CSR

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CENTER FOR SPACE RESEARCH
CAMBRIDGE, MASSACHUSETTS 02139

REVISION LOG

TITLE: Software Detailed Design
Telemetry Device

DOC. NO. 36-53209

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
—	1/25/95	—	—	Initial version for walkthrough	
A	5/3/95	36-230	all	Incorporate review comments	5/16/95

9.0 Telemetry Device (36-53209 A)

9.1 Purpose

The purpose of the Telemetry Device is to provide access to Back End's Telemetry Interface logic and to the Downlink Transfer Controller.

9.2 Uses

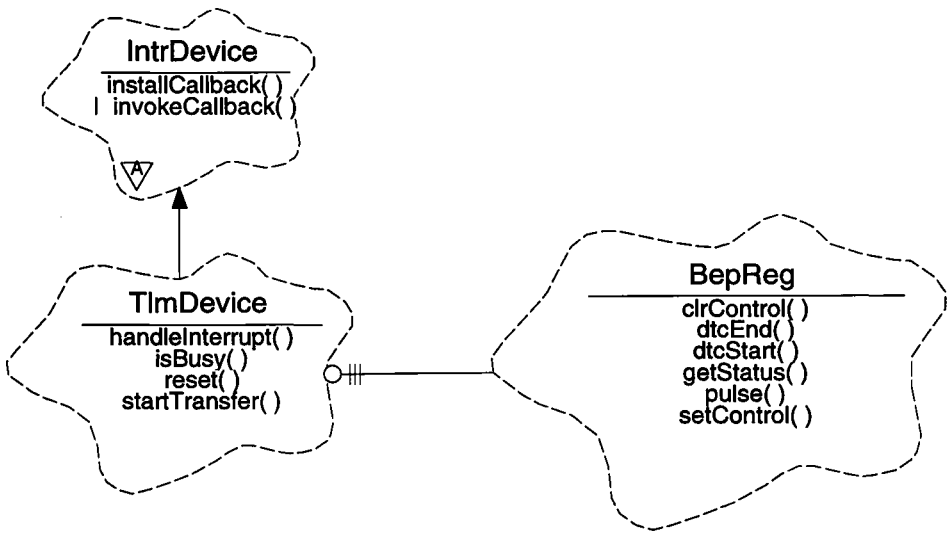
The Telemetry Device class, **`TlmDevice`**, provides the following features:

Use 1:: Initiate a transfer of data from uncached memory to the RCTU hardware
Use 2:: Determine if a telemetry transfer is in progress
Use 3:: Reset the downlink logic
Use 4:: Handle downlink interrupts

9.3 Organization

The Telemetry Device, **`TlmDevice`**, is an interruptible device, and is therefore a subclass of **`IntrDevice`** (see Section 6.0). This class relies on the **`BepReg`** class to provide access to the downlink hardware control logic and Downlink Transfer Controller (DTC).

FIGURE 20. Telemetry Device Class Relationships



`TlmDevice` - This class represents the Back End's downlink logic and controller hardware. It provides functions which reset the controller, start a downlink transfer, determine if a transfer is in-progress, and handle interrupts. In addition to theses functions, it inherits the ability to install and invoke interrupt callback instances from **`IntrDevice`**.

IntrDevice - This is an abstract class which defines the common interface to all types of interruptible devices. It is used by the Interrupt Controller (not shown) to dispatch control to interruptible devices, and by client code to install callback functions. Child classes of **IntrDevice**, including **TlmDevice**, may use their parent's protected method, `invokeCallback()` to invoke the installed callback instance (see Section 6.0).

BepReg- This class represents the lowest level hardware access to the features provided by the Back End hardware control, status, and pulse registers, and to the Downlink Transfer Controller.

9.4 Downlink Transfer Controller Description

The Back End Processor's Downlink Transfer Controller (DTC) is a type of Direct-Memory-Access (DMA) device. It transfers buffers of data from the BEP's uncached memory to the Remote Command and Telemetry Unit (RCTU) telemetry serial port interface logic. This logic handles the insertion of the ACIS time-stamp into the first 32-bits of science data of each Science Telemetry Frame. This logic also provides two deep, 32-bit word FIFO between the DTC and the RCTU. During a transfer, the DTC enqueues 32-bit words from uncached memory into the interface logic's buffer. Once the last word of the block has been transferred from uncached memory, the logic generates a Downlink Interrupt. Assuming that peak transfer rate out of the FIFO is 128Kbps, the 2-deep FIFO gives the software about 0.5 milliseconds to handle the interrupt and start a new transfer before a gap is introduced between the transfers. If a gap is introduced, the hardware will respond to RCTU requests with an 8-bit pattern whose value is 0xb7 in hexadecimal (see Section TBD).

9.5 Scenarios

9.5.1 Use 1: Initiate Downlink Transfers from Uncached Memory

To start a telemetry transfer, a client invokes **TlmDevice**::startTransfer(), passing the start address and number of 32-bit words in the transfer. **TlmDevice**::startTransfer() then uses **BepReg**::clrControl() to disable the DTC. It then uses **BepReg**::dtcStart() and **BepReg**::dtcEnd() to obtain the address of the start and end registers. It then programs the registers for the transfer, and enables the controller using **BepReg**::setControl(). Once enabled, the hardware transfers the contents of the buffer to the RCTU interface logic.

It is the callers responsibility to ensure that a transfer is not already in-progress before calling this function.

9.5.2 Use 2: Determine if a transfer is in-progress

A client determines if a transfer is in-progress by calling **TlmDevice**::isBusy(). Currently, there is no way of determining if a transfer is busy by examining the hardware. In order to support this function, the **TlmDevice** class contains a state variable which is asserted whenever a transfer is started, and which is cleared whenever a transfer interrupt is handled. **TlmDevice**::isBusy() first checks the state variable. If the state variable is de-asserted, then no transfers have been requested, or the previous transfer has completed and its interrupt has been handled. If the state variable is asserted, then either a transfer is underway, or a transfer has completed, but the corresponding interrupt has not yet been handled. In this case, the function uses **BepReg**::getStatus() to examine the Downlink Interrupt bit. If the bit is de-asserted, then a transfer is still in-progress. If the bit is asserted, then a pending downlink interrupt has not yet been handled, but the previous transfer has completed.

9.5.3 Use 3: Reset the downlink logic

In order to support initialization, and best-effort attempts to send fatal error messages, the **`TlmDevice`** class provides a **`TlmDevice`**::`reset`(). This function ensures that the DTC is disabled, that any pending interrupts are cleared, and places the start and end registers into a known state. Any transfers in-progress when this function is called will be aborted, even though a portion of the previous transfer may have already been sent.

9.5.4 Use 4: Handle downlink interrupts

Whenever all of a request's data have been transferred from uncached memory to the RCTU interface logic, the hardware will generate a Downlink Interrupt. This will eventually cause **`TlmDevice`**::`handleInterrupt`() to be called. This function then clears the transfer busy state variable, resets the interrupt cause, and uses **`IntrDevice`**::`invokeCallback`() to call the telemetry device's callback instance, if one is installed.

9.6 Class TlmDevice

Documentation:

This class provides access to the Back End Processor's telemetry hardware.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **IntrDevice**

Implementation Uses:

BepReg

Public Interface:

Operations:
```
handleInterrupt()
isBusy()
reset()
startTransfer()
```

Protected Interface:

Has-A Relationships:

Boolean *busyFlag*: This flag indicates whether a downlink transfer is underway. If `BoolTrue`, a transfer has been started. If `BoolFalse`, no transfers are underway.

Concurrency: Synchronous

9.6.1 handleInterrupt()

Public member of: **TlmDevice**

Return Class: **void**

Documentation:

This function handles the Downlink Controller Interrupt. This function overloads the **IntrDevice** function of the same name.

Semantics:

Clear the internal *busyFlag*, and then clear the downlink interrupt using **BepReg**::pulse(). Then invoke **IntrDevice**::invokeCallback() to pass control to the client code.

Concurrency: Synchronous

9.6.2 isBusy()

Public member of: **TlmDevice**

Return Class: **Boolean**

Documentation:

This function tests to see if a telemetry transfer is underway. If so, it returns *BoolTrue*. If the telemetry hardware is idle, it returns *BoolFalse*.

Semantics:

Test the internal variable *busyFlag*. If cleared, then no transfer is underway. If the flag is set, then use **BepReg**::getStatus() to read the status register. If the downlink interrupt is asserted, then a started transfer just completed, but the interrupt handler hasn't been invoked yet (probably because we're in a higher priority interrupt). Return *BoolFalse*. If the interrupt bit is not asserted, then a transfer is underway and return *BoolTrue*.

Concurrency: Synchronous

9.6.3 reset()

Public member of: **TlmDevice**

Return Class: **void**

Documentation:

This function initializes the Downlink Controller hardware, interrupting any transfer in progress. NOTE: This function should only be used during initialization and during error processing.

Semantics:

Clear the *busyFlag* and then disable the Downlink Controller using **BepReg**::clrControl() and reset the start and end registers to equal values. Clear any pending interrupt using **BepReg**::pulse().

Postconditions:

The downlink controller will be in a idle state.

Concurrency: Sequential

9.6.4 startTransfer()

Public member of: **TlmDevice**

Return Class: **void**

Arguments:

const unsigned* *srcaddr*
unsigned *wordcnt*

Documentation:

This function starts a telemetry transfer of *wordcnt* 32-bit words from *srcaddr* to the RCTU interface hardware.

Preconditions:

A transfer must not already be in progress. *srcaddr* must be in uncached memory and *wordcnt* must be greater than 0.

Semantics:

First, set the *busyFlag* to indicate that a transfer is about to start. Then disable the controller using **BepReg**::clrControl(). Use the **BepReg**::dtcStart and **BepReg**::dtcEnd() functions to obtain the start and end register addresses. Program the start and end registers for the transfer and enable the downlink controller using **BepReg**::setControl().

Postconditions:

wordcnt words located at *srcaddr* will be transferred to the RCTU interface hardware. A telemetry interrupt will indicate when the transfer is complete.

Concurrency: Synchronous