|  |  |  |
|---|---|---|
| CSR | MASSACHUSETTS INSTITUTE OF TECHNOLOGY<br>CENTER FOR SPACE RESEARCH<br>CAMBRIDGE, MASSACHUSETTS 02139 |  |
| REVISION LOG | TITLE: Software Detailed Design Telemetry Management | DOC. NO. 36-53215 |

| Revision | Date (mm/dd/yy) | ECO No. | Page(s) Affected | Reason | Approval |
|---|---|---|---|---|---|
| — | 2/21/95 | — | all | Initial version for design walk-through |  |
| A | 5/10/95 | 36-248 | all | Incorporate walk-through comments | [illegible] 5/22/95 |

# 15.0 Telemetry Management Classes (36-53215 A)

## 15.1 Purpose

The purpose of the Telemetry Management system is to build and send a time-ordered list of telemetry packets to the instrument's telemetry hardware.

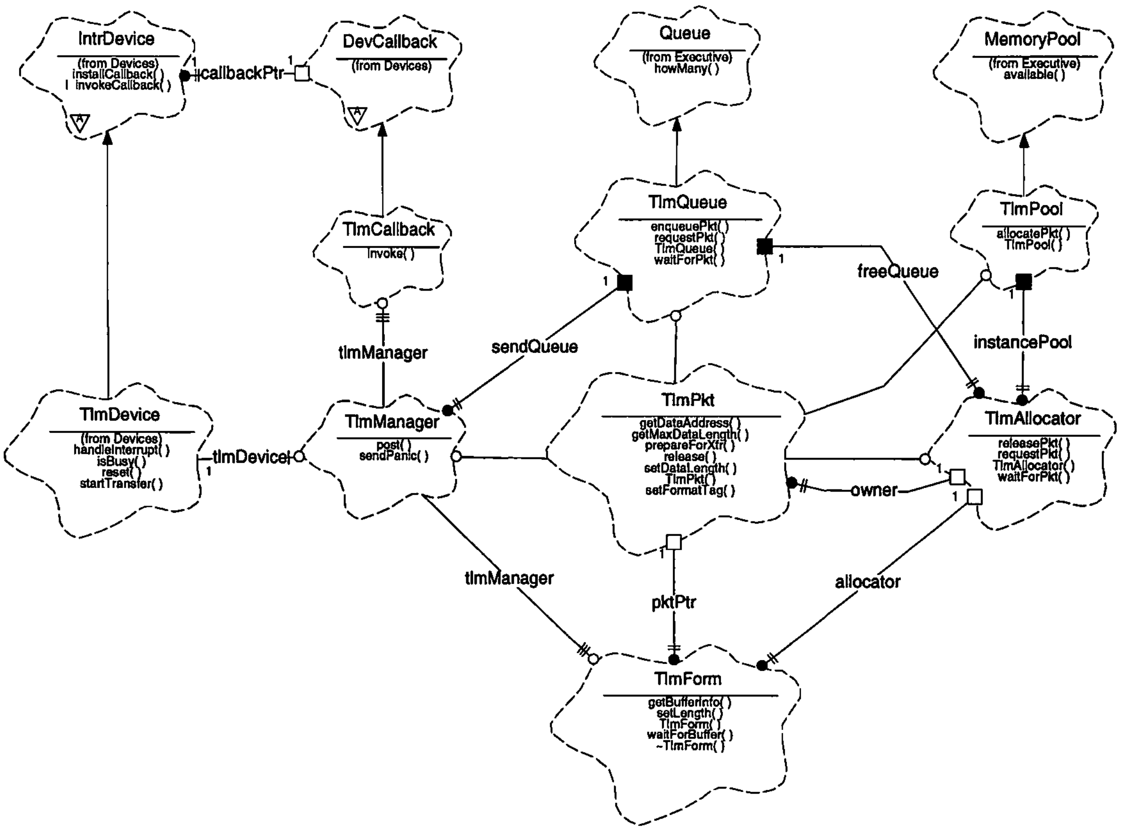
## 15.2 Uses

Use 1:: Allocate telemetry buffers and control structures
Use 2:: Manage the ordered transfer of telemetry information to the instrument hardware
Use 3:: Provide support for a final fatal error telemetry message prior to controlled crashes of the instrument software.

## 15.3 Organization

Figure 55 illustrates the top-level classes and their relationships involved in processing telemetry.

**FIGURE 55. Telemetry Management Class Relationships**



**TlmManager** - The **TlmManager** class is responsible for maintaining a queue of telemetry packets (see **TlmQueue** and **TlmPkt**) to feed to the telemetry device (***Devices***::**TlmDevice**).

**TlmPkt** - The **TlmPkt** class is responsible for managing the raw telemetry buffer located in Single Event Upset (SEU) vulnerable RAM, and for maintaining a small amount of critical packet header information within SEU immune RAM. This class serves as the container for telemetry information within the ACIS software.

**TlmAllocator** - The **TlmAllocator** class acts as a memory manager for a group of telemetry packet buffer types. It maintains a collection of equally sized **TlmPkt** instances. The allocator class uses the **TlmPool** class to obtain a set of **TlmPkt** instances during start-up, and then uses a **TlmQueue** to manage these instances at run-time.

**TlmQueue** - The **TlmQueue** class is a subclass of ***Executive***::**Queue**. It serves as a type-safe interface to a **Queue** instance which maintains an first-in/first-out list of pointers to **TlmPkt** instances. This class is used by the **TlmAllocator** to maintain its list of free telemetry packets. It is also used by the **TlmManager** to maintain a list of active telemetry packets awaiting transfer out of the instrument.

**TlmPool** - The **TlmPool** class is a subclass of ***Executive***::**MemoryPool**. It serves as a type-safe interface to a **MemoryPool** instance which contains the memory buffers used for **TlmPkt** instances. This class is used by the **TlmAllocator** during start-up to reserve memory for a set of **TlmPkt** buffers. The intent of this class is to provide the ability to patch the number or size of **TlmPkts** within a given pool instance, if the unlikely need arises after launch.

**TlmForm** -The **TlmForm** class is used as a base-class for the different telemetry packet formats. It and its subclasses are responsible for providing the interface functions needed to allocate telemetry packets, format the contents of the packets, and post the packets to the **TlmManager** for transfer out of the instrument. By convention, the name of all subclasses of **TlmForm** shall be prefixed with the abbreviation, "**Tf**" (i.e. a command log telemetry format class would be called **TfCmdLog**).

**TlmDevice** - The ***Devices***::**TlmDevice** is responsible for commanding the hardware to transfer a telemetry packet. It is a subclass of ***Devices***::**IntrDevice**, and is provided by the ***Devices*** class category. It is described in Section 9.0 .

**TlmCallback** - The **TlmCallback** class is a subclass of ***Devices***::**IntrCallback**. It is used by the **TlmManager** to obtain control during telemetry interrupt processing.

## 15.4 Telemetry Processing Assumptions and Restrictions

### 15.4.1 Transfer buffers and packet formats

The hardware requires that all telemetry transfers must be from uncached RAM. In ACIS, this memory is vulnerable to Single-Event Upsets (SEU). In order to minimize the possibility of an upset corrupting crucial buffer management information, or important packet header information, the ACIS software maintains this information in its data cache space, which uses SEU immune RAM. This information is contained within a **TlmPkt** class instance. Each **TlmPkt** instance has a pointer to a packet transfer buffer within uncached RAM. In general, memory for **TlmPkt** instances are reserved using **TlmPool** instances (which use ***Nucleus RTX*** Memory Partitions), and lists of **TlmPkt**'s are maintained using **TlmQueue** instances (which use ***Nucleus RTX*** Queues).

During operation, all telemetry information except for the packet headers is written directly into a packet's transfer buffer by a **TlmForm** instance (or by a user of **TlmForm**). Except for the telemetry packet header, once a packet has been formatted and posted, no further modifications to a telemetry packet are needed. Telemetry buffer management and formatting are treated as two different activities. The **TlmPkt** class is used for buffer management, and the **TlmForm** subclasses are used for formatting the buffer contents. Only **TlmPkt** instances need to be dynamically allocated and released at run-time.

### 15.4.2 Packet Formats

#### 15.4.2.1 Packet Header

The ACIS produces telemetry packets asynchronously to the spacecraft telemetry frames, and these packets can appear anywhere within the telemetry frame science data sections allocated for ACIS by the spacecraft. Each packet consists of a series of 32-bit words. In order to allow the ground software to easily locate the start of an ACIS telemetry packet within the science stream, ACIS provides a synchronization word at the start of each packet. Gaps between ACIS packets are filled using a hardware generated fill pattern (see Section 9.4 ). Within the instrument software, the format of this header is managed by the **TlmPkt** class. Table 16 illustrates the format of the ACIS telemetry packet header.

**TABLE 16. Telemetry Packet Header**

| Word | Bits | Field | Description | Min. Value | Max Value |
|---|---|---|---|---|---|
| 0 | 0:31 | Packet Synch. | This field contains a constant 32-bit synch. pattern, used to locate the start of a telemetry packet within a spacecraft minor frame. | 0x4329da2c | N/A |
| 1 | 0:9 | Packet Length | This field contains the total number of 32-bit words in the packet, including the Packet Synch. | 2 | 1023 |

**TABLE 16. Telemetry Packet Header**

| Word | Bits | Field | Description | Min. Value | Max Value |
|---|---|---|---|---|---|
| 1 | 10:15 | Packet Format Tag | This field identifies the type of data contained in the body of the packet. | 0 | 63 |
| 1 | 16:31 | Packet Sequence Number | This field contains an telemetry packet counter, which increments for each packet sent by ACIS | 0 | 65,535 |
| 2:Packet Length - 1 | - | Data | The remainder of the packet contains format tag-specific data. | - | - |

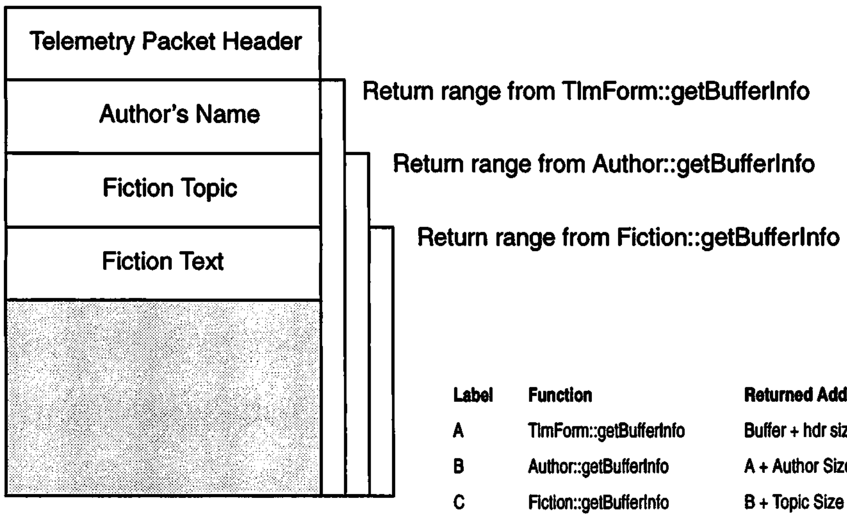
### 15.4.2.2 Packet Format Layers

Within the ACIS software, each packet data format is represented by some subclass of **TlmForm**. These subclasses may inherit directly from **TlmForm**, or may inherit from some intermediate abstract class which provides fields common to a set of different formats. Each intermediate layer may provide a set of fields which consume a portion of the packet data area. In order to de-couple the details of an intermediate layer from its subclasses, each intermediate class must overload the **TlmForm** functions, getBufferInfo() and setLength(). Any subclass of **TlmForm** or one of these intermediate classes must use getBufferInfo() to obtain the starting address of the region of the packet available for use by the child class, and setLength() to inform the parent class of how much data was actually written into the buffer.

Figure 56 illustrates a example situation. In this fabrication, there are two main types of telemetry packets, a "Fiction Packet", and a "Non-Fiction Packet." The "Fiction Packet" contains an author field, a topic field and a variable amount of fiction text. In a "Non-Fiction Packet," there is also an author, but instead of a fiction topic, there is a non-fiction reference number, which has a different length than fiction topic. In the example, an "Author Packet" is a subtype of **TlmForm**, which appends the "Author" field to a telemetry header. The two final packet types, "Fiction" and "Non-fiction" are subclasses of "Author," and append their respective information to the name field. The boxes in the figure represent portions of a given telemetry packet buffer.

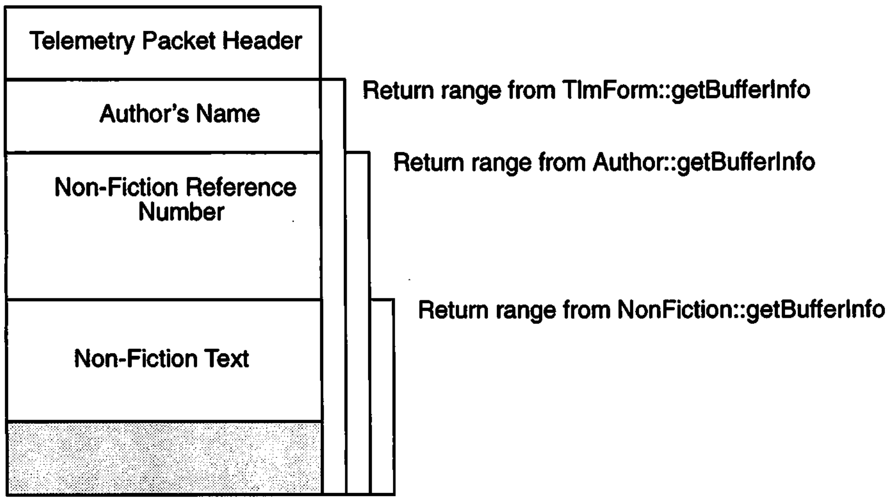
FIGURE 56. Example Telemetry Packet Formats

**"Fiction" Packet's Buffer**


Telemetry Packet Header
Author's Name
Fiction Topic
Fiction Text
Return range from TlmForm::getBufferInfo
Return range from Author::getBufferInfo
Return range from Fiction::getBufferInfo

| Label | Function | Returned Address | Returned Word Cnt |
|---|---|---|---|
| A | TlmForm::getBufferInfo | Buffer + hdr size | Bufsize - hdr size |
| B | Author::getBufferInfo | A + Author Size | A - Author Size |
| C | Fiction::getBufferInfo | B + Topic Size | B - Topic Size |
| D | Non-fiction::getBufferInfo | B + Reference Size | B - Reference Size |

**"Non-Fiction" Packet's Buffer**


Telemetry Packet Header
Author's Name
Non-Fiction Reference Number
Non-Fiction Text
Return range from TlmForm::getBufferInfo
Return range from Author::getBufferInfo
Return range from NonFiction::getBufferInfo

## 15.4.3 Transfer Turn-around Time

In order to reduce the number and size of gaps between telemetry packets when there are a set of packets ready to be sent, the Telemetry Device states that the time between the end of one telemetry transfer, and the start of the next start take less than 0.5 milliseconds (see Section 9.0 ). In order to avoid possible long delays due to context switching, the Telemetry Manager performs back-to-back telemetry transfers using the **`TlmDevice`**'s callback instance. When a telemetry transfer completes, the **`TlmDevice`**'s interrupt handler is invoked. This handler then invokes the installed callback, which in turn, calls the **`TlmManager`**'s `serviceDevice()` member function. During this interrupt callback processing, `serviceDevice()` then obtains the next packet from the telemetry queue, and starts the next transfer. If the total time spent handling a telemetry interrupt using a non-empty telemetry queue is less than 0.5 milliseconds (assuming no higher priority interrupts occur and that interrupts are in an enabled state when telemetry transfer completes), no gaps will occur between adjacent, prepared telemetry packets.

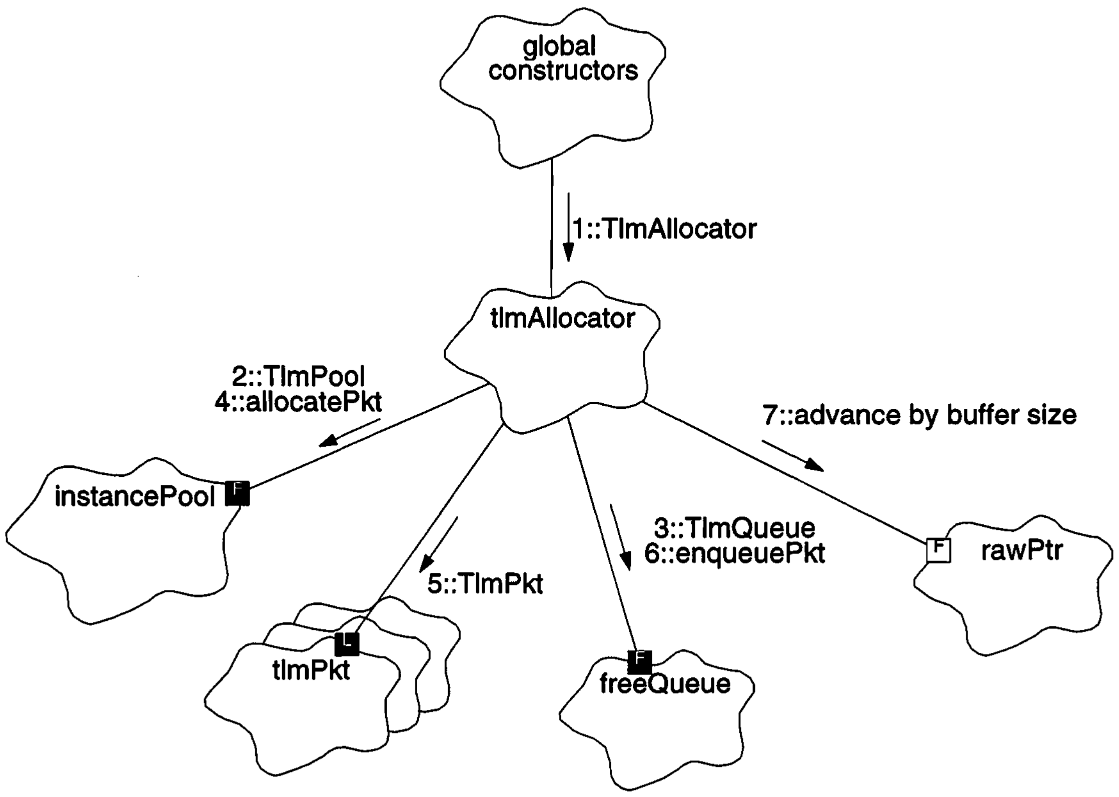
## 15.5 Scenarios

### 15.5.1 Use 1:: Allocate telemetry buffers and control structures

Within ACIS, memory for all telemetry buffers and control structures (**TlmPkt**) are reserved (via ***Nucleus RTX*** memory partitions), allocated and initialized during system initialization and start-up. Once running, groups consisting of equally sized telemetry packet buffers are maintained by **TlmAllocator** instances. Any given **TlmForm** uses a single **TlmAllocator** to obtain packet buffers, and each **TlmPkt** has a pointer back to its owning **TlmAllocator** instance.

Figure 57 illustrates the allocation of telemetry buffers and control structures during the system initialization process.

FIGURE 57. System Initialization Telemetry Packet Allocation



1. During system start-up, `__main()` is invoked, which calls the constructors for each global **TlmAllocator** instance.
2. *tlmAllocator* constructor's initialization section calls the constructor for its **TlmPool** instance, *instancePool*, verifying the size and number of the contained telemetry packet control structure instances. At this point, *instancePool* is ready to supply the buffers for all **TlmPkt** instances maintained by *tlmAllocator*.

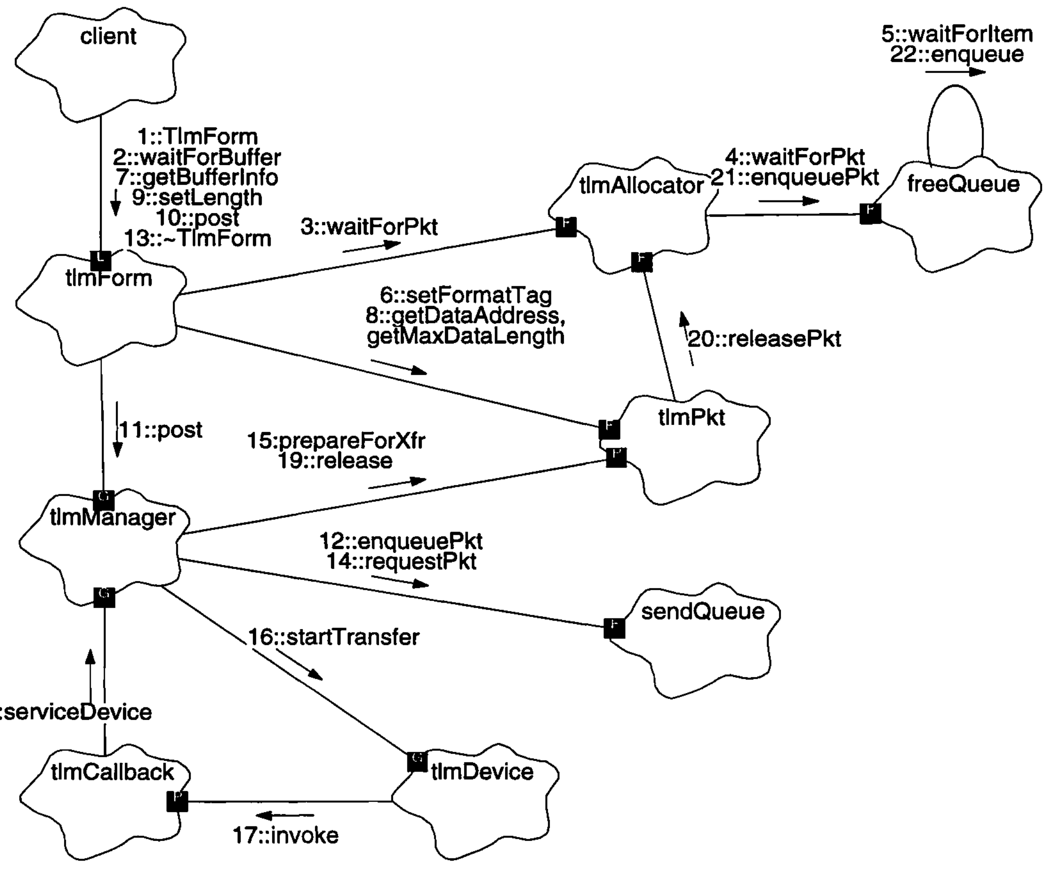
3. *tlmAllocator* constructor's initialization section calls the constructor for its **TlmQueue** instance, *freeQueue*, verifying the maximum number of packet pointers supported by the queue.
4. *tlmAllocator*'s constructor then enters its main body. The body consists of a loop which allocates space for each telemetry packet managed by the allocator instance. The loop starts by calling *instancePool*.allocatePkt() to obtain buffer space to contain a **TlmPkt** instance.
5. The loop then invokes the **TlmPkt** constructor on the obtained buffer, passing *this* as the owner of the packet, and the current value of *rawPtr* as the address of the memory buffer to use for the telemetry information.
6. The loop then invokes *freeQueue*.enqueuePkt() to place the packet on its list of available telemetry packets.
7. Finally, the loop advances *rawPtr* by the size of the telemetry packet buffer.

Once the allocator has been constructed, its *freeQueue* will contain a pointer to each telemetry packet instance maintained by the allocator. Each instance can support a telemetry packet up to the size specified as part to the allocator's constructor.

### 15.5.2 Use 2:: Manage ordered transfer of telemetry data to hardware

From the client's point of view, all telemetry packet acquisition, formatting and posting is accomplished using subclasses of **TlmForm**. **TlmForm** maintains a pointer to the telemetry packet currently being formatted. Upon construction, this pointer is 0, and the client must invoke **TlmForm**::waitForBuffer() to obtain a telemetry packet. In order to give client code a certain degree of flexibility, this is NOT done automatically by **TlmForm**'s constructor. Once the packet is posted to the telemetry manager, TlmForm zeros this pointer to prevent further modifications of the packet's contents, and the client code must call TlmForm::waitForBuffer() to obtain another packet. If a **TlmForm** instance is destroyed prior to posting an allocated telemetry packet, its destructor invokes the packet's **TlmPkt**::release() function to ensure that it is not lost. Figure 58 illustrates the overall life-cycle of a **TlmForm** instance, and its formatted telemetry packet.

FIGURE 58. Telemetry formatting, buffering and transfer



1. A client decides to create and send a telemetry packet. It declares *tlmForm*, whose constructor initializes the state of the instance, and zeros its packet instance pointer.
2. The client then waits for a telemetry packet buffer to become available using *tlmForm*.`waitForBuffer`().
3. *tlmForm*.`waitForBuffer`() in-turn calls its allocator's member function, *tlmAllocator*.`waitForPkt`() to attempt to reserve an unused **`TlmPkt`** instance.
4. *tlmAllocator*.`waitForPkt`() then invokes *freeQueue*.`waitForPkt`().
5. And finally, *freeQueue*.`waitForPkt`() invokes its protected member function, **Queue**::`waitForItem`() to block until a telemetry packet pointer becomes available. Once the allocator returns, *tlmForm* retains the acquired packet pointer until it is either posted, (see step 11), or until *tlmForm* is destroyed. If *tlmForm* is destroyed prior to the packet being posted, *tlmForm*'s destructor releases the packet back to its allocator (not shown). If the packet is posted to the *tlmManager*, *tlmForm* is no longer responsible for the packet, and it is *tlmManager*'s responsibility to ensure that the packet is released.
6. Once a packet has been obtained by *tlmForm*, it sets the packet's format tag using *tlmPkt*.`setFormatTag`().

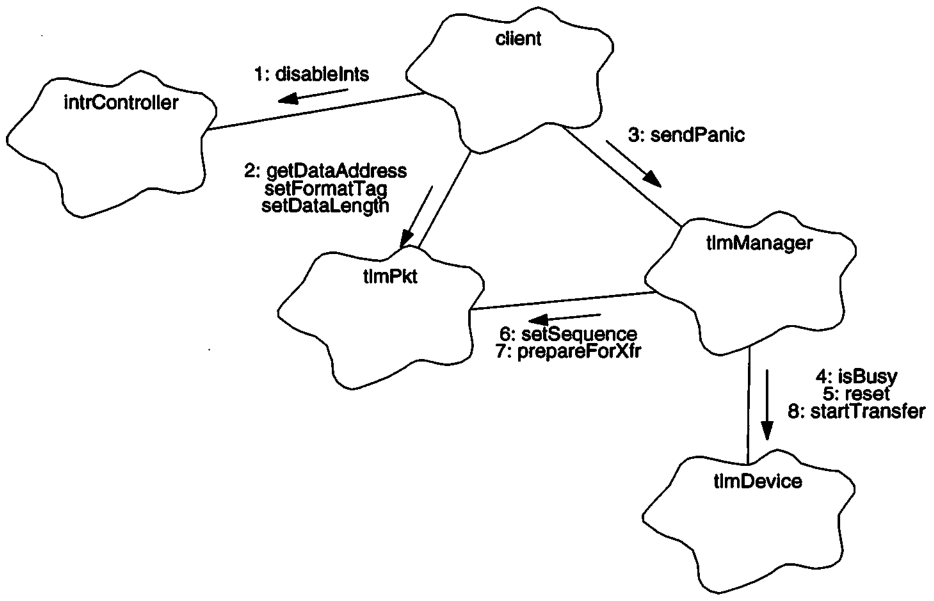
7. The client then uses various member functions of the type of *tlmForm* they're using to obtain the address and length of the packet's available user data area. The subclass implementations use getBufferInfo() to determine where their sub-headers belong, and verify the available space. Each subclass which provides nested header information is responsible for overloading getBufferInfo() and setLength() functions to allow its subclasses to determine where their respective data belong.
8. **TlmForm**::getBufferInfo() then uses *tlmPkt*.getDataAddress() and *tlmPkt*.getMaxDataLength() to get the telemetry packet's user data area and length. NOTE: Subclasses of **TlmForm** should use their respective parent's getBufferInfo() function to get the corresponding information.
9. If the packet supports a variable amount of user data, the client must specify the amount of data written, using *tlmForm*.setLength(). **TlmForm**::setLength() then uses *tlmPkt*.setDataLength() (not shown) to tell the packet the amount of data written (including the sizes of all sub-headers). NOTE: Subclasses of **TlmForm** should invoke their parent's setLength() member function, adding their sub-header size to the passed data size, rather than using *tlmPkt*.setDataLength().
10. Once the client has completed the packet, it tells the *tlmForm* to transfer the packet out of the instrument using *tlmForm*.post().
11. *tlmForm*.post() then invokes *tlmManager*.post(), passing the address of *tlmPkt*.
12. *tlmManager*.post() sets the packet's sequence number using *tlmPkt*.setSequence() (not shown) and then places the packet address on the end of its queue, using *sendQueue*.enqueuePkt(), which then uses **Queue**::enqueue() (not shown).
13. Once *tlmManager*.post() returns, *tlmForm* zeros its local pointer to the packet. This ends *tlmForm*'s responsibility concerning *tlmPkt*, and the client can destroy (~TlmForm) *tlmForm*, without affecting the posted *tlmPkt*.
14. Once the telemetry device is ready, *tlmManager* gets the next telemetry packet to send from the queue using *sendQueue*.requestPkt().
15. *tlmManager* then copies packet header information and obtains the packet's buffer address and length using *tlmPkt*.prepareForXfr().
16. *tlmManager* then tells the telemetry device to transfer the packet, using *tlmDevice*.startTransfer().
17. Once the transfer completes, *tlmDevice*'s interrupt handler invokes the installed telemetry callback, *tlmCallback*.invoke().
18. *tlmCallback*.invoke() then invokes the *tlmManager*.serviceDevice() to release the packet's buffer and start the next transfer (if a packet is available).
19. *tlmManager*.serviceDevice() function then releases the completed packet using *tlmPkt*.release().
20. *tlmPkt*.release() in-turn calls its owner's **TlmAllocator**::releasePkt()
21. *tlmAllocator*.releasePkt() then invokes *freeQueue*.enqueuePkt()

22.Finally, *freeQueue*.enqueuePkt() uses the protected function, **Queue**::enqueue() to place the available packet onto the end of the queue.

### 15.5.3 Use 3:: Transmission of fatal error messages

The **TlmManager** class provides the ability to jam a fatal error message into the telemetry stream. It assumes that, when doing this, all interrupts have been disabled by the client. Figure 59 illustrates the overall fatal error message scenario.

FIGURE 59. Sending a Fatal Error Message



1. The client first ensures that interrupts are disabled using *intrController*.disableInts(), or by declaring an **IntrGuard** instance (not shown).
2. The client then uses the various member functions of *tlmPkt* to form a fatal error message telemetry packet.
3. The client then sends the message using *tlmManager*.sendPanic()
4. *tlmManager*.sendPanic() firsts polls the telemetry device until the current packet being sent completes, using *tlmDevice*.isBusy(), or until its polling loop counter reaches its limit.
5. *tlmManager*.sendPanic() then resets the telemetry hardware using *tlmDevice*.reset().
6. *tlmManager*.sendPanic() then sets the sequence number for the fatal message packet using *tlmPkt*.setSequence().

7. *tlmManager*.`sendPanic()` then copies the packet's header information into the packet's transfer buffer, and obtains the address and length of the transfer using *tlmPkt*.`prepareForXfr()`.
8. *tlmManager*.`sendPanic()` then transfers the packet out of the instrument using *tlmDevice*.`startTransfer()`. *tlmManager*.`sendPanic()` polls the telemetry device until the fatal message packet is sent, using *tlmDevice*.`isBusy()`, or until its polling loop counter reaches its limit (not shown).

## 15.6 Class TlmManager

Documentation:

This class manages a queue of telemetry packets, and the transmission of these packets via the instrument telemetry hardware.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Public Uses:

**TlmPkt**

Implementation Uses:

**TlmDevice** *tlmDevice*

Public Interface:

Operations:

```
TlmManager()
post()
sendPanic()
serviceDevice()
```

Private Interface:

Has-A Relationships:

**TlmQueue** *sendQueue*: This is a queue of pointers to telemetry packets to be sent out of the instrument.

**TlmPkt*** *curPkt*: This is a pointer to the telemetry packet currently being transferred out of the instrument. If no transfer is underway, this pointer will be 0.

**const unsigned** *panicTimeout*: This value indicates loop iterations to wait for the telemetry system to go idle, prior to, and after sending a panic message. This value must result in a delay greater than 64 seconds (4K bytes/pkt @ 512 bits/second)

**unsigned** *curSequence*: This variable contains the sequence number of the next packet to send out of the instrument.

Concurrency: Synchronous

### 15.6.1 TlmManager()

Public member of: **`TlmManager`**

Arguments:

**`unsigned`** *`queueId`*

Documentation:

This constructor initializes the telemetry queue, using the passed ***`Nucleus RTX`*** *`queueId`*. It also zeros the current transfer packet pointer, indicating that no transfers are underway.

Semantics:

Set *`panicTimeout`* to its initial value, zero *`curPkt`* and *`curSequence`*, and construct *`sendQueue`* using the passed *`queueId`* and with the maximum supported number of packets for the entire system (final number is TBD). Then tell the *`tlmDevice`* to install the address of the telemetry manager's interrupt callback instance, *`tlmCallback`*.

Concurrency: Sequential

### 15.6.2 post()

Public member of: **TlmManager**

Return Class: **void**

Arguments:

**TlmPkt*** *pkt*

Documentation:

This function places the referenced telemetry packet, *pkt*, onto the manager's telemetry queue. Once all previously posted packets have been transferred, the referenced packet will be transferred out of the instrument. Once its transfer is complete, its `release()` member function will be invoked.

Semantics:

Set the packet sequence number to *curSequence* using *pkt*->`setSequence()`. Invoke *sendQueue*.`enqueuePkt()` to place the packet on the send queue. If *curPkt* is 0, no transfer is underway, then invoke `serviceDevice()` to start a fresh transfer. If *curPkt* is not 0, then a packet is transfer in progress. Once the transfer completes, the interrupt callback will start the subsequent transfer.

Concurrency: Synchronous

### 15.6.3 sendPanic()

Public member of: **TlmManager**

Return Class: **void**

Arguments:

**TlmPkt*** *pkt*

Documentation:

This function “jams” a telemetry packet into the telemetry stream.

Preconditions:

Interrupts must be disabled prior to calling this function

Semantics:

Loop until *tlmDevice*.isBusy() returns *BoolFalse*, or until *panicTimeout* is reached. Then reset the telemetry hardware using *tlmDevice*.reset(). Prepare the packet for transfer and obtain its buffer address and length using *pkt*->prepareForXfr(). Then tell the telemetry hardware to transfer the packet, *tlmDevice*.startTransfer(). Once the transfer has started, loop until *tlmDevice*.isBusy() returns *BoolFalse*, or until *panicTimeout* is reached.

Concurrency: Sequential

### 15.6.4 serviceDevice()

|  |  |
|---|---|
| Public member of: | **TlmManager** |
| Return Class: | **void** |
| Arguments: | **IntrDevice*** *devptr* |

Documentation:

This function is invoked by the telemetry callback instance once a telemetry transfer has been completed. This function invokes the just completed packet's `release()` member function. It then attempts to dequeue another packet from the *sendQueue* and start its transfer.

Semantics:

If *curPkt* is not 0, then a transfer has just completed. Invoke *curPkt*->`release()` to release the packet. Invoke *sendQueue*.`requestPkt()` to dequeue a telemetry packet and store the result in *curPkt*. If the result is not zero, then store and increment *curSequence* into the packet (*curPkt*->`setSequence()`), and prepare the packet and obtain its transfer buffer and length using *curPkt*->`prepareForXfr()`. Then use *tlmDevice*.`startTransfer()` to tell the hardware to transfer the packet.

Concurrency: Synchronous

## 15.7 Class TlmQueue

Documentation:

This class represents a fixed length queue of telemetry packet pointers.

| | |
|---|---|
| Export Control: | Public |
| Cardinality: | n |
| Hierarchy: | |
| Superclasses: | **Queue** |
| Public Uses: | **TlmPkt** |
| Public Interface: | |
| Operations: | `TlmQueue()`<br>`enqueuePkt()`<br>`requestPkt()`<br>`waitForPkt()` |
| Concurrency: | Synchronous |
| Persistence: | Persistent |

## 15.7.1 TlmQueue()

Public member of: **TlmQueue**

Arguments:

**unsigned** *queueId*
**unsigned** *nitems*

Documentation:

This constructor initializes the queue of telemetry packet pointers. *queueId* is the ***Nucleus RTX*** queue identifier used for the particular instance being initialized. *nitems* is the number of packet pointers which can be contained in this queue (used for sanity checking against the ***RTX*** queue instance). This function invokes the parent constructor, **Queue**::Queue(), using the number of words in a telemetry packet pointer as the element size within the queue (*queueId* and *nitems* are passed unmodified).

Concurrency: Sequential

## 15.7.2 enqueuePkt()

Public member of: **TlmQueue**

Return Class: **void**

Arguments:

**TlmPkt*** *pkt*

Documentation:

This function enqueues a pointer, *pkt*, to a telemetry packet onto this queue, using its parent's **Queue**::enqueue() function.

Concurrency: Synchronous

### 15.7.3 requestPkt()

| | |
|---|---|
| Public member of: | **`TlmQueue`** |
| Return Class: | **`TlmPkt*`** |

Documentation:

This function attempts to dequeue a packet pointer, using its parent's **`Queue`**::`dequeue`() function. If no packets are available, this function returns 0.

| | |
|---|---|
| Concurrency: | Synchronous |

### 15.7.4 waitForPkt()

| | |
|---|---|
| Public member of: | **`TlmQueue`** |
| Return Class: | **`TlmPkt*`** |
| Arguments: | **`unsigned`** *`timeout`* |

Documentation:

This function waits for and dequeues a packet from the queue, using its parent's **`Queue`**::`waitForItem`() function. If no packets are ready, this function waits no longer than *`timeout`* timer ticks (1/10 second) for one to be enqueued. If *`timeout`* expires, this function returns 0.

| | |
|---|---|
| Concurrency: | Synchronous |

## 15.8 Class TlmPkt

Documentation:

This class represents a telemetry packet to be transferred out of the instrument.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:
`TlmPkt()`
`prepareForXfr()`
`release()`
`setFormatTag()`
`setSequence()`

Protected Interface:

Operations:
`getDataAddress()`
`getMaxDataLength()`
`setDataLength()`

Has-A Relationships:

**unsigned** *pktLength*: This represents the total number of 32-bit words to transfer as part of this telemetry packet. This field is maintained with the packet instance, rather than in the bulk telemetry buffer, due to the single-event upset vulnerability of the main bulk memory. `prepareForXfr()` copies this field to the main buffer just prior to transfer out of the instrument.

**unsigned** *pktSeq*: This field represents the packet sequence number. It is maintained with the packet instance, rather than with the main bulk memory buffer due to single-event upset considerations. It is copied to the bulk buffer by `prepareForXfr()` just prior to transfer out of the instrument.

**TlmFormatTag** *pktFormat*: This is a copy of the packet format tag. It is stored with the packet instance structure separate from the main telemetry buffer due to the vulnerability of the main buffer to single-event upsets. It is copied to the main buffer when `prepareForXfr()` is invoked, just prior to transfer out of the instrument.

Private Interface:

Has-A Relationships:

**`unsigned* const`** *`bufAddr`*: This is the address in bulk memory allocated for this telemetry packet.

**`const unsigned`** *`bufLength`*: This is the total number of 32-bit words pointed to by *`bufAddr`* for this telemetry packet.

**`const TlmAllocator*`** *`owner`*: This is a pointer to the **`TlmAllocator`** instance which allocated the packet. This pointer is used to release the packet once it is no longer needed.

| | |
|---|---|
| Concurrency: | Synchronous |
| Persistence: | Transient |

### 15.8.1 TlmPkt()

Public member of: **TlmPkt**

Arguments:

**TlmAllocator*** *allocator*
**unsigned*** *buffer*
**unsigned** *wordcnt*

Documentation:

This constructor sets up the non-volatile state of a telemetry packet instance, assigning its read-only *owner*, *bufAddr*, and *bufLength* values to the corresponding passed arguments, and zeroing *pktLength*, *pktSeq* and *pktFormat*. *owner* and *buffer* must not be 0, and *wordcnt* must be greater than or equal to 2.

Concurrency: Sequential

### 15.8.2 getDataAddress()

Protected member of: **TlmPkt**

Return Class: **unsigned ***

Documentation:

This function returns a pointer to the data area of the telemetry packet. Clients should use this method to determine where to place their respective sub-headers and data.

Concurrency: Sequential

## 15.8.3 getMaxDataLength()

Protected member of: **TlmPkt**

Return Class: **unsigned**

Documentation:

This function returns the number of 32-bit words contained in the data buffer. The address of the data buffer is provided by `getDataAddress()`.

Concurrency: Sequential

## 15.8.4 setDataLength()

Protected member of: **TlmPkt**

Return Class: **void**

Arguments:

**unsigned** *datacnt*

Documentation:

This function is used to inform the packet of the amount of information written to its data area. A pointer to the data area is provided by `getDataAddress()`. The argument *datacnt* reflects the number of 32-bit words written into this data area. *datacnt* must not exceed the value returned by `getMaxDataLength()`. This function stores the passed *datacnt* plus the number of words in the telemetry packet header into *pktLength*.

Concurrency: Sequential

### 15.8.5 setFormatTag()

Public member of: **TlmPkt**

Return Class: **void**

Arguments:

**TlmFormatTag** *tag*

Documentation:

This function sets the format tag of the telemetry packet instance, by copying *tag* into *pktFormat*.

Concurrency: Sequential

### 15.8.6 prepareForXfr()

Public member of: **TlmPkt**

Return Class: **void**

Arguments:

**unsigned*&** *xfraddr*

**unsigned&** *xfrlen*

Documentation:

This function prepares a telemetry buffer to be transferred to the telemetry interface hardware. Upon return, *xfraddr* will contain the address of the buffer to transfer, and *xfrlen* will contain the number of 32-bit words to transfer. NOTE: Given the volatile nature of the telemetry transfer buffer, this function should be called just prior to instructing the telemetry hardware to transfer the buffer.

Semantics:

Copy *pktLength*, *pktFormat* and *pktSeq* into the SEU-soft header space of the buffer, and fill-in the *xfraddr* and *xfrlen* output arguments.

Concurrency: Synchronous

### 15.8.7 release()

Public member of: **TlmPkt**

Return Class: **void**

Documentation:

This function releases a telemetry packet for re-use by invoking *owner*->releasePkt().

Concurrency: Synchronous

### 15.8.8 setSequence()

Public member of: **TlmPkt**

Return Class: **void**

Arguments:

**unsigned** *sequence*

Documentation:

This function sets the sequence number within the packet by copying *sequence* to *pktSeq*.

Concurrency: Sequential

## 15.9 Class TlmAllocator

Documentation:

This class is responsible for managing the run-time allocation and releasing of telemetry packets.

| | |
|---|---|
| Export Control: | Public |
| Cardinality: | n |

Hierarchy:

Superclasses: **none**

Public Uses:

**TlmPkt**

Public Interface:

Operations:
```
TlmAllocator()
releasePkt()
requestPkt()
waitForPkt()
```

Private Interface:

Has-A Relationships:

**TlmQueue** *freeQueue*: This queue stores a list of available telemetry packets.

**TlmPool** *instancePool*: This buffer pool is used gain access to memory reserved for telemetry packet instances. This pool is only used during initialization.

**static unsigned* const** *rawBase*: This is a shared constant which points to the start of the raw telemetry buffer space in uncached memory.

**static const unsigned** *rawWordCnt*: This constant shared variable indicates the total number of 32-bit words contained in the raw telemetry buffer space.

**static unsigned*** *rawPtr*: This is a shared pointer to the next available section of the raw telemetry buffer space, and is used by each **TlmAllocator** instance during setup to reserve a section of the buffer.

### 15.9.1 TlmAllocator()

Public member of: **`TlmAllocator`**

Arguments:

**`unsigned`** *`poolId`*
**`unsigned`** *`queueId`*
**`unsigned`** *`wordcnt`*
**`unsigned`** *`npkts`*

Documentation:

This constructor initializes its memory pool, `instancePool`, and free packet list, `freeQueue`, using *`poolId`* and *`queueId`*, and allocates sections of the raw telemetry buffer, placing references to the allocated packet instances into its `freeQueue`.

Preconditions:

The caller must ensure that no other **`TlmAllocator`** constructor preempts this call. There must be at least (*`npkts`***`wordcnt`*) 32-bit words remaining in the raw telemetry buffer space (i.e. *`rawPtr`* + (*`npkts`***`wordcnt`*) <= *`rawBase`* + *`rawWordCnt`*).

Semantics:

Initialize `freeQueue` and `instancePool`, passing the number of words in a **`TlmPkt`** instance as the size of each element in the `instancePool`. For each packet (i.e. iterate *`npkts`* times), allocate a **`TlmPkt`** instance from the `instancePool`, invoke its constructor, and make it available using `releasePkt`(). When constructing the packet, pass the current value of *`rawPtr`* as the raw telemetry buffer pointer, and advance *`rawPtr`* by *`wordcnt`*.

Postconditions:

The `instancePool` will be empty, *`rawPtr`* will have been advanced by *`npkts`** *`wordcnt`* words, and `freeQueue` will contain pointers to initialized instances every available telemetry packet associated with this **`TlmAllocator`** instance.

Concurrency: Sequential

### 15.9.2 releasePkt()

Public member of: **TlmAllocator**

Return Class: **void**

Arguments:

**TlmPkt*** *pkt*

Documentation:

This function releases a telemetry packet back into the *freeQueue*, using *freeQueue*.enqueuePkt().

Concurrency: Synchronous

### 15.9.3 requestPkt()

Public member of: **TlmAllocator**

Return Class: **TlmPkt***

Documentation:

This function attempts to allocate a telemetry packet. If successful, it returns a pointer to the obtained packet. If none are available, it returns 0. This function uses *freeQueue*.requestPkt() to allocate the next available telemetry packet.

Concurrency: Synchronous

### 15.9.4 waitForPkt()

Public member of: **TlmAllocator**

Return Class: **TlmPkt***

Arguments:

**unsigned** *timeout*

Documentation:

This function attempts to dequeue a telemetry packet from its *freeQueue*. If *timeout* expires before a packet becomes available, the function returns 0. If a packet is obtained, it returns a pointer to the constructed packet. This function uses *freeQueue*.waitForPkt() to wait for and dequeue the packet pointer.

Concurrency: Synchronous

## 15.10 Class TlmPool

Documentation:

This class represents a pool of buffers used to contain instances of telemetry packets. The intended use of this class is to provide start-up allocation of a patchable number of telemetry packets. As such, this class provides no mechanism for releasing a buffer back into its Memory Pool.

| | | |
|---|---|---|
| Export Control: | | Public |
| Cardinality: | | n |
| Hierarchy: | | |
| | Superclasses: | **`MemoryPool`** |
| Public Uses: | | |
| | | **`TlmPkt`** |
| Public Interface: | | |
| | Operations: | `TlmPool()`<br>`allocatePkt()` |
| Concurrency: | | Sequential |
| Persistence: | | Persistent |

### 15.10.1 TlmPool()

| | |
|---|---|
| Public member of: | **TlmPool** |

Arguments:

**unsigned** *poolId*
**unsigned** *instanceSize*
**unsigned** *nbufs*

Documentation:

This constructor initializes a pool of buffers to be used to reserve space for telemetry packet instances. *poolId* is the ***Nucleus RTX*** partition identifier, *instanceSize* and *nbufs* are used to sanity check the **RTX** pool capabilities and are respectively the size of the buffered class instance and number of instances.

| | |
|---|---|
| Concurrency: | Sequential |

### 15.10.2 allocatePkt()

| | |
|---|---|
| Public member of: | **TlmPool** |
| Return Class: | **TlmPkt*** |

Documentation:

This function retrieves a telemetry packet buffer from the pool. This function uses its parent's **MemoryPool**::`allocate`() function to acquire a pointer to space reserved for a packet. NOTE: The caller is responsible for invoking the constructor on the returned packet pointer.

Preconditions:

This function must be called no more than once for each reserved telemetry packet instance.

| | |
|---|---|
| Concurrency: | Synchronous |

## 15.11 Class TlmForm

Documentation:

This base class represents a telemetry packet formatter. It is responsible for acquiring a telemetry packet buffer, formatting the contents of the buffer, and for posting the buffer to the telemetry manager. Different types of telemetry packets use different subclasses of this base class.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

**TlmManager** *tlmManager*

Public Interface:

Operations:

```
TlmForm()
~TlmForm()
post()
waitForBuffer()
```

Protected Interface:

Operations:

```
getBufferInfo()
setLength()
```

Has-A Relationships:

**const TlmFormatTag** *formatTag*: This is the telemetry packet format tag to use for this particular type of formatter. It's value is used to set the packet's tag value when the packet is allocated.

## Private Interface:

### Has-A Relationships:

**`const TlmAllocator*`** *`allocator`*: This is a constant pointer to the TlmAllocator instance to use for all telemetry packets corresponding to this formatter instance.

**`TlmPkt*`** *`pktPtr`*: This is a pointer to the current telemetry packet instance being formatted by this formatter.

### 15.11.1 TlmForm()

Public member of: **`TlmForm`**

Arguments:

**`TlmAllocator*`** *`pktsrc`*
**`TlmFormatTag`** *`tag`*

Documentation:

This constructor initializes the top-level state of the formatter, by setting the read-only allocator instance from *`pktsrc`*, and by zeroing the current packet pointer. *`tag`* is the telemetry packet format tag to use for packets written by this formatter instance.

Postconditions:

The only valid functions that may be called immediately after construction are `waitForBuffer`() and the destructor.

Concurrency: Sequential

### 15.11.2 ~TlmForm()

Public member of: **`TlmForm`**

Documentation:

This destructor tests for and releases a telemetry packet buffer, if one has been obtained, but not yet posted.

Concurrency: Sequential

### 15.11.3 getBufferInfo()

Protected member of: **`TlmForm`**

Return Class: **`void`**

Arguments:

**`unsigned*&`** *`bufaddr`*
**`unsigned&`** *`bufcnt`*

Documentation:

This function supplies the caller with the user portion of the telemetry packet buffer, and the number of words available in the buffer. Upon return, bufaddr will contain a pointer to the user data portion of the packet, and bufcnt will contain the maximum number of 32-bit words which can be written by the client.

Preconditions:

`waitForBuffer()` must have succeeded after construction, or after a `post()` call.

Concurrency: Synchronous

### 15.11.4 post()

<u>Public member of:</u> **`TlmForm`**

<u>Return Class:</u> **`void`**

<u>Documentation:</u>

This function posts the obtained telemetry packet buffer to the telemetry manager for transfer out of the instrument.

<u>Preconditions:</u>

`waitForBuffer()` must be called prior to each call to this function.

<u>Postconditions:</u>

The current telemetry packet is disassociated from the formatter, preventing modifications to the packet after being posted to the telemetry manager. Another call to `waitForBuffer()` must be made prior to using any of the set or get calls.

<u>Concurrency:</u> Sequential

### 15.11.5 requestBuffer()

<u>Public member of:</u> **`TlmForm`**

<u>Return Class:</u> **`Boolean`**

<u>Documentation:</u>

This function attempts to obtain a telemetry packet from the packet allocator. If successful, this function returns *`BoolTrue`*. If no packets are available at the time of the call, it returns *`BoolFalse`*.

<u>Concurrency:</u> Sequential

### 15.11.6 setLength()

Protected member of: **`TlmForm`**

Return Class: **`void`**

Arguments:

**`unsigned`** *`wordcnt`*

Documentation:

This function tells the formatter to set the packet length (plus the header size) of the telemetry packet.

Preconditions:

`waitForBuffer()` must be called after construction, or after a `post()` call.

Concurrency: Sequential

### 15.11.7 waitForBuffer()

| | |
|---|---|
| Public member of: | **TlmForm** |
| Return Class: | **Boolean** |
| Arguments: | **unsigned** *timeout* |

Documentation:

This function uses its allocator to wait for and allocate a telemetry packet. If *timeout* is reached, this function returns *BoolFalse*. If a packet was obtained, it returns *BoolTrue*.

Preconditions:

A packet must not have already been allocated.

Postconditions:

The formatter is ready to accept functions which query for buffer information, or set fields within the packet.

Concurrency: Synchronous

## 15.12 Class TlmCallback

Documentation:

This class handles interrupt callbacks from the telemetry device.

| | |
|---|---|
| Export Control: | Public |
| Cardinality: | 1 |
| Hierarchy: | |
| Superclasses: | **`DevCallback`** |
| Implementation Uses: | |
| | **`TlmManager`** *`tlmManager`* |
| Public Interface: | |
| Operations: | `invoke()` |
| Concurrency: | Synchronous |
| Persistence: | Persistent |

### 15.12.1 invoke()

| | |
|---|---|
| Public member of: | **`TlmCallback`** |
| Return Class: | **`void`** |
| Arguments: | |
| | **`IntrDevice*`** *`devptr`* |

Documentation:

This function is called by the telemetry device during its interrupt processing. This function invokes the *`tlmManager`*`.serviceDevice()` function to handle the end of a telemetry packet transfer.

Concurrency: Synchronous