CI app commands pends on the schedule pipe, which is subscribed to the CI_WAKEUP_MID. Section: 5.1.1.1
CI-10	The CFS CI application shall allow commands to be received over multiple input channels *Rationale: In order to accept commands from different sources, the application must be able to accommodate multiple channels.*	The IO_LIB Select library allows CI to monitor multiple i/o devices over UDP and RS-422. The CI custom layer design accommodates multiple input channels. Sections: 5.1.2.6
CI-11	The CFS CI application shall provide a means to capture sensitive commands not destined for the software bus *Rationale: Certain commands should only be visible by the CI application, such as arming commands.*	The CI application includes a message called CI_GATE_CMD_MID which serves this purpose. Section: 4.2.1.1, 0
CI-12	The CFS CI application shall provide the option to keep the C3I DEM header within the CCSDS message user data sent to the software bus *Rationale: To provide flexibility on the use of DEM messages, for applications which require DEM header information*	The IO_LIB CCSDS_DEM translation library provides an option to keep the DEM header within the message user data. Section: 1.1.1
CI-13	The CFS CI application shall allow the use of custom libraries *Rationale: So that the application is expandable*	The layered design of the CI application allows for any libraries to be used, including custom designed libraries. Section: 4.3, 5.3.3

REQ	Description	Design Traceability
	with custom logic or user defined protocols	
CI-14	The CFS CI application shall, if the WAKEUP message is not received in a configurable ms, timeout, still execute its command processing loop. *Rationale: To add robustness to the CI app to ensure that it will continue to execute received commands even with malfunctioning scheduler.*	The scheduler pipe pends on WAKEUP message with a timeout, and executes its main functions on timeout. Section: 5.1.1.4
CI-15	The CFS CI application shall, if the WAKEUP message is not received in a configurable ms, timeout, still execute its main processing loop. *Rationale: To add robustness to the CI app to ensure that it will continue to process commands and provide status data from the system to the operator, even with malfunctioning scheduler.*	The scheduler pipe pends on WAKEUP message with a timeout, and executes its main functions on timeout. Sections: 5.1.1.4

6.2 TO Application Requirements

Table 31 lists the TO application requirements, their rationale, on how they are satisfied in this design.

Table 31: TO Application Requirements

REQ	Description	Design Traceability
TO-01	The CFS TO application shall initialize from configurable parameters. *Rationale: Configurable parameters should include packet or framing, CCSDS or DEM protocol, frame size, output method selection(UDP, 422, user expandable options), subscription table, event messages filters, etc.*	General configuration of I/O transport protocols and framing is done through macros defined in to_platform_cfg.h. Particular sequence of format and framing is implemented through the custom layer source file to_custom.c. Tables are included for event message filters and message subscriptions. Sections: **4.2.2.3, 5.2.1, 5.2.2.3**
TO-02	The CFS TO application shall subscribe to messages defined in the telemetry message subscription table. *Rationale: Need to use a CFS table services for the subscription table for re-configurability.*	A configuration table is included in the TO application to allow configuration of message subscriptions and route definitions for which particular IO_LIB services to perform on which message. Sections: **4.2.2.1, 5.2.1.2**
TO-03	The CFS TO application shall, if configured for simple packet only output, package each subscribed packet from the software bus into	UDP is supported through a dedicated transport protocol library in the IO_LIB services. TO can access this library through the to_custom.c implementation.

REQ	Description	Design Traceability
	a UDP packet and forward to a UDP socket. *Rationale: Common I/O interface for ground testing.*	Sections: **5.3.1.1**
TO-04	The CFS TO application shall, if configured for CCSDS framing, multiplex its input stream of packets from the software bus into CCSDS frames and forward to a UDP socket. *Rationale: Morpheus TO supports outputting CCSDS frames to UDP. Morpheus chose to a constraint of 1 packet per frame. A general CCSDS implementation multiplexes the incoming packet stream into frames (multiple packets of possibly different sizes into a single frame) to maximize bandwidth utilization.*	The CCSDS TM-SDLP framing protocol is supported through a dedicated library in the IO_LIB, accessible by the TO application in the custom implementation of to_custom.c. This library supports multiplexing a packet stream into frames for maximum bandwidth utilization. Sections: **5.3.2.3**
TO-05	The CFS TO application shall, if configured for CCSDS framing, multiplex its input stream of packets from the software bus into CCSDS frames and forward to an RS-422 interface. *Rationale: Output CCSDS frames to a serial interface that most likely would interface to an RF communication system.*	RS-422 is supported through a dedicated transport protocol library in the IO_LIB services. TO can access this library in the custom implementation of to_custom.c. Framing is performed by the TM-SDLP service in the IO_LIB. Synchronization of frames is performed by TM-SYNC. Sections: **5.3.1.2, 5.3.2.5, 5.3.2.6**
TO-06	The CFS TO application shall, if configured for C3I output, package each subscribed packet from the software bus into a C3I DEM and forward to a UDP socket. *Rationale: Protocol compatibility with Orion.*	C3I DEM is supported through a dedicated DEM format library in the IO_LIB formats, accessible through the custom implementation of to_custom.c. Sections: **5.4.3**
TO-07	The CFS TO application shall utilize a scheduler issued WAKEUP message as a trigger to perform its main loop. *Rationale: To follows the general CFS App design pattern.*	The main loop waits on the scheduler pipe, pending on the WAKEUP message. On receipt of WAKEUP, TO executes its main functions (process commands and telemetry). Sections: **5.2.1.4**
TO-08	The CFS TO application shall, if the WAKEUP message is not received in a configurable ms, timeout, still execute its main processing loop. *Rationale: To add robustness to the TO app to ensure that it will continue to output telemetry and provide status data from the system to the operator, even with malfunctioning scheduler.*	The scheduler pipe pends on WAKEUP message with a timeout, and executes its main functions on timeout. Sections: **5.2.1.4**

REQ	Description	Design Traceability
TO-09	The CFS TO application shall output an HK (housekeeping) message when it receives a SEND_HK message. *Rationale: The HK message should contain any TO application status that an operator might need to know to understand if CI is executing properly.*	The Command pipe is subscribed to the SEND_HK message. On receipt of SEND_HK, TO sends the housekeeping message to SB. The HK packet is user defined with a default implementation. Sections: **5.2.5**
TO-10	The CFS TO application shall be thread safe. *Rationale: Thread safe design is required for memory protection.*	TO is single threaded, thus inherently thread-safe.
TO-11	The CFS TO application shall allow the use of custom libraries *Rationale: So that the application is expandable with custom logic or user defined protocols*	The layered design of the TO application allows for any libraries to be used, including custom designed libraries. Sections: 4.3
TO-12	The CFS TO application shall provide the ability to send messages at a lower rate compared to the TO application or the rate at which the message is received. *Rationale: The rate of messages transmitted should be independent from the rate of message generation or the rate of the TO application itself for greater flexibility.*	Each route can set to a wake Period with respect to the TO application rate. Sections: **5.2.1.7**
TO-13	The CFS TO application shall provide a means of receiving messages over multiple pipes assigned to specific routes / virtual channels. *Rationale: So as to isolate messages on a specific route / virtual channel and prevent important messages from being dropped due to buffer overflow.*	Each route has a dedicated telemetry pipe. Sections: **5.2.1.8**
TO-14	The CFS TO application shall provide a means of prioritizing specific routes / virtual channels. *Rationale: A virtual channel may contain critical information which should take precedence over other virtual channels.*	The priority of a route is based on its order precedence as each route is processed sequentially in ascending order. Sections: **5.2.1.8**
TO-15	The CFS TO application shall allow telemetry to be sent over multiple channels *Rationale: Messages may be required over multiple output channels*	Through the use of the routing tables, a message can be routed to one or multiple routes destined for specific output channels. Sections: 5.2.1.5
TO-16	The CFS TO application shall provide	The TO routing table is reconfigurable

REQ	Description	Design Traceability
	reconfigurable routing of messages through commands and cFE table updates *Rationale: So that message routing to specific channels can be controlled and reconfigured*	through the cFE table services and through several TO reconfiguration commands. Data Groups are also provided to configure multiple messages at once. Sections: 5.2.1.5, 5.2.1.2
TO-17	The CFS TO application shall provide the option to transmit messages which hold a C3I DEM header within the CCSDS message user data. *Rationale: To provide flexibility for applications which need to define the DEM header information*	The IO_LIB CCSDS_DEM translation library provides a flag to indicate if the message holds a DEM header within the message user data. Sections: 5.4.3
TO-18	The CFS TO application shall perform throttling of message streams received from the CF application through the CF throttling semaphore *Rationale: The output files sent by CF application must be throttled as to not overflow the TO application pipe.*	The CF counting semaphore is throttled based on the available free slots of the corresponding telemetry pipe. Sections: 5.2.1.9
TO-19	The CFS TO application shall provide control to enable specific output routes from custom command parameters *Rationale: Individual routes destined for specific output channels may require specific enabling parameters and should be enabled individually.*	The TO application pattern allows the user to enable one or multiple routes based on the custom defined TO_ENABLE_OUTPUT_CC command. Sections:5.2.1.2, 5.2.2.2
TO-20	The CFS TO application shall allow custom implementation of the disabling of output routes *Rationale: It may or may not be required or desired to have the ability to disable specific output routes.*	The implementation of the disabling of routes is done at the custom layer in TO_CustomDisableOutputCmd(). Sections: 5.2.2.2
TO-21	The CFS TO application shall provide control to activate & deactivate specific enabled output routes *Rationale: The user may require control to activate/deactivate specific enabled routes.*	A TO command is provided to activate/deactivate route(s) based on a supplied route mask. Sections: 5.2.1.2
TO-22	The CFS TO application shall provide the ability to pause and resume all downlink through command *Rationale: There may be restrictions on when messages may be transmitted, such that the*	The TO application responds to two commands to pause /resume transmission of data (TO_PAUSE_OUTPUT_CC, TO_RESUME_OUTPUT_CC). Sections: 5.2.1.2

REQ	Description	Design Traceability
	downlink should be interrupted.	
TO-23	The CFS TO application shall provide the ability to ensure that specific critical messages are always included within the routing configuration table. *Rationale: To protect against unintentional reconfiguration of the table without critical messages.*	A critical message ID array is included (g_TO_AppData.criticalMid) in the TO application, which is populated in the custom layer. Sections: 5.2.3
TO-24	The CFS TO application shall provide the ability to accept other commands which have a custom implementation. *Rationale: To provide flexibility for user defined commands.*	User defined custom TO commands are handled by the TO_CustomAppCmds() function. Sections: 5.2.2.2

6.3 Bi-Directional Traceability Matrix

This section documents the mapping of the software requirements to the design elements and from the design elements to the software requirements.

6.3.1 From Requirement to Design Element

	Design Traceability CSC	Design Traceability CSU
CI-1	CI	
CI-2	CI cFE SB	UDP Transport Protocol
CI-3	CI cFE SB	RS-422 Serial Transport Protocol
CI-4	CI	CCSDS - TC-SDLP COP-1 CLCW
CI-5	CI cFE SB	C3I-DEM
CI-6	CI	
CI-7	CI	
CI-8	CI	
CI-9	CI	
CI-10	CI	Select Transport Library
CI-11	CI	
CI-12	CI	DEM-CCSDS Library
CI-13	CI, IO_LIB	
CI-14	CI	
TO-1	TO	
TO-2	TO cFE TABLE	

	Design Traceability CSC	Design Traceability CSU
TO-3	TO	UDP Transport Protocol
TO-4	TO	UDP Transport Protocol CCSDS-TM-SDLP
TO-5	TO	RS-422 Serial Transport Protocol CCSDS-TM-SDLP
TO-6	TO cFE SB	C3I-DEM UDP Transport Protocol
TO-7	TO SCH	
TO-8	TO cFE SB	
TO-9	TO	
TO-10	TO	
TO-11	TO, IO_LIB	
TO-12	TO	
TO-13	TO	
TO-14	TO	
TO-15	TO	
TO-16	TO, cFE TABLE	
TO-17	TO	DEM-CCSDS Library
TO-18	TO, CF	
TO-19	TO	
TO-20	TO	
TO-21	TO	
TO-22	TO	
TO-23	TO, cFE TABLE	
TO-24	TO	

6.3.2 From Design Element to Requirement

Design Traceability CSU	Design Traceability CSC	SRS Requirement
UDP Transport protocol	CI, TO	CI-2, TO-3, TO-4, TO-6
RS-422 Serial Transport Protocol	CI, TO	CI-3, TO-5
Select Transport Library	CI	CI-10
CCSDS-TC-SDLP	CI	CI-4
CCSDS-TM-SDLP	TO	TO-4, TO-5
C3I-DEM	CI, TO	CI-5, TO-6
DEM-CCSDS Translation	CI, TO	CI-12, TO-17

Appendix A – Abbreviations and Acronyms

AD	Acceptance check, Data (for Type-AD transfer frames)
API	Application Programming Interface
ASM	Attached Sync Marker
BC	Bypass acceptance check, Control (for Type-BC transfer frames)
BCH	Bose-Chaudhuri-Hocquenghem
CADU	Communication Access Data Unit
C3I	Command, Control, Communication, and Information
CCSDS	The Consultative Committee for Space Data Systems
cFE	Core Flight Executive
CFDP	CCSDS File Delivery Protocol
CFS	Core Flight Software
CI	Command Ingest
CLCW	Communications Link Control Word
CLTU	Communications Link Transmit Unit
COP-1	Communications Operation Procedure-1
CRC	Cyclic Redundancy Check
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
DEM	Data Exchange Message
EA	Engineering Directorate Mail Code
EVS	Event Services
FARM-1	Frame Acceptance and Reporting Mechanism-1
FOP-1	Frame Operation Procedure-1
GFE	Government Furnished Equipment
HK	Housekeeping
IO_LIB	Input / Output Library
ITOS	Integrated Test and Operating System
JSC	Johnson Space Center
LRO	Lunar Reconnaissance Orbiter
MID	Message ID
MPCV	Multi-Purpose Crew Vehicle
NASA	National Aeronautical and Space Administration
NPR	NASA Procedural Requirements
OS	Operating System
OSI	Open System Interconnection model
PDU	Protocol Data Unit
PMP	Project Management Plan
QOS	Quality Of Service
SB	Software Bus
SCH	Scheduler (application)
SCID	Spacecraft Identifier
SDD	Software Design Document
SDLP	Space Data Link Protocol
SDP	Software Development Plan
SPP	Space Packet Protocol

SRS	Software Requirements Specification
TBD	To Be Determined
TC-SDLP	Telecommand Space Data Link Protocol
TFVN	Transfer Frame Version Number
TM-SDLP	Telemetry Space Data Link Protocol
TF	Transfer Frame
TO	Telemetry Output
UDP	User Datagram Protocol
VC	Virtual Channel
VCID	Virtual Channel Identifier
WI	Work Instruction

Appendix B – Glossary of Terms

Channel	A channel refers to a virtual channel as defined by the CCSDS standard. More generally, it refers to data over a specific I/O device, such as a UDP socket or RS422 Serial Port for which data is received or transmitted.
Device	A device is the end point of a transport protocol such as a UDP socket or RS422 Serial port. It may also refer to the software bus or a custom I/O device.
Pipe	A pipe is a cFE specific term as part of the cFE software bus services. It represents a first-in/first-out message queue which can subscribe to multiple messages.
Route	A route is an abstract concept used in the Telemetry Output application for which a message passing through a specific route is subject to a sequence of operations ending with the transmission of the manipulated message over a specific output device. A route is user defined and may represent a CCSDS virtual channel or a set of operations to be performed on a specific set of messages which require special manipulation. Each route has a cFE Pipe such that messages over each route is isolated.
Stream	A stream represents a sequence of packets related to multiplexing. Generally a stream of packets will form a CCSDS transfer frame for transmission. A “Stream ID” is a CCSDS packet identifier.
Service Protocol	A service corresponds to protocol library performing services for data manipulation, exchange or packaging between OSI layers. Examples are the CCSDS COP-1 services, transport protocols (UDP, RS422), or DEM to CCSDS translation services.
Format Protocol	A format protocol library performs data manipulation of a specific format, such as header manipulations. The API end-points of a format protocol are accessed mainly by service protocols. Examples are the C3I DEM protocol, or the SDLP transfer frame protocols.

Appendix C – Example Implementations

The following examples show pseudo-code implementation of primary custom functions for CI & TO, for different transport protocols.

Example #1: CCSDS Packets over UDP

CI_CustomInit()

1. Set IO_TransUDPConfig_t config
    a. strncpy(config.cAddr, IO_TRANS_UDP_INADDR_ANY, 16)
    b. config.usPort = 5010
    c. config.timeoutRcv = 500
2. Create and bind the uplink socket
    a. IO_TransUdp_t udp
    b. fd=IO_TransUdpInit(&config, &udp)
3. Create Child Task
    a. CFE_ES_CreateChildTask(&taskId, “CI Custom Child Task”, CI_CustomMain, NULL, 0x4000, 118, 0)

CI_CustomEnableTO(CFE_SB_MsgPtr_t pCmdMsg)

1. Replace message header for TO
    a. CFE_PSP_MemCpy(pOutCmdMSg, pCmdMsg, sizeof(CI_EnableTOCmd_t))
    b. CFE_SB_InitMsg(pOutCmdMsg, TO_APP_CMD_MID, sizeof(TO_EnableOutputCmd_t), FALSE)
    c. CFE_SB_SetmCmdCode(pOutCmdMsg, TO_APP_CMD_MID)
    d. CFE_SB_GenerateChecksum(pOutCmdMsg)
2. Send message over SB
    a. CFE_SB_SendMsg(pOutCmdMsg)

* Message contains field:
- char cDestIp[16] (eg: “127.0.0.1”)
- uint16 usDestPort (eg: “5011”)

CI_CustomMain()

1. Check if fd != 1

while(size >= 0) {

2. Pend forever on Socket:
    a. size = IO_TransUdpRcvTimeout(fd, &destAddr, &buffer[0], 1000, IO_TRANS_PEND_FOREVER)
3. On receipt of message:
    a. msgId = CFE_SB_GetMsgId((CFE_SB_MsgPtr_t) &buffer[0])
4. Validate Checksum
    a. CFE_SB_ValidateChecksum((CFE_SB_MsgPtr_t) &buffer[0]) == TRUE
5. Deal with GATE_CMD
    a. IF msgID == CI_GATE_CMD_MID, THEN CI_CustomGateCmds(CFE_SB_MsgPtr_t) &buffer[0])
6. Else send to Software BUS
    a. CFE_SB_SendMsg(sbMsg)

}

TO_CustomInit()

1. Create destination socket
    a. destSocketId = IO_TransUdpCreateSocket()
2. Set Critical Message IDs
    a. g_TO_AppData.criticalMid[0] = CFE_ES_SHELL_TLM_MID
    b. g_TO_AppData.criticalMid[1] = CFE_EVS_EVENT_MSG_MID
    c. g_TO_AppData.criticalMid[2] = CFE_SB_ALLSUBS_TLM_MID
3. Set Routes
    a. g_TO_AppData.routes[0].usExists = 1
    b. g_TO_AppData.routes[0].sCfCntSemIdx = 0

TO_CustomEnableOutputCmd(CFE_SB_MsgPtr_t pCmdMsg)

1. Set Destination Address
    a. iStatus = IO_TransUdpSetDestAddr(udp_fd, pCmdMsg->cDestIp, pCmdMsg->usDestPort)
2. Set route as configured
    a. TO_SetRouteAsConfigured(0)
3. Return route Mask
    a. Return 0x0001

TO_CustomProcessData(CFE_SB_MsgPtr_t msgPtr, int32 size, int32 tblIdx, uint16 usRouteId)

1. IO_TransUdpSnd(destSocketId, &destSockAddr, msgPtr, size)

Example #2: CCSDS Packets over RS422

CI_CustomInit()

1. Set IO_TransRS422Config_t config
    a. strcpy(&config.device, "/dev/ttyS0")
    b. config.baudRate = 19200
    c. config.timeout = 100
    d. config.minBytes = 6
2. Open the RS422 serial port
    a. fd=IO_TransRS422Init(&config)
3. Create Child Task
    a. CFE_ES_CreateChildTask(&taskId, "CI Custom Child Task", CI_CustomMain, NULL, 0x4000, 118, 0)

CI_CustomEnableTO(CFE_SB_MsgPtr_t pCmdMsg)

1. Create a message header for TO
    a. CFE_SB_SetMsgId(pCmdMsg, TO_APP_CMD_MID)
    b. CFE_SB_SetmCmdCode(pCmdMsg, TO_APP_CMD_MID)
    c. CFE_SB_GenerateChecksum(pCmdMsg)
2. Set file descriptor argument from local fd.
    a. cmdMsgPtr->fileDesc = fd
3. Send message over SB
    a. CFE_SB_SendMsg(cmdMsgPtr)

CI_CustomMain()

1. Check if fd != 1

while(size >= 0) {

2. Pend forever on Serial Port for header size:
    a. size = IO_TransRS422ReadTimeout(fd, &buffer[0], 6, -1)
3. On receipt of header (size = 6):
    a. msgId = CFE_SB_GetMsgId((CFE_SB_MsgPtr_t) &buffer[0])
    b. msgSize = CFE_SB_GetTotalMsgLength(CFE_SB_MsgPtr_t) &buffer[0])
    c. dataSize = msgSize -6
4. Read remainder of message:
    a. size = IO_TransRS422Read(fd, &buffer[6], dataSize)
5. Validate Checksum
    a. CFE_SB_ValidateChecksum((CFE_SB_MsgPtr_t) &buffer[0]) == TRUE
6. Deal with GATE_CMD
    a. IF msgID == CI_GATE_CMD_MID, THEN CI_CustomGateCmds(CFE_SB_MsgPtr_t) &buffer[0])
7. Else send to Software BUS
    a. CFE_SB_SendMsg(sbMsg)

}

TO_CustomInit()

1. Set Critical Message IDs
    a. g_TO_AppData.criticalMid[0] = CFE_ES_SHELL_TLM_MID
    b. g_TO_AppData.criticalMid[1] = CFE_EVS_EVENT_MSG_MID
    c. g_TO_AppData.criticalMid[2] = CFE_SB_ALLSUBS_TLM_MID
2. Set Routes

a. g_TO_AppData.routes[0].usExists = 1
b. g_TO_AppData.routes[0].sCfCntSemIdx = 0

TO_CustomEnableOutputCmd(CFE_SB_MsgPtr_t pCmdMsg)

1. Set local fd to input argument
    a. fd = pCmdMsg->fileDesc
2. Set route as configured
    a. TO_SetRouteAsConfigured(0)
3. Return route Mask
    a. Return 0x0001

TO_CustomProcessData(CFE_SB_MsgPtr_t msgPtr, int32 size, int32 tblIdx, uint16 usRouteId)
    IO_TransRS422Write(fd, msgPtr, size)