Downlink Module Specification

James E. Francis

MIT 64-30110.0401

Rev. D

October 25, 1993

Contents

Figures

1 Introduction

1.1 Scope

This document describes the components that make up the Downlink Software Module (DOWNLINK). This module consists of a set of functions that the Flight Software uses to form various types of downlink packets and send them to the Flight Data System (FDS) via an RS422 interface.

This document does not describe the overall design of the EDS, nor does it describe the operating details of each component in this module. It assumes that the reader is extremely familiar with the Flight Software Global Design, with the "C" programming language and with the architecture of the EDS.

This document should be of primary interest to Flight Software Engineering, Flight Hardware Engineering, Ground Software Engineering and Software Quality Assurance.

1.2 References

MIT 64-03007 Rev. D .... "FDS to EDS Interface Control Document"
MIT 64-30102 Rev. F .... "Flight Software Global Design"
MIT 64-30110.2301 Rev. D .... "Startup Module Specificatin" (STARTUP)
MIT 64-30110.2401 Rev. C .... "DMA Module Specification" (DMA)
MIT 64-30111.0201 Rev. C .... "Buffer Pool Manager Module Specification" (BUFP)
MIT 64-30110.0701 Rev. C .... "Timing Module Specification" (TIMING)
MIT 64-30110.2001 Rev. C .... "Utility Module Specification" (UTIL)
MIT 64-30110.1301 Rev. C .... "Queue Manager Module Specification" (QUEUE)
MIT 64-30110.1201 Rev. E.............. "Interrupt Manager Module Specification" (INTR)
MIT 64-30110.0101 Rev. C .... "Back Registers Module Specification" (BREG)

1.3 Notations

This document assumes the following notations:

UINT8 .... Refers to an 8-bit unsigned value
UINT16 .... Refers to a 16-bit unsigned value.
BOOL .... Refers to a 16-bit value which represents a TRUE or FALSE condition.

QUEUE .... Refers to a queue control structure, which is used to maintain the state of a queue.

1.4 Module Perspective

The following diagram is a copy of the Level 1 data flow diagram from the **Flight Software Global Design**, Section 4.1.2. The scope of the Downlink Module is indicated by the shaded area:

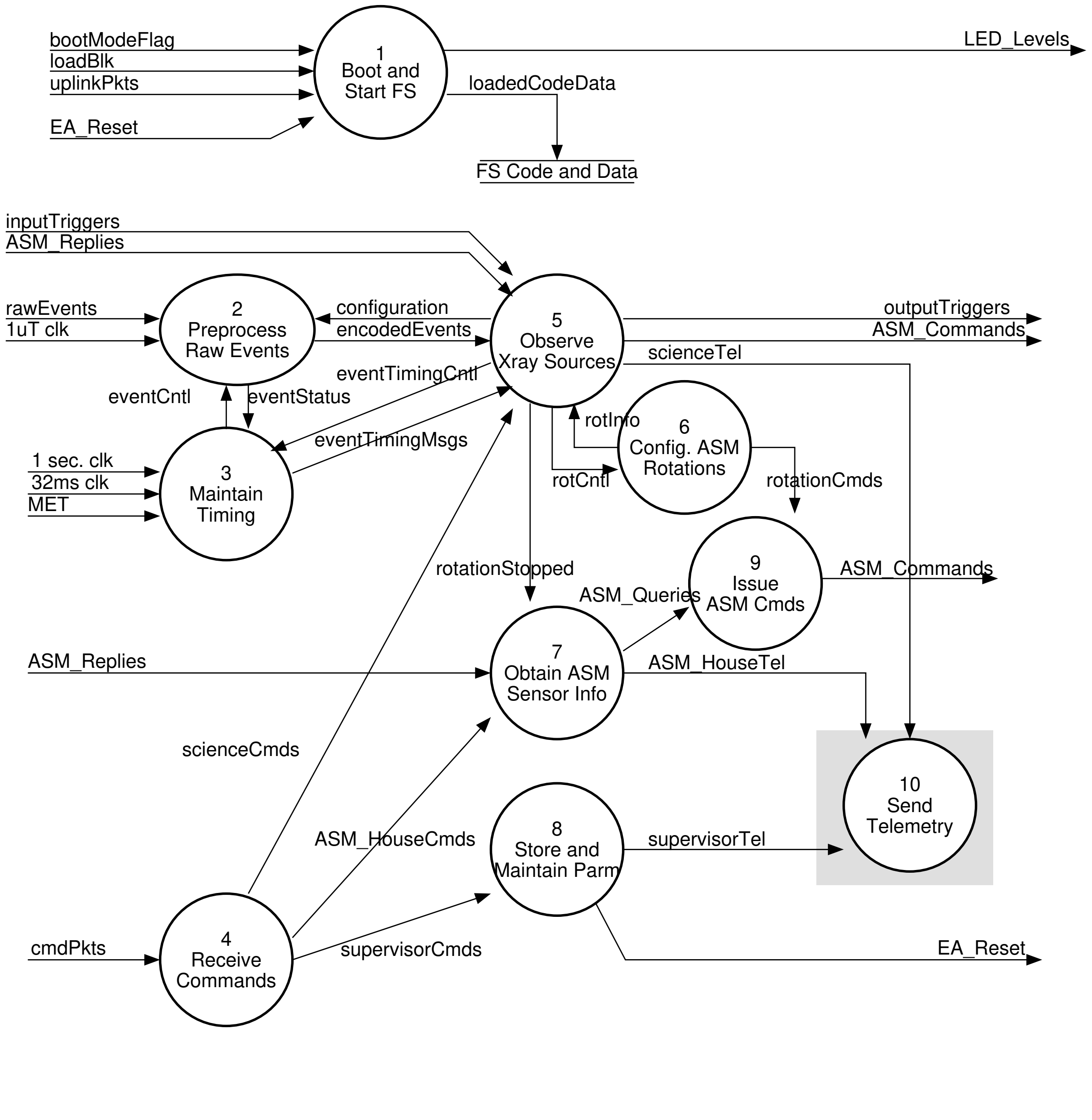

Figure 1 Module Perspective

1.5 Document Overview

Sections 1 and 2 provide an overview of the Downlink Module, summarizing the purpose, services, context, and interfaces of the module. Section 3 describes all software structures and constants visible to modules outside of the Downlink Module. Section 4 describes the interrupts on which the Downlink Module relies. Section 5 describes the functions provided to other modules by the Downlink Module. Section 6 describes some of the implementation details of the module. Section 7 describes which requirements are met by the module. Section 8 describes what types of tests need to be performed to ensure that the implementation of the module is operating properly. Appendix A lists items which are currently undefined, or that need to be reviewed.

1.6 Acronyms

Please refer to the **Flight Software Global Design Specification**, Section 1.5, for a list of acronyms and abbreviations.

2 Overview

2.1 Purpose

The purpose of the Downlink Module is send data to the Flight Data System (FDS) via an RS422 interface. This module builds downlink packets from the data supplied by its callers. The packets built by this module conform to the CCSDS standard for telemetry packets. This module can also issue panic messages. This module uses the facilities provided by the DMA Module to perform downlink packet transfer.

2.2 Services

The Downlink Module provides the following services:

1) Send telemetry packets .... Fills in the CCSDS Primary Header and Secondary Headers, and asynchronously sends the packet to the FDS via the RS422 Interface, using DMA.

2) Send panic message packets......... Fills in the CCSDS Primary and Secondary Headers and the panic message code and argument, and synchronously sends the packet to the FDS via the RS422 Interface, using DMA.

3) Access packet data .... Provides the caller with a pointer to the start of the application data within a telemetry packet buffer.

4) Idle Indication .... Indicates when the telemetry system is not transferring any telemetry packets and has none waiting to be sent.

2.3 Module Context

The following illustrates the Downlink Module relationships with the rest of the Flight Software residing in an EA:

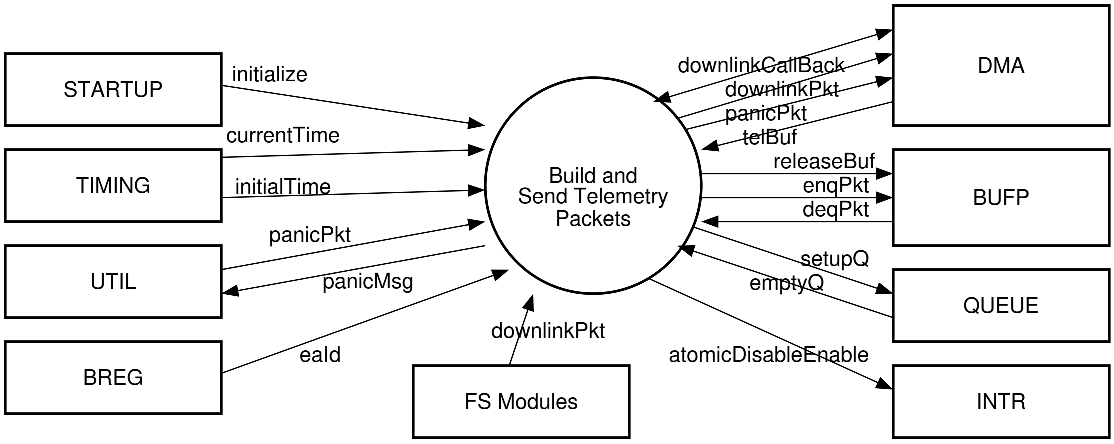

Figure 2 Downlink Module Context Diagram

Upon startup, the **Startup Module** calls Downlink's initialization routine (initialize). This routine uses the **Back End Register Module** to obtain the event analyzer's id (eaId), initializes all of Downlink's internal tables and state variables, uses the **TIMING Module** to obtain the reset time (initialTime), which Downlink later uses to tag all telemetry packets, uses the **DMA Module** to install its downlink callback function (downlinkCallBack), and uses the **Queue Utility Module** to setup the downlink queue control structure.

Once the Flight Software is running, other Flight Software Modules post telemetry packets to be transferred (downlinkPkt). The Downlink Module fills in the packets' CCSDS Primary Header, using the obtained event analyzer id to select the appropriate "Application Id", and fills part of the secondary header with an Event Analyzer Run Identifier, which is the initial reset time read during startup. If transfers are already in progress, the Downlink Module enqueues the packets onto its downlink queue, using the **Buffer Pool Module** (enqPkt). If there are no transfers under way, it obtains the current time (currentTime), using the **Timing Module**, fills in the time in the secondary header and then uses the **DMA Module** to start transferring the packet (downlinkPkt).

Once a packet has been transferred, the **DMA Module** invokes Downlink's callback function (downlinkCallBack). The callback function is passed a pointer to the start of the data buffer that was transferred (telBuf), and the number of bytes transferred (not shown). If the passed transfer count doesn't match the expected transfer length, the callback function issues a panic message and resets the EA, using the **Utility Module** (panicMsg). If the transfer count is valid, the callback function then returns the buffer (releaseBuf) to its buffer pool, using the **Buffer**

Pool Module. The callback function then examines the downlink queue, using the **Queue Utility Module**. If there are more packets to transfer, the callback function dequeues the next packet to send (deqPkt), using the **Buffer Pool Module**. It then obtains the current time (currentTime) from the **Timing Module**, fills in the time field in the secondary header, and starts another DMA transfer (downlinkPkt) using the **DMA Module**.

The **Utility Module** uses the Downlink Module to send panic message packets (panicPkt) to the FDS. The Downlink Module uses the event analyzer id, and the current time (currentTime), obtained from the **Timing Module**, to fill in the CCSDS primary and secondary headers, and stores the panic message code and argument into the body of the packet. The Downlink Module then passes the panic message packet onto the **DMA Module**, for transfer to the FDS.

At various stages of this process, the Downlink Module temporarily disables interrupts (atomicDisableEnable) using the **Interrupt Manager Module**.

If the Downlink Module suspects that the EA has been corrupted, it issues panic messages (panicMsg) and resets the EA using the **Utility Module**. The **Utility Module** ensures that recursive panics don't occur.

2.4 Theory of Operation

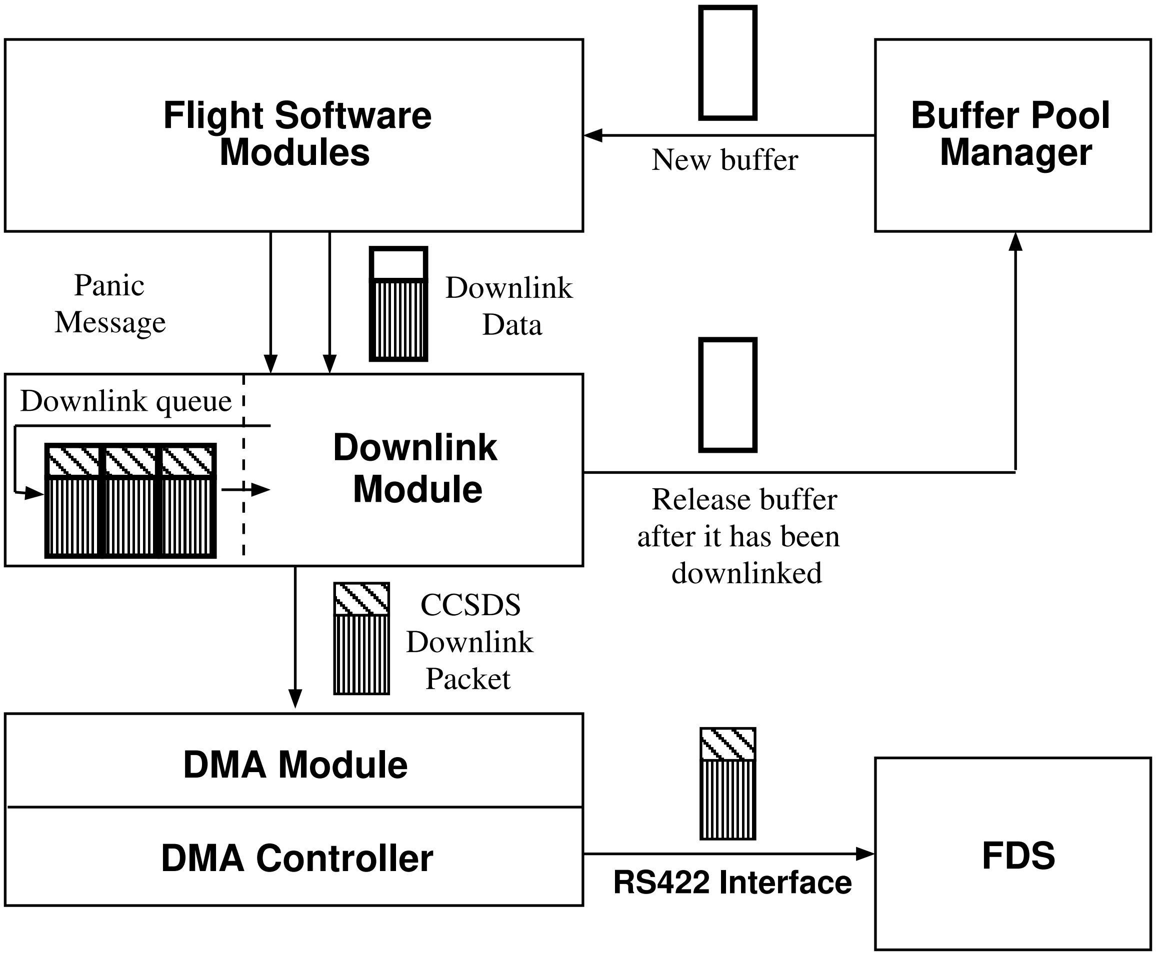

Figure 3 Downlink subsystem data flow diagram

The Downlink Module can transfer data packets asynchronously and a panic message synchronously to the Flight Data System using facilities provided in the DMA Module.

This Module fills the header portion of the data packet supplied by the client. This header consists of the CCSDS primary header, the CCSDS secondary header and the packet stream Id (explained below).

The Downlink Module is used by different Flight Software subsystems to downlink data packets of different type. The packet type is referred to as a "packet stream", meaning a sequence of packets of a given type. The packet stream Id is supplied by the client code and placed into the data packet header by the Downlink Module.

The CCSDS secondary header contains the time (MET value) at which the packet transfer is initiated, and an Event Analyzer Run Identifier (EA Run Id). The EA Run Id is identical to the time (MET value) at which the EA was initialized after a reset. The transfer time is obtained using the Timing Module's services, and the reset time is obtained during Downlink's initialization, again using the Timing Module's services.

The FDS stores and transmits packets using different priority levels. The Flight Software running on a given EA may post downlink packets using four priority levels. From a given EA's perspective, these levels are numbered 0 through 3, 0 being intended for the highest priority and 3 for the lowest. Each priority level assigned to an EA has an application Id number associated with it, which is placed into the CCSDS primary header. The mapping of the priority levels to application Id's takes place during startup procedure. The FDS uses this application id to determine at what priority to telemeter the packet. The **FDS EDS Interface Control Document**, Section 2.2.1, defines priority levels 0, 1, 2 and 3 as "Real-time", "Priority 1", "Priority 2" and "Priority 3." Currently, the Flight Software does not explicity use the "real-time" level. It uses "Priority 1" for all software housekeeping packets. The Flight Software, however, can be configured to use any priority level, including the "Real-Time" level, to send science data (see **Science Thread Module Specification**, Section 3.2.2.1).

Packets with the same application Id belong to one "packet sequence". An ascending sequence number is maintained for each packet sequence and is attributed to each packet in the sequence. This number is reset to zero on EA restart and is incremented for every new packet belonging to the packet sequence. A packet's sequence number is placed into the packet's CCSDS primary header.

Asynchronous downlink data tranfer works as follows. The client obtains a buffer using the Buffer Pool Manager Module. It then fills the data portion of the buffer, leaving the header space blank. This buffer is then passed to the Downlink Module. First, this module builds the downlink packet filling in all the header information. Then either the packet is time-stamped and the DMA controller is programmed immediately to transfer the packet to the FDS or the packet is placed into the downlink FIFO queue. The DMA controller is programmed immediately if there are no packets waiting in the downlink queue and if the DMA controller is not currently busy with another downlink packet's transfer. Then, program control is returned to the caller.

Upon completing a downlink packet transfer, the buffer which contained that packet is released back to the buffer pool. The Downlink Module then obtains the next packet from the downlink queue, time-stamps the packet and transfers the packet using the DMA Downlink channel.

Synchronous operation works by not returning to the caller until the transfer is complete. This is very inefficient for most types of data transfer, and is used only for panic situation, where the state of the system is suspect, and the caller is about to reboot the EA.

To post a packet for downlink, the client code must use the following procedure:

1) Obtain a buffer for the downlink packet using the Buffer Pool Manager services.

2) Ask this module for the pointer to the data portion of the downlink buffer.

3) Fill the data portion of the downlink packet.

4) Post the downlink data for transmission using this module.

To post a panic message, the client code must use the following procedure:

1) Construct the panic message (see the Utility Module Specification).

2) Send the panic message using this module.

2.5 Interface Summary

The Downlink Module uses the services provided by the following modules:

TIMING Module .... Provides the current MET value for the time-stamp used in the telemetry packets secondary header.
Provides the first MET value read during EA initialization, which is used to identify each telemetry packet with a given EA boot (EA Run Id).

UTIL Module .... Issues panic messages to the Downlink Module.
Forms panic messages, and resets the EA.

BREG Module .... Provides the event analyzer id.

DMA Module .... Transfers telemetry packets to the FDS.
Calls the downlink callback function when a transfer completes.
Synchronously transfers a panic message packet.

BUFP Module .... Returns used buffers to the appropriate pool.
Enqueues and dequeues buffers to and from the downlink queue.

QUEUE Module .... Sets up the downlink queue control structure.
Indicates when the downlink queue is empty.

INTR Module .... Peform nested interrupt enables and disables, to allow operations to be performed without interruptions.

2.6 Data Structures and Constants Summary

The Downlink Module defines the format of the telemetry packet headers and packet stream id values. It also defines identifiers for the FDS priority levels. The Downlink Module adheres to the interface definitions defined by the **FDS to EDS Interface Control Document**.

2.7 Interrupt Summary

Downlink Interrupt	Must call downlink_complete_callback()

2.8 Function List Summary

This module defines the following functions:

DOWNLINKinit()	Initialize the DOWNLINK module.
DOWNLINKpost()	Post data to send to the ground.
DOWNLINKpanic_send()	Send a panic message to the ground.
DOWNLINKdata()	Get a pointer to the data portion of a downlink packet.
DOWNLINKidle()	Indicate when no telemetry packets are being sent or are waiting to be sent
downlink_complete_callback()	Must be called once every time the DMA data transfer on the downlink channel is completed.

2.9 System Resource Utilization Summary

This module uses the following system resources:

Telemetry .... 16 bytes/packet of header information. Each packet is 1024 bytes (may be truncated by FDS).
DMA .... One of four available DMA channels.

3 Data Structures and Constants

3.1 Data Structures

3.1.1 Downlink Packet Header

This section describes the format of the Downlink Packet Header. This description reiterates the description provided by the **Flight Software Global Design**, Section 3.2.2. All Downlink Packets start with a packet header which contains the following:

CCSDS Primary Header [6 bytes]	This header is attached to the EDS downlink data to conform to CCSDS packet telemetry recommendations.
CCSDS Secondary Header [8 bytes]	This header contains the time at which the packet was sent to the FDS, and an Event Analyzer Run Identifier (the initial startup time).
EDS Header [2 bytes]	This header is filled with the EDS-specific information, which is interpreted by the Instrument Support Software.

The downlink packet header is then followed by 0 to 1008 bytes of downlink data. Since the Primary and Secondary Headers may be interpreted by FDS, the byte ordering of the headers within the EA must be chosen to ensure that the FDS interprets the fields correctly. The Downlink hardware fetches each 16-bit word in the order that they appear in memory, and transmits the word to the FDS most-significant bit first. The Downlink hardware accomodates the fact that the 80286 is a "little-endian" machine (i.e. stores words least-significant byte first), and therefore, the Downlink Module must form these headers by writing each 16-bit word least-significant byte first. The Downlink Module must, however, write 32-bit words most-significant word first, where each byte in the word is written least-significant byte first. For example, to send a 32-bit value such as 0x012345678 as one of the header fields, the EA's memory, in words, is: 0x01234, 0x05678 and in bytes, is: 0x034, 0x012, 0x078 0x056 (NOTE: the natural ordering on the 80286 would be 0x078, 0x056, 0x034, 0x012).

3.1.1.1 CCSDS Primary Header

[application Id, lsb] .... Packet Id (lower 8 bits)
[v, v, v, t, s, a, a, a] .... Packet Id (upper 8 bits)
(v) .... Version number (3 bits)
(t) .... Type (1 bit)

(s) .... Secondary header flag (1 bit)
(a) .... Application Id (upper 3 bits)
[source sequence count, lsb] .... Packet sequence control (lower 8 bits)
[f, f, s, s, s, s, s, s] .... Packet sequence control (upper 8 bits)
(f) .... Segment flags (2 bits)
(s) .... Source sequence count (upper 6 bits)
[length, lsb] .... Packet length (lower 8 bits)
[length, msb] .... Packet length (upper 8 bits)

Description

The fields of the CCSDS primary header have the following meanings:

Version number .... It will have a value of all zeros.
Type .... This bit is used to distinguish uplink packets from downlink packets. For downlink packets, this bit is set to zero.
Secondary header flag .... This bit is set to 1 to inidicate the presence of the CCSDS secondary header.
Application process Id .... This is the primary method for identifying packets (see the "Theory of Operation" section).
Segment flags .... No Segmentation of source packets will be supplied by the FDS. The EDS sets these bits to 11 (base 2).
Source sequence count...................... This is a 14 bit counter maintained for each application Id, which increments for each packet sent with the specified application Id (see the "Theory of operation section).
Packet Length .... This is the length of the entire packet less the length of the primary header (6 bytes) less one.

3.1.1.2 CCSDS Secondary Header

[Send Time, seconds, msw, lsb] .... MET (S/C Clock, in seconds) just before transmission (lower byte of the upper word).
[Send Time, seconds, msw, msb] ..... MET just before transmission (upper byte of the upper word).
[Send Time, seconds, lsw, lsb] .... MET just before transmission (lower byte of the lower word).
[Send Time, seconds, lsw, msb] .... MET just before transmission (upper byte of the lower word).

[EA Run Id, msw, lsb] .... Event Analyzer Run Identifier (lower byte of the upper word).
[EA Run Id, msw, msb] .... Event Analyzer Run Identifier (upper byte of the upper word).
[EA Run Id, lsw, lsb] .... Event Analyzer Run Identifier

(lower byte of the lower word).
[EA Run Id, seconds, lsw, lsb] .... Event Analyzer Run Identifier (upper byte of the lower word).

Description

Secondary header of the downlink packet contains two fields consisting of the MET (seconds) at which the packet was transferred to the FDS by the downlink subsystem (Send Time), and an Event Analyzer Run Identifier (EA Run Id), which is unique for each EA reset, and which never decreases in value from one reset to the next (it is actually the time at which the EA ran its initialization procedure).

NOTE: The bytes used for the EA Run Id used to be defined as a Sub-second field of the current time (as of Rev. B of the **FDS to EDS Interface Control Document**). The Flight Software had always set these bytes to zero. Due to an enhancement request from NASA's Science Operation Center (SOC), these bytes now contain an EA Run Id, which is the MET value in effect when the EA runs its initialization procedure. This allows the SOC to group EDS Flight Software telemetry packets, and associate the group with a particular EA reset (regardless of whether the reset was intentional or not).

3.1.1.3 EDS Header

[stream Id] .... Packet Stream Id (1 byte)
[reserved] .... Reserved field (1 byte)

Description

Packet Stream Id .... Indicates the type of the packet being sent (see Section 3.2.29).

3.1.2 Panic Message Packet Format

The following describes the format of the data portion of a Panic Message, whose stream id is STREAM_ID_PANIC_MSG (see Section 3.2.29.2):

Message Code (2 bytes) .... A 16-bit value which uniquely identifies a given message. Some messages require an additional 32-bit integer value, which is sent in the next argument.
Message Argument (4 bytes) .... This is a 32-bit integer argument, which may or may not be needed for the message specified in the previous argument. The bytes are always sent. If the message doesn't use this field, the contents are ignored.

3.2 Constants

3.2.1 DOWNLINK_NUM_PRIORITIES (constant value 4)

Each EA may write its downlink packets using one of four FDS priority levels. Higher priority means the data should be actually sent to the ground sooner. From the EA perspective these levels are numbered 0 through 3.

3.2.2 DOWNLINK_PKT_PRIORITY_REALTIME (constant value 0)

This is the value that a Flight Software Module uses to tell the Downlink Module to send a telemetry packet using the "real-time" priority level. This level is mapped, by the Downlink Module, to a unique application id for the EA.

3.2.3 DOWNLINK_PKT_PRIORITY_HIGH (constant value 1)

This is the value that a Flight Software Module uses to tell the Downlink Module to send a telemetry packet using the high priority level. This level is mapped, by the Downlink Module, to a unique application id for the EA.

3.2.4 DOWNLINK_PKT_PRIORITY_MEDIUM (constant value 2)

This is the value that a Flight Software Module uses to tell the Downlink Module to send a telemetry packet using the middle priority level. This level is mapped, by the Downlink Module, to a unique application id for the EA.

3.2.5 DOWNLINK_PKT_PRIORITY_LOW (constant value 3)

This is the value that a Flight Software Module uses to tell the Downlink Module to send a telemetry packet using the low priority level. This level is mapped, by the Downlink Module, to a unique application id for the EA.

3.2.6 DOWNLINK_PANIC_LEN (constant value 6)

This is the length of a panic message (see Section 3.1.2).

3.2.7 DOWNLINK_VERSION (constant value 0)

This value is written into the Version bits within the CCSDS Primary Header's Packet Identification word (see Section 3.1.1.1).

3.2.8 DOWNLINK_TELEMETRY_TYPE (constant value 0)

This value is written into the Packet Type bit within the CCSDS Primary Header's Packet Identification word (see Section 3.1.1.1).

3.2.9 DOWNLINK_SECONDARY_HEADER (constant value 1)

This value is written into the Secondary Header Flag bit within the CCSDS Primary Header's Packet Identification word (see Section 3.1.1.1).

3.2.10 DOWNLINK_SEGMENTATION_FLAGS (constant value 3)

This value is written into the Segmentation Flags bits within the CCSDS Primary Header's Packet Sequence Control word (see Section 3.1.1.1).

3.2.11 DOWNLINK_VERSION_MASK (constant value 0x007)

This mask, when "AND"ed with a Version code, zeros the upper unused bits (see Section 3.1.1.1).

3.2.12 DOWNLINK_VERSION_SHIFT (constant value 5)

A Version code must be shifted left by this number of bits to properly place it within the most-significant byte of the CCSDS Primary Header's Packet Identification word (see Section 3.1.1.1).

3.2.13 DOWNLINK_TYPE_MASK (constant value 0x001)

This mask, when "AND"ed with a Packet Type code, zeros the upper unused bits (see Section 3.1.1.1).

3.2.14 DOWNLINK_TYPE_SHIFT (constant value 4)

A Packet Type code must be shifted left by this number of bits to properly place it within the most-significant byte of the CCSDS Primary Header's Packet Identification word (see Section 3.1.1.1).

3.2.15 DOWNLINK_SEC_HDR_FLAG_MASK (constant value 0x001)

This mask, when "AND"ed with a Secondary Header Flag, zeros the upper unused bits (see Section 3.1.1.1).

3.2.16 DOWNLINK_SEC_HDR_FLAG_SHIFT (constant value 3)

A Secondary Header Flag must be shifted left by this number of bits to properly place it within the most-significant byte of the CCSDS Primary Header's Packet Identification word (see Section 3.1.1.1).

3.2.17 DOWNLINK_UPPER_APP_ID_MASK (constant value 0x007)

This mask, when "AND"ed with the most-significant byte of an Application Id, zeros the upper unused bits (see Section 3.1.1.1).

3.2.18 DOWNLINK_UPPER_APP_ID_SHIFT (constant value 0)

The most-significant byte of an Application Id must be shifted left by this number of bits to properly place it within the most-significant byte of the CCSDS Primary Header's Packet Identification word (see Section 3.1.1.1).

3.2.19 DOWNLINK_SEG_FLAGS_MASK (constant value 0x003)

This mask, when "AND"ed with a Segmentation Flags code, zeros the upper unused bits (see Section 3.1.1.1).

3.2.20 DOWNLINK_SEG_FLAGS_SHIFT (constant value 6)

A Segmentation Flags code must be shifted left by this number of bits to properly place it within the most-significant byte of the CCSDS Primary Header's Packet Sequence Control word (see Section 3.1.1.1).

3.2.21 DOWNLINK_UPPER_SEQ_NUM_MASK (constant value 0x003f)

This mask, when "AND"ed with the most-significant byte of a Source Sequence Count, zeros the upper unused bits (see Section 3.1.1.1).

3.2.22 DOWNLINK_UPPER_SEQ_NUM_SHIFT (constant value 0)

The most-significant byte of a Source Sequence Count must be shifted left by this number of bits to properly place it within the most-significant byte of the CCSDS Primary Header's Packet Sequence Control word (see Section 3.1.1.1).

3.2.23 DMA_DOWN_CHANNEL (DMA)

This is the DMA controller channel used for transferring downlink data packets (downlink channel).

3.2.24 FS_DOWNLINK_SIZE

This reflects the total downlink CCSDS packet size in bytes.

3.2.25 MSG_DOWNLINK_POST_BAD_LENGTH

This message code is passed to UTILpanic() if DOWNLINKpost() is passed a "length" argument which is either odd, or, when added to the header length, exceeds the maximum size of a downlink packet (see Section 3.2.24). The message should read something like:

"PANIC: -- DOWNLINKpost(%d) --Bad Length"

where %d represents the "length" argument passed to the function.

3.2.26 MSG_DOWNLINK_POST_BAD_PRIORITY

This message code is passed to UTILpanic() if DOWNLINKpost() is passed an invalid channel "priority" argument. The message should read something like:

"PANIC: -- DOWNLINKpost(%d) --Bad Priority"
where %d represents the priority argument passed to the function.

3.2.27 MSG_DOWNLINK_BAD_XFR

This message code is passed to UTILpanic() if downlink_complete_callback() detects that a DMA transfer didn't complete properly. The message should read something like:

"PANIC: -- downlink_complete_callback(%d) --Bad Transfer Length"
where %d represents the length of the transfer, as passed by the DMA Module's Downlink interrupt handler.

3.2.28 APPLICATION ID's

A list of Application Id's, listed by Event Analyzer Id, are provided in the **FDS to EDS Interface Control Document**, "CCSDS Telemetry Primary Header Format."

3.2.29 STREAM ID's

The following are the currently defined Packet Streams. A Packet Stream consists of a series of similarly formatted downlink packets. The data in each packet has some relationship to previously downlinked packets of the same type (that's why the word "stream" is used). This module has no knowledge of the internal formats of these types of packet.

3.2.29.1 STREAM_ID_STARTUP_MSG

This type of packet contains a startup message.

3.2.29.2 STREAM_ID_PANIC_MSG

This type of packet contains a panic message.

3.2.29.3 STREAM_ID_CMD_ACK

This type of packet contains a command acknowledgment.

3.2.29.4 STREAM_ID_SW_HOUSE

This type of packet contains Flight Software Housekeeping data.

3.2.29.5 STREAM_ID_ASM_SENSOR

This type of packet contains ASM sensor data.

3.2.29.6 STREAM_ID_OBS_PARAM

This type of packet contains Observation Parameter Dump data.

3.2.29.7 STREAM_ID_PARTHDR

This type of packet contains Partition Header information.

3.2.29.8 STREAM_ID_PARTDATA

This type of packet contains Partition data.

3.2.29.9 STREAM_ID_OBS_CODA

This type of packet contains Observation Coda data.

3.2.29.10 FS_STREAM_DATA_SIZE

This is the number of bytes available in a telemetry packet for stream specific data. Although this value is defined as a system global constant, it is coupled to the total size of a telemetry packet, FS_DOWNLINK_SIZE (see Section 3.2.24), and the total size of the Downlink Packet Header (see Section 3.1.1).

4 Interrupts

4.1 Downlink Interrupt (DMA)

The Downlink Module expects the Downlink interrupt handler, in the DMA Module, to call downlink_complete_callback() whenever this interrupt is caused by completion of data transfer on the DMA downlink channel. This function is responsible for releasing the buffer used by the downlink packet back to the buffer pool, and for programming the DMA controller to transfer the next packet from the downlink queue (if any).

5 Functions

5.1 DOWNLINKinit() - Initialize the Downlink Module

Usage

void DOWNLINKinit(void);

Description

DOWNLINKinit() initializes the Downlink Module. It initializes module specific values and obtains the initial start time, for use as an EA Run Id, from the **TIMING Module**. This function must be called before any other functions in this module. Any previously posted, untransmitted data, will not be transmitted.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) This function must be called before any other function in this module.

2) All data previously posted but not yet transmitted, will not be transmitted.

See Also

Timing Module Specification

5.2 DOWNLINKpost() - Downlink Data

Usage

void DOWNLINKpost (UINT8 **buf*, UINT16 *length*,
UINT8 *priority*, UINT8 *stream_id*);

Description

DOWNLINKpost() posts a data packet to be asynchronously transferred to the Flight Data System over the DMA downlink channel. This function is passed a pointer to a buffer "*buf*" to be downlinked. The data portion of this buffer must be filled by the caller. This function builds the downlink packet header in accordance with the passed parameters.

"*length*" specifies the size of the data portion of the packet. "*priority*" indicates which of the four priority levels to use when sending the packet. "*stream_id*" identifies the format of the data pointed to by "*buf*" and the packet stream to which the posted packet belongs.

This function returns immediately after posting the packet. The transfer is completed at a later time.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) DOWNLINKinit() must be called before calling this function.

See Also

DOWNLINKinit().

5.3 DOWNLINKpanic_send() - Send a Panic Message

Usage

void DOWNLINKpanic_send(UINT8 *buf, UINT16 *msg_code*, UINT32 *arg*);

Description

DOWNLINKpanic_send() synchronously sends a panic message to the Flight Data System via the DMA downlink channel. The message consists of one 16-bit message code, *msg_code*, and a 32-bit message argument value, "*arg*". "*buf*" points to the telemetry buffer to use for the panic message. The panic message is sent using priority level PKT_PRIORITY_HIGH. This call doesn't return until the message has been completely transferred. The DMA transfer of the panic message is programmed as soon as the transfer currently underway is completed, or times out.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) DOWNLINKinit() must be called before calling this function.

See Also

DOWNLINKinit(), **Utility Module Specification**.

5.4 DOWNLINKdata() - Get Pointer to Data Portion of Buffer

Usage

UINT8 *DOWNLINKdata (UINT8 * *buffer*);

Description

DOWNLINKdata() allows the client of the Downlink Module to get the pointer to the data portion of a downlink packet buffer.

Return Values

Pointer to the data portion of a downlink buffer.

Caveats

The caller must make sure that the pointer passed to this function is really a pointer to a buffer obtained from a buffer pool with the buffer size not less than FS_DOWNLINK_SIZE.

5.5 DOWNLINKidle() - Indicate Idle Telemetry System

Usage

BOOL DOWNLINKidle (void)

Description

DOWNLINKidle() indicates when there are no telemetry packets being sent or waiting to be sent out of the Event Analyzer.

Return Values

TRUE - No telemetry packets are being or waiting to be sent

FALSE - Telemetry is either being transferred or is waiting to be sent

Caveats

1) DOWNLINKinit() must be called before calling this function.

See Also

DOWNLINKinit(), **DMA Module Specification**.

5.6 downlink_complete_callback() - DMA Complete Callback

Usage

void downlink_complete_callback (char **buf*, UINT16 *xfer_cnt*);

Description

downlink_complete_callback() must be called once every Downlink interrupt caused by the completed downlink packet transfer. This function releases the buffer, which contained the downlink packet, back to the buffer pool and programs the DMA transfer of the next packet in the downlink queue (if any). The DMA Module's Downlink interrupt handler must call this function with two parameters: *buf* points to the buffer that has been transferred, *xfer_count* is the number of bytes actually transferred.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) DOWNLINKinit() must be called before using this function.

2) This module expects this function to be called once every Downlink interrupt caused by the completed downlink packet transfer.

See Also

DOWNLINKinit(), **DMA Module Specification**

6 Detailed Design Issues

6.1 Data Flow

This section describes the Level 1 data flow within the Downlink Module.

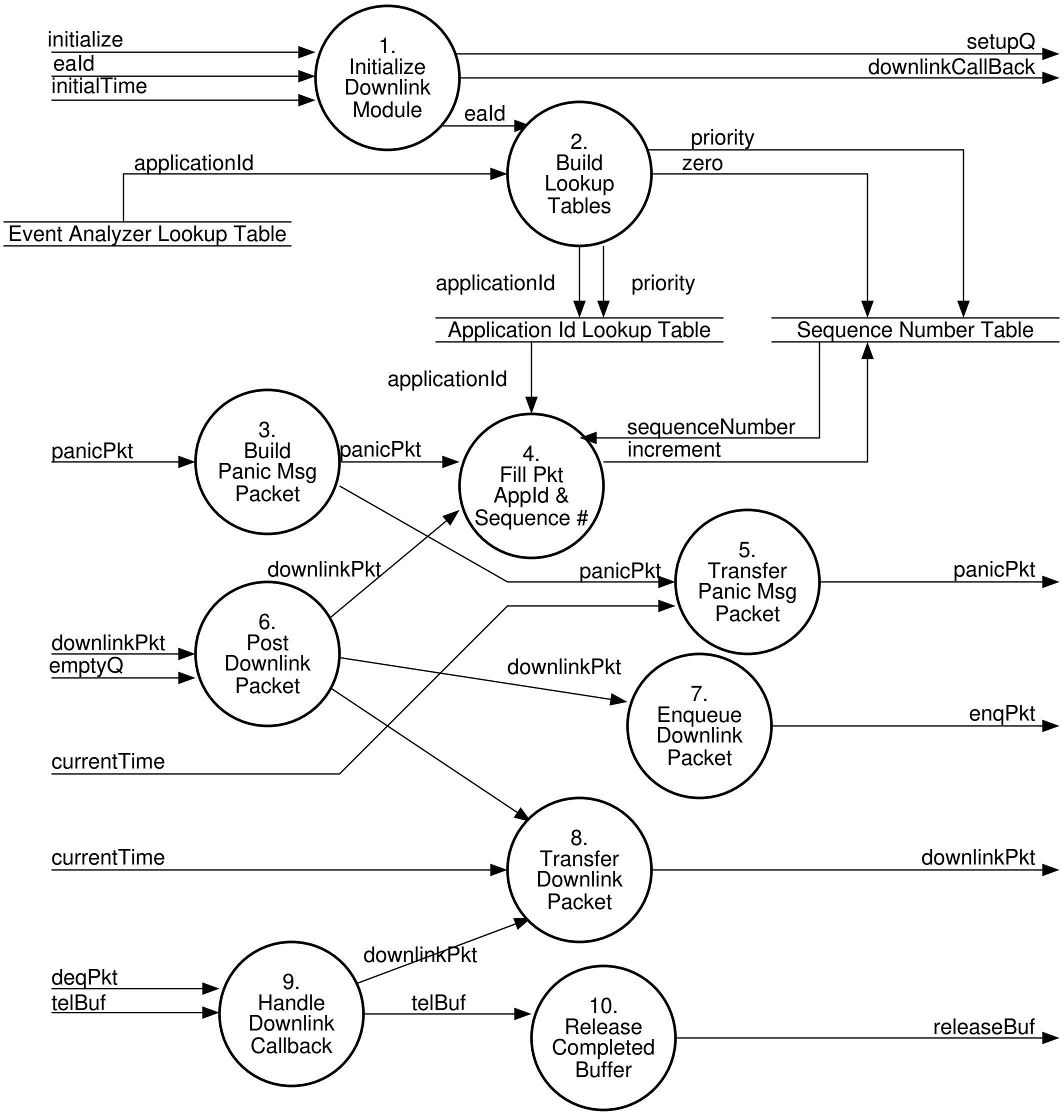

Figure 4 Downlink Level 1 Data Flow

During Downlink's initialization (bubble 1), the module initializes its downlink queue (setupQ), installs its downlink callback function (downlinkCallBack). It also initializes an internal FDS priority level to application id lookup table (Application Id Lookup Table) using the event analyzer id (eaId), and a common table which contains a list of the application ids, indexed by event analyzer id and priority level (bubble 2). The initialization process also zeros the current sequence numbers for all of the FDS priority levels (Sequence Number Table, priority, zero), and gets the MET read during system initialization (initialTime) which is used later as the EA Run Id.

If a panic message packet (panicPkt) or telemetry packet (downlinkPkt) is sent to the Downlink Module, the module fills in the packet's header information (bubble 4), including the EA Run Id (not shown), looking up the application id (applicationId) from the application id table, and obtaining the current sequence number for the requested priority level (sequenceNumber). It then increments the current sequence number for that level (increment).

If the packet was a panic message (panicPkt, bubble 3), the module timestamps (currentTime) the packet and transfers it synchronously to the FDS (bubble 5). If the packet is a normal telemetry packet (downlinkPkt, bubble 6), it tests to see if the downlink queue is empty (emptyQ) and that the DMA's downlink channel is idle (not shown). If the queue is empty and the DMA's downlink channel is not performing a transfer, the Downlink module time stamps the packet (currentTime) and starts an asynchronous transfer (bubble 8). If the queue is not empty, or a downlink transfer is in progress, the Downlink Module enqueues the packet to the downlink queue (enqPkt, bubble 7).

When a transfer completes, the downlink callback function is invoked (bubble 9). This function is passed a pointer to the buffer that was just transferred (telBuf). The callback function then releases the buffer (releaseBuf) to its buffer pool (bubble 10). It then dequeues the next packet from the downlink queue (deqPkt), and starts its transfer (bubble 8).

6.2 Control Considerations

In this section the flowcharts of two of this module's functions are presented: DOWNLINKpost() and downlink_complete_callback(). They are designed in such a way that the DOWNLINKpost() function can be called from the interrupt handlers by using an internal semaphore. The Downlink Module uses a semaphore to indicate when the DMA Downlink Channel is being directed to start a transfer.

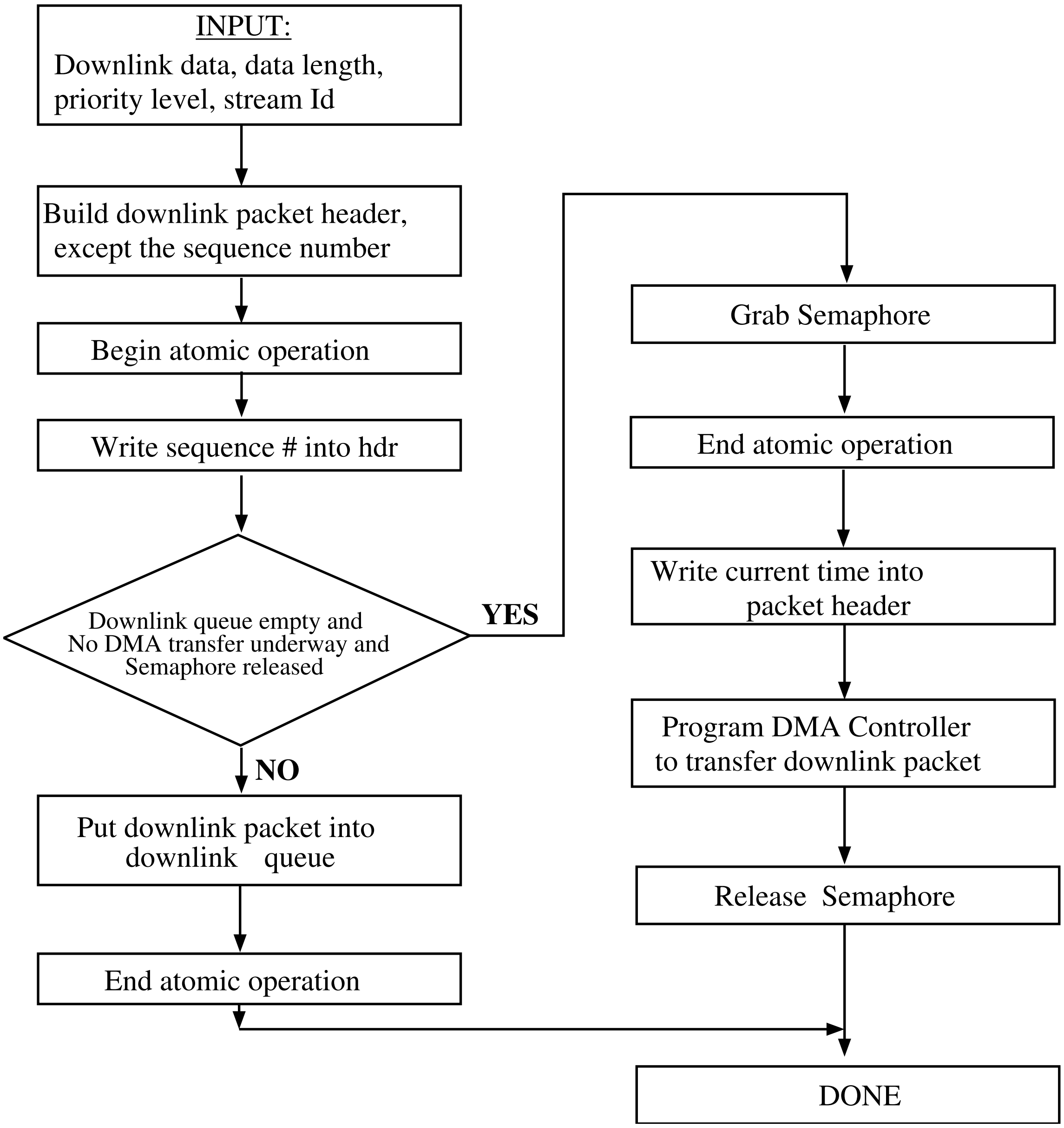

Figure 5 DOWNLINKpost() Flowchart

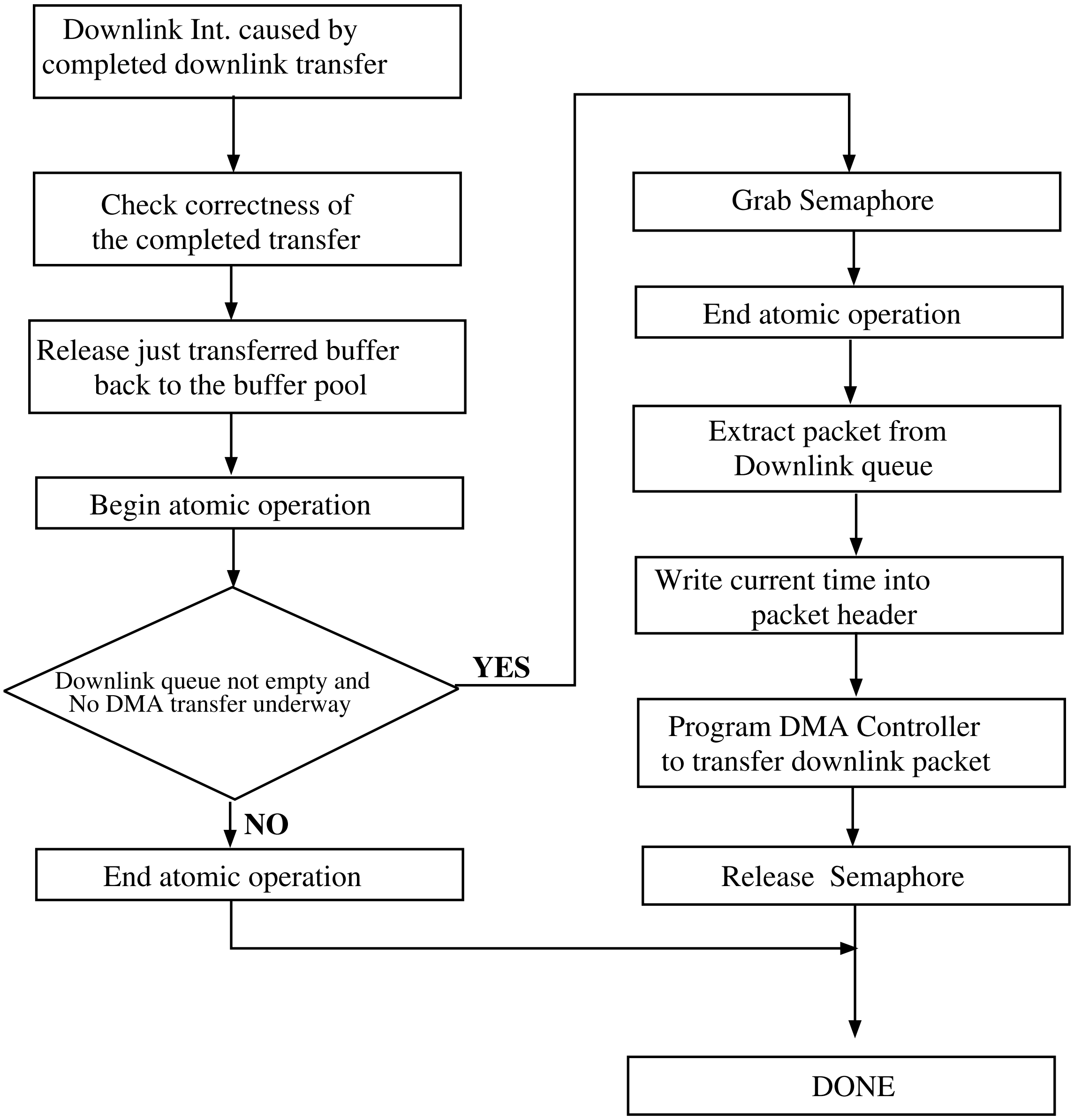

Figure 6 downlink_complete_callback() Flowchart

The above flowcharts are self-explanatory. The following just explains several design details.

Both DOWNLINKpost() and downlink_complete_callback() contain a critical section - programming the DMA controller to transfer a downlink packet. If this section is interrupted, functions called as a result of the interrupt must not program the DMA controller (otherwise all the downlink subsystem's operation will be disrupted). To ensure this, a Semaphore is introduced that protects the critical section from multiple concurrent access.

The status of the Semaphore is not tested before entering the critical section in the downlink_complete_callback() function because the Downlink Interrupt caused by the downlink transfer completion can not occur while the DMA controller is being programmed.

An alternative to the semaphore scheme would have been to extend the atomic operation until after the DMA channel has been configured. A design decision was made to keep the atomic operation as short as possible, allowing high priority interrupts to occur while configuring the DMA Controller.

When the DOWNLINKpost() function can not program the DMA controller immediately, it must post the packet to the downlink queue in one atomic step. Otherwise the following situation may occur:

1) DOWNLINKpost() is called to post the ***packet j***. At the moment of the DOWNLINKpost() call the downlink queue is empty but the DMA controller is busy transferring another downlink ***packet i***.

2) The DOWNLINKpost() starts enqueuing the ***packet j*** into the downlink queue.

3) The enqueueing operation gets interrupted by the Downlink Interrupt caused by completion of the ***packet i*** transfer. The downlink_complete_callback() finds that the downlink queue is empty and returns.

4) ***packet j*** enqueueing completes.

All subsequent DOWNLINKpost() calls will find the downlink queue not empty and no other downlink packets will get transferred to the FDS.

6.3 Module Structure

6.3.1 Downlink Exported Function Structures

This section describes the structure of the functions exported by the Downlink Module. Bolded boxes and text represent functions which are provided by modules other than the Downlink Module.

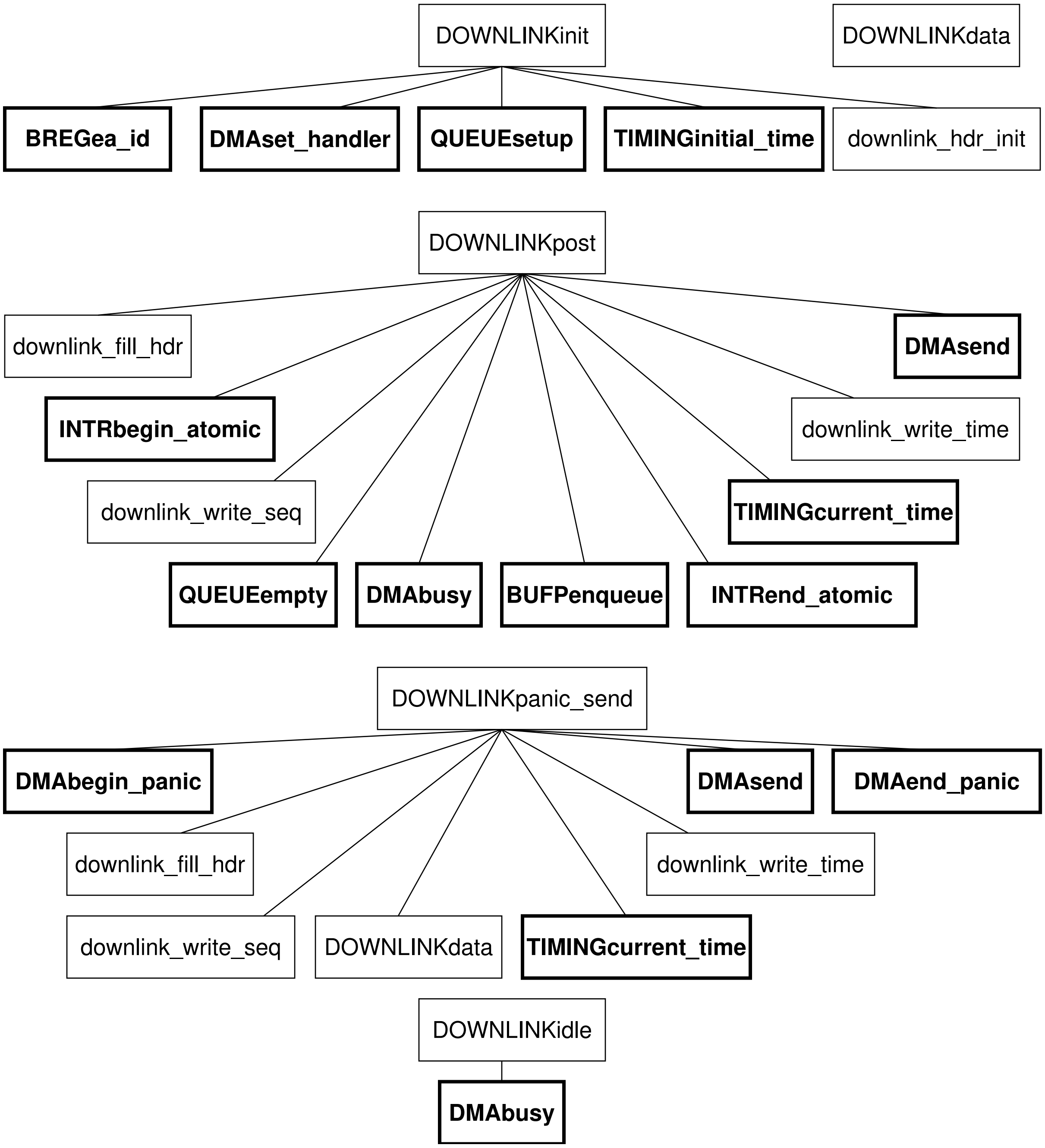

Figure 7 Downlink Exported Function Structures

The following describes each of the functions shown in Figure 7. The details of each function's operation and arguments are described in the Downlink Module's detailed design document.

DOWNLINKinit .... This function is called by the **Startup Module**, during system initialization. DOWNLINKinit uses the **Back End Registers Module** (BREGea_id) to obtain the event analyzer's id. It uses this id to index the appropriate set of CCSDS Application Ids, and fill in the priority level to Application Id table. DOWNLINKinit also zeros the sequence counts for each priority level. DOWNLINKinit uses the **DMA Module** (DMAset_handler) to install a downlink channel complete callback function, and uses the **Queue Utility Module** (QUEUEsetup) to initailize the downlink queue control structure. DOWNLINKinit() uses the **Timing Module** (TIMINGinitial_time) to get the system initialization time, and passes the obtained value to downlink_hdr_init to construct a telemetry packet header template, which is used later by downlink_fill_hdr when forming the complete telemetry packet headers.

DOWNLINKdata .... This function is used by the Downlink Module and other Flight Software modules to obtain a pointer to the application data area of a telemetry packet buffer.

DOWNLINKpost .... This function is called by other Flight Software modules to post a telemetry packet for transfer to the FDS. DOWNLINKpost fills in the CCSDS Primary header of the packet, and the EDS-specific header by calling downlink_fill_hdr. It then begins an atomic sequence, disabling interrupts using the **Interrupt Manager Module** (INTRbegin_atomic). Once interrupts are disabled, DOWNLINKpost obtains the current sequence number for the specified priority level and stores the count in the CCSDS Primary header using downlink_write_seq. It then checks to see if it needs to initiate a DMA transfer, or just enqueue the packet. If the downlink queue is empty, determined using the **Queue Utility Module** (QUEUEempty), and the DMA is not performing a downlink transfer, determined using the **DMA Module** (DMAbusy), and it is not interrupting a transfer setup in progress, determined by the state of an internal semaphore, DOWNLINKpost grabs the semaphore, and ends the atomic sequence, using the **Interrupt Manager Module** (INTRend_atomic). It then obtains the current time, using the **Timing Module** (TIMINGcurrent_time), stores the time, using downlink_write_time, and starts the DMA transfer, using the **DMA Module** (DMAsend). Once the transfer is started, DOWNLINKpost releases the semaphore. If the downlink queue isn't empty, or the DMA is performing a downlink transfer, or the semaphore is locked, DOWNLINKpost enqueues the packet, using the **Buffer Pool Module** (BUFPenqueue), and ends the atomic sequence, using the **Interrupt Manager Module** (INTRend_atomic).

DOWNLINKpanic_send .... This function is called by the **Utility Module** (UTILpanic) to send a panic message packet to the ground. DOWNLINKpanic_send starts the panic sequence using the **DMA Module** (DMAbegin_panic) to wait for the current downlink transfer to complete or timeout. It then fills in the CCSDS Primary header of the packet, and the EDS-specific header by calling downlink_fill_hdr, and writes the sequence number using downlink_write_seq. DOWNLINKpanic_send then obtains a pointer to the packet's data area, using DOWNLINKdata, and fills the area with the passed message code and argument value. DOWNLINKpanic_send then obtains the current time, using the **Timing Module** (TIMINGcurrent_time), stores the time in the packet, using downlink_write_time, and starts the DMA transfer, using the **DMA Module** (DMAsend). Once the transfer has started, DOWNLINKpanic_send waits for the transfer to complete or timeout, using the **DMA Module** (DMAend_panic).

DOWNLINKidle .... This function is called by other Flight Software modules to determine if telemetry is being transferred out of the EA or is awaiting transfer. DOWNLINKidle tests the telemetry queue, queries the **DMA Module** (DMAbusy) and tests its internal semaphore. If the downlink queue is empty, the DMA is not busy, and the Downlink Module is not in the process of setting up a transfer, DOWNLINKidle indicates that the telemetry system is idle. Otherwise, it indicates that packets are in the process of being sent out of the EA.

6.3.2 Downlink Local Function Structures

This section describes the structure of the functions interal to the Downlink Module. Bolded boxes and text represent functions which are provided by modules other than the Downlink Module.

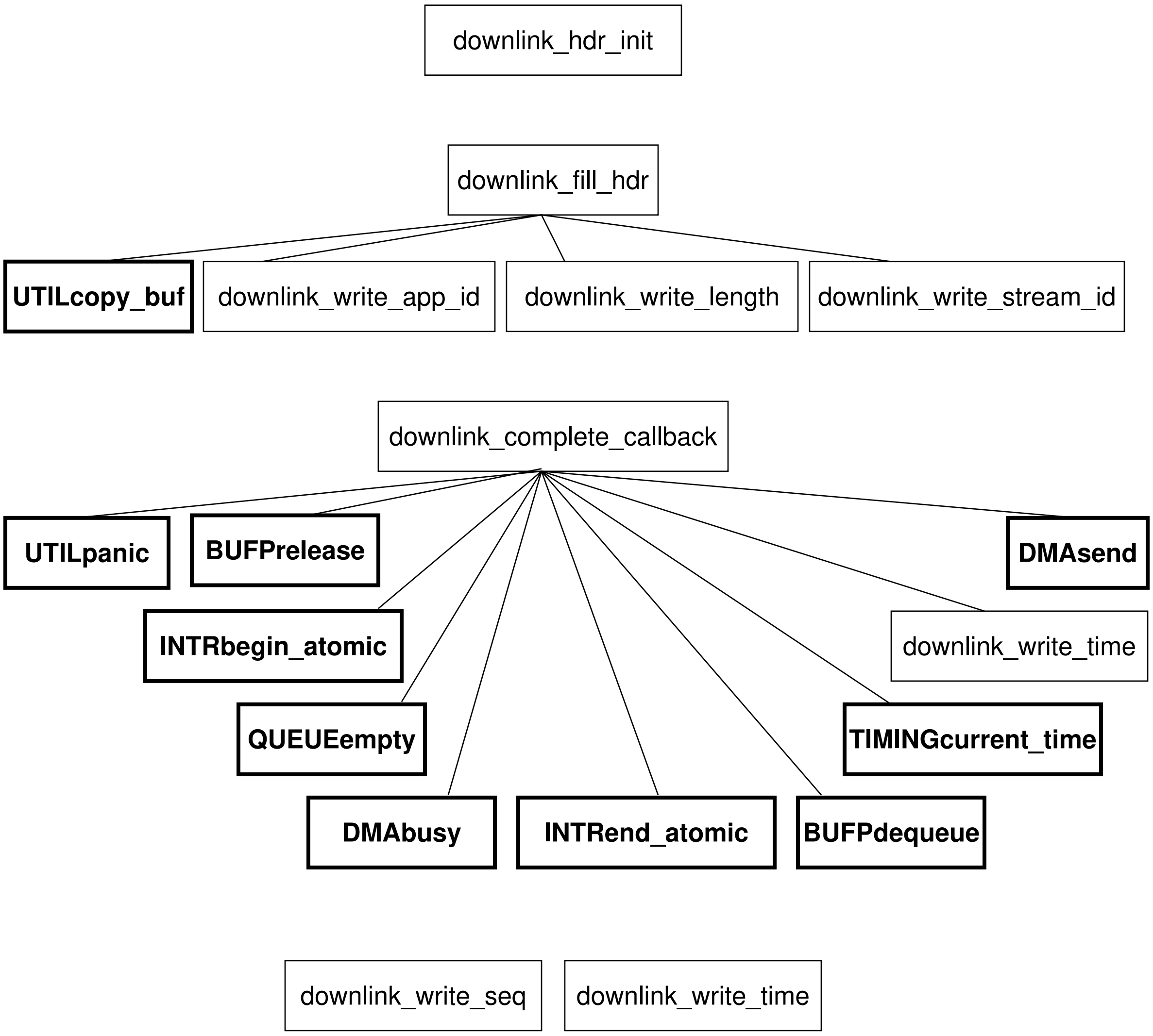

Figure 8 Downlink Local Function Structures

The following describes each of the functions shown in Figure 8. The details of each function's operation and arguments are described in the Downlink Module's detailed design document.

downlink_hdr_init .... This function is called by DOWNLINKinit. downlink_hdr_init constructs a downlink packet header template, which is used by downlink_fill_hdr when constructing the header portion of a given downlink packet. This function is passed, by DOWNLINKinit, the initial MET value, read during EA initialization, which is used as the EA Run Id. This value, in conjunction with other constants are used to produce a header template. This function initializes all variable fields within the header to zero.

downlink_fill_hdr .... This function is called by DOWNLINKpost and DOWNLINKpanic_send to fill most of the contents of the CCSDS Primary Header, and the EDS-specific header. This function first copies the downlink packet header template onto the passed packet header, using the **Utility Module** (UTILcopy_buf). It then uses downlink_write_app_id, downlink_write_length, and downlink_write_stream_id to set the corresponding variable fields in the headers (note: the "send time" field in the secondary header is not set until the packet transfer starts).

downlink_complete_callback .... This function is installed by DOWNLINKinit, using the **DMA Module** (DMAset_handler), and is called by the **DMA Module** whenever a DMA downlink transfer completes. When this function is invoked, it checks the previous transfer, and if the transfer encountered an error, issues a panic message and resets the EA, using the **Utility Module** (UTILpanic). If the previous transfer completed properly, it returns the buffer to its pool, using the **Buffer Pool Module** (BUFPrelease). The callback function then starts an atomic sequence, using the **Interrupt Manager Module** (INTRbegin_atomic). It then tests to see if it needs to start a new transfer. If the downlink queue is not empty, as indicated by the **Queue Utility Module** (QUEUEempty), and a DMA downlink transfer is not in progress (i.e. a new transfer was started by an interrupt handler of a higher priority than DMA), as indicated by the **DMA Module** (DMAbusy), downlink_complete_callback starts the next transfer. It starts by grabbing the semaphore, and then ends the atomic sequence, using the **Interrupt Manager Module** (INTRend_atomic). The callback then dequeues a packet from the downlink queue, using the **Buffer Pool Module** (BUFPdequeue). It then obtains the current time, using the **Timing Module** (TIMINGcurrent_time) and stores the time in the packet, using downlink_write_time. The callback then starts the transfer, using the **DMA Module** (DMAsend), and releases the semaphore. If a transfer is not started, the callback just ends the atomic sequence, using the **Interrupt Manager Module** (INTRend_atomic) and returns.

downlink_write_seq .... This function is called by DOWNLINKpost and DOWNLINKpanic_send to write the current sequence number into a packet's CCSDS Primary header (see Section 3.1.1.1).

downlink_write_time .... This function is called by DOWNLINKpost, DOWNLINKpanic_send, and downlink_complete_callback to write the current time into the CCSDS secondary (timestamp) header (see Section 3.1.1.2).

downlink_write_app_id .... This function is called by downlink_fill_hdr to write the Application Id field of the CCSDS Primary Header (see Section 3.1.1.1).

downlink_write_length .... This function is called by downlink_fill_hdr to write the Length field of the CCSDS Primary Header (see Section 3.1.1.1). This function expects to be passed the length of the packet, minus the size of the CCSDS Primary header. This function decrements the passed length before writing to meet the CCSDS specification.

downlink_write_stream_id .... This function is called by downlink_fill_hdr to write the Stream Id and reserved fields of the EDS Header (see Section 3.1.1.3).

7 Requirements Traceability

7.1 Science Requirements

This module does not address any specific Science Requirements, beyond the ability to build and send telemetry packets to the FDS. It does, however, adhere to the requirements specified in the **FDS to EDS Interface Control Document**.

8 Testing Considerations

8.1 Unit Testing

To perform unit tests on this module, the following equipment is required:

a. An Event Analyzer equipped with the Intermetrics ROM Monitor

b. A workstation running the Intermetrics debugger

c. An RS-232 serial cable between the ROM Monitor's serial port and the workstation

d. A logic analyzer

To unit test this module, the tester must link and locate this module's functions to some EA RAM memory location. Using a resident monitor/debugger, the tester should first verify the operation of all functions internal to this module, and then verify the interactions between all functions internal to the module. These tests should verify all branches, calls, and legal and illegal parameters. The tester should then verify all calls made to functions outside of this module, and ensure that the passed arguments are always within the limits specified by the corresponding module specifications. The tester should also verify that the caller always handles values returned by these functions in an appropriate manner.

In addition to these tests, the tester should ensure that all telemetry packets produced by this module conform to the requirements specified in the **FDS to EDS Interface Control Document**. The tester should also ensure that the module operates correctly when packets are posted by interrupt handlers. The tester should verify that panic messages are sent properly, under a variety of circumstances.

8.2 Integration Testing

For integration tests, the following equipment is required:

a. An Event Analyzer

b. A populated ROM Repository board

c. A logic analyzer

d. Hardware which can:

    1. Receive telemetry packets via the RS422 Interface (S/C Simulator)

The integration tests should verify the operation of the Downlink Module in conjunction with the actual hardware and other FS modules. These tests should include:

a. Verify that telemetry packets are properly formed, enqueued and transferred out of an EA.

b. Verify that panic message packets are properly formed, and transferred out of an EA.

c. Measure packet construction times, and Downlink Interrupt processing time to determine telemetry throughput capabilities.

Appendix A

A.1 Items Under Review

This section lists items and issues which need to be resolved:

Document Reference	Section	Resolution Expected by
NONE		