CSR	MASSACHUSETTS INSTITUTE OF TECHNOLOGY CENTER FOR SPACE RESEARCH CAMBRIDGE, MASSACHUSETTS 02139	
REVISION LOG	**TITLE: Software Detailed Design Parameter Block Management**	**DOC. NO. 36-53239 Rev. 01**

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
01	4/9/96	36-573	all	Initial version. Incorporated comments from review.	

37.0 Bias Thief Class (36-53239 01)

37.1 Purpose

The purpose of the Bias Thief is to copy the contents the Front End Processor (FEP) pixel bias map values to telemetry during science processing. The Bias Thief copies the values directly from the FEP's bias map in BEP-FEP shared-memory without directly interacting with the software running on the FEP, hence the name "thief."

As is for other telemetry producers, its telemetry utilization is bounded by the number of telemetry packet buffers allocated to it by the system during startup. By convention, the Science Manager (see Section 33.0) is allocated the bulk of the telemetry buffers, hence the Bias Thief tends to trickle the maps to telemetry when telemetry is saturated with science data.

37.2 Uses

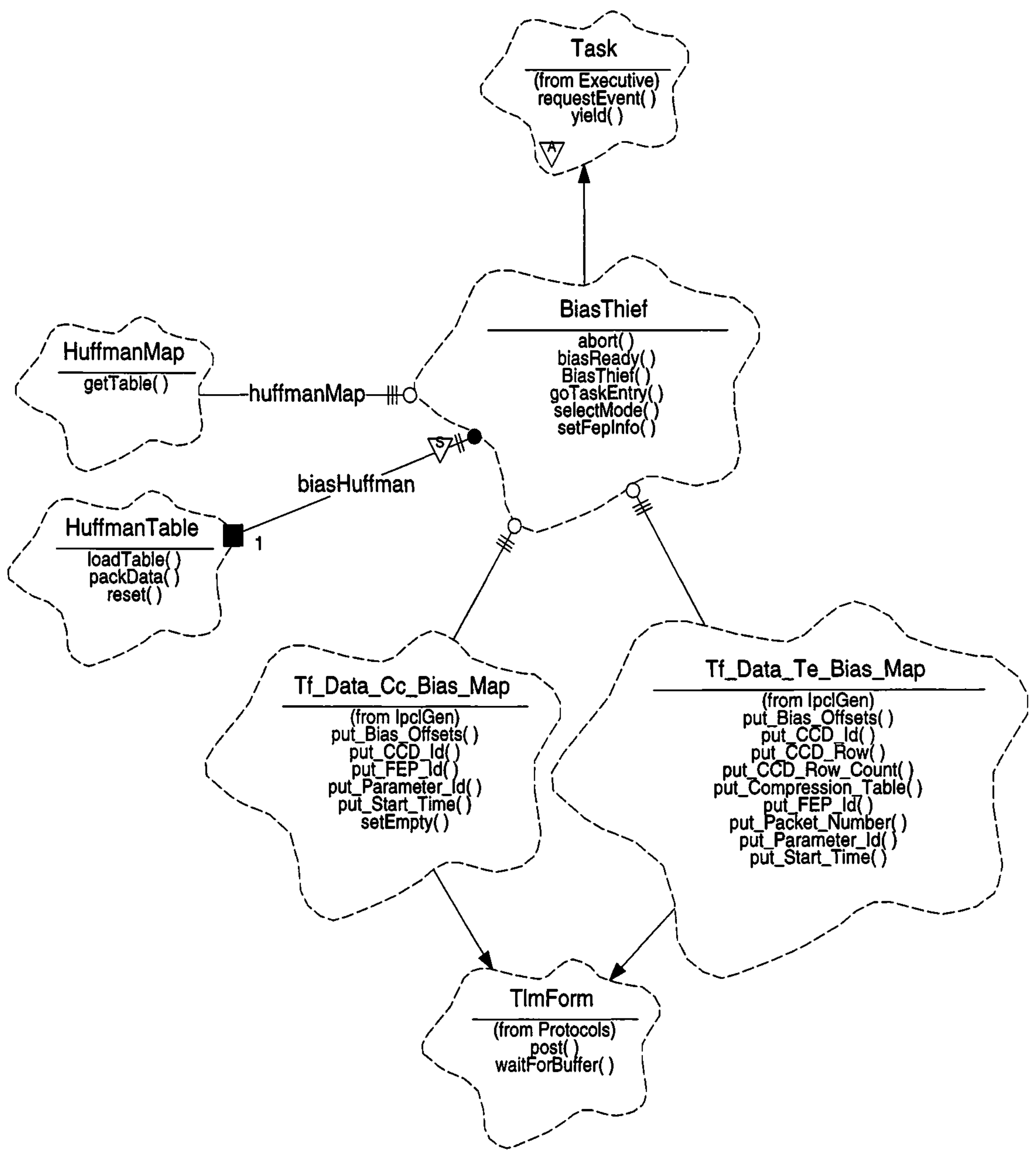
The following lists the use of the Bias Thief class:

Use 1:: Select which type of bias maps are to be sent
Use 2:: Specify the bias map parameters for each Front End Processor
Use 3:: Start transmission of the pixel bias maps
Use 4:: Abort transmission of the pixel bias maps

37.3 Organization

Figure 166 illustrates the class relationships used by the **`BiasThief`** class.

FIGURE 166. Bias Thief Class Relationships



BiasThief- This class is a subclass of ***Executive***::**Task** and is responsible for sending the pixel bias maps to telemetry while science processing is underway. This class provides a function which selects the type of pixel bias map to be sent, Timed Exposure or Continuous Clocking (`selectMode`). It provides a function to load the properties of a given FEP's bias map, such as which CCD was being processed, what the initial overclock values were, etc. (`setFepInfo`). The class provides functions to start the bias operation (`biasReady`) and to abort the transmission of the bias maps (`abort`).

Task- This class is supplied by the ***Executive*** class category. It represents and controls an active running task. The **BiasThief** class inherits from this class, and uses the class's functions to relinquish control to allow other tasks of the same priority to run (yield), and to detect queries from the **TaskMonitor** (requestEvent).

TaskMonitor (not shown)- This class is supplied by the ***Executive*** class category, and is responsible for periodically polling each task in the instrument. When polled, the **BiasThief** task responds using this classes member function (respond).

HuffmanMap - This class maintains the collection of Huffman compression tables store in I-cache. It provides functions to map an table index to the address in I-cache corresponding to the selected table (getTable).

HuffmanTable - This class is responsible for compressing data using a selectable compression table. It provides functions which load a table from I-cache (loadTable), to reset its state-machine to start compressing a set of data (reset), and to compress input data and append the data to a user-supplied output buffer (packData).

Tf_Data_Cc_Bias_Map - This class is generated by the IP&CL code-generator, and belongs to the ***IpclGen*** class category). It is a subclass of ***Protocols***::**TlmForm** and is responsible for formatting a Continuous Clocking Bias Map telemetry packet. It provides functions which write the initial overclock values for the current bias map (put_Bias_Offsets), write the CCD identifier used to produce the map (put_CCD_Id), write the identifier of the FEP which produced the map (put_FEP_Id), write the parameter block id used to compute the bias map (put_Parameter_Id), and write the ACIS science timestamp, latched at the start of the bias computation (put_Start_Time). It also provides a function which resets the contents of the bias map data (setEmpty), and provides functions which return the address and length of the bias map data buffer within the packet and set the number of 32-bit words written into the buffer (get_Data_Address, get_Data_Avail, set_Data_Written, not shown).

Tf_Data_Te_Bias_Map -This class is generated by the IP&CL code-generator, and belongs to the ***IpclGen*** class category). It is a subclass of ***Protocols***::**TlmForm** and is responsible for formatting a Timed Exposure Bias Map telemetry packet. It provides functions which write the initial overclock values for the current bias map (put_Bias_Offsets), write the CCD identifier used to produce the map (put_CCD_Id), write the identifier of the FEP which produced the map (put_FEP_Id), write the parameter block id used to compute the bias map (put_Parameter_Id), and write the ACIS science timestamp, latched at the start of the bias computation (put_Start_Time). It provides functions which write the starting CCD row identifier into the packet, sets the number of rows written into the packet buffer (put_CCD_Row, put_CCD_Row_Count), and writes the compression table identifier used to pack the data (put_Compression_Table). It also provides a function which resets the contents of the bias map data (setEmpty, not shown), sets the bias data packet number (put_Packet_Number) and provides functions which return the address and length of the bias map data buffer within the packet and set the number of 32-bit words written into the buffer (get_Data_Address, get_Data_Avail, set_Data_Written, not shown).

`TlmForm` - This class is provided by the ***`Protocols`*** class category, and is responsible for overall formatting of telemetry packet buffers. It provides functions which wait for and allocate a telemetry packet buffer (`waitForBuffer`), and which post the buffer for transfer to telemetry (`post`).

37.4 Scenarios

37.4.1 Use 1: Select which type of bias maps are to be sent

Prior to packing data, the `client` must select which type of bias maps are to be telemetered, and specify the start time of the bias computation to be telemetered, and parameter block id used to compute the maps by calling *biasThief*.`selectMode()`. `selectMode()` then records the information within the *biasThief*, and clears all FEP-specific information within the biasThief.

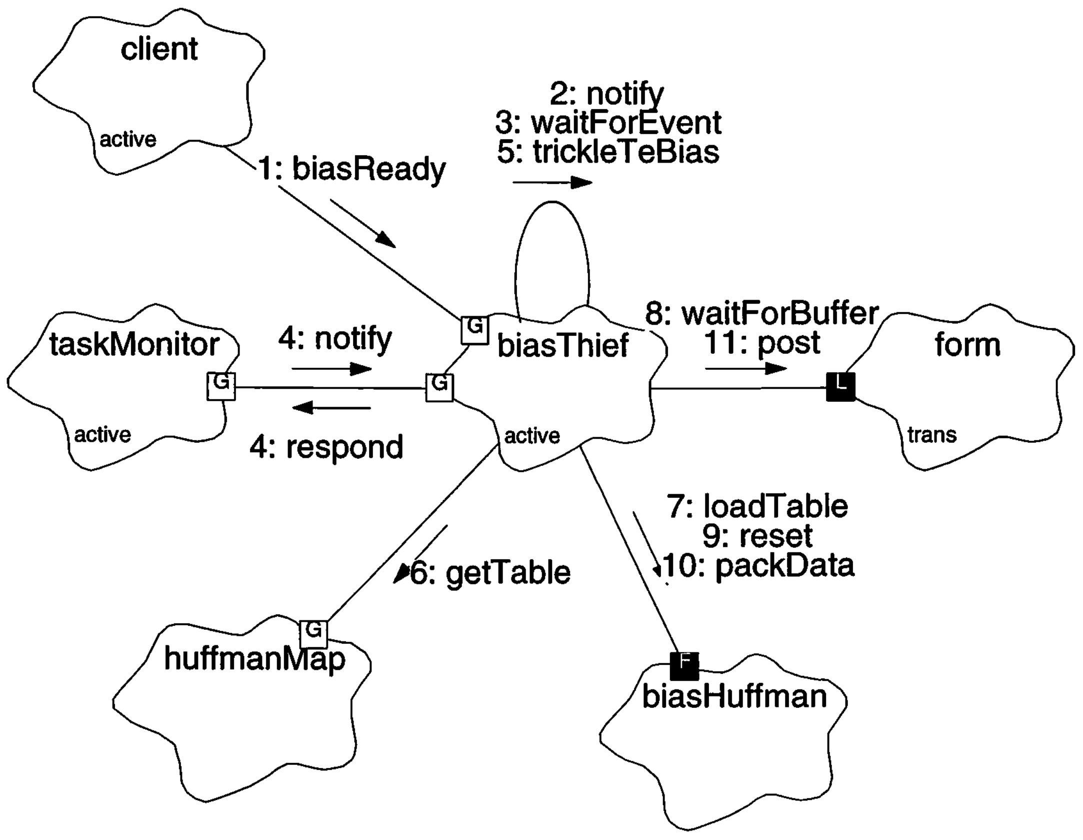
37.4.2 Use 2: Specify the bias map parameters for each Front End Processor

After selecting the bias map mode and prior to packing data, the client must specify various FEP-specific parameters for each bias map to send, using *biasThief*.`setFepInfo()`. This function selects which FEPs to use, which CCD produced the map, what the bias map base address is (in BEP address space), how many bias pixels there are per row for that FEP, how many rows there are in the map, what the initial overclock values are for the map, and which compression table to use for this map. After calling this function for each configured FEP, the *biasThief* is ready to telemeter maps.

37.4.3 Use 3: Start transmission of the pixel bias maps

Figure 167 illustrates the overall process used to telemetry the pixel bias maps from each configured FEP. Some details have been omitted to simplify the explanation.

FIGURE 167. Trickle Pixel Bias Maps



1. In order to start sending the configured bias maps, the client informs the biasThief that the maps are ready, using the binding function, *biasThief*.`biasReady`().
2. *biasThief*.`biasReady`() de-asserts the abort flag, and calls `notify`() to inform the task portion that the map is ready.
3. The main task loop, `goTaskEntry`, waits for notification that it should start, using `waitForEvent`().
4. While waiting, if it receives a query from the taskMonitor, it replies using *taskMonitor*.`respond`().
5. If it `waitForEvent`() receives a start signal, `goTaskEntry` iterates through each FEP until each of the bias maps have been sent, or until it is aborted. If a particular FEP is configured, `goTaskEntry`() calls `trickleTeBias`() or `trickleCcBias`(), depending on the mode specified by `selectMode`(), to telemeter that FEP's map. For this example, assume that a Timed Exposure bias is to be sent.
6. `trickleTeBias`() gets the address of the desired compression table using *huffmapMap*.`getTable`().

7. It passes this address to *biasHuffman*.loadTable() to copy the table from protected I-cache memory into a usable buffer.
8. trickleTeBias() constructs a telemetry format object, form, which is used to format the bias map's telemetry buffer. It obtains a buffer for the form using getBuffer() (not shown), which then calls form.waitForBuffer(). getBuffer() uses the time-out feature of form.waitForBuffer() to occasionally respond to taskMonitor queries and detect abort requests.
9. If a buffer is obtained without aborting the operation, trickleTeBias() initiates a new compression operation using *biasHuffman*.reset().
10. It then compresses a set rows of pixel bias map values directly into the telemetry buffer using *biasHuffman*.packData().
11. If the buffer becomes full, the maximum allowed rows for one telemetry packet has been packed, or the last row of the bias image has been processed, trickleTeBias() posts the telemetry packet buffer to telemetry, using *form*.post(). If there are more rows to process, it obtains a new buffer for the form, and repeats the packing operation. Once the entire bias map for the FEP has been processed, trickleTeBias() returns to its caller. The caller may then re-invoke trickleTeBias() for another FEP's map. This continues until all of the maps have been posted, or until the operation is aborted.

37.4.4 Use 4: Abort transmission of the pixel bias maps

In order to stop the bias trickle algorithm before it completes, the client calls *biasThief*.abort(). This sets the thief's abort flag, and sends an event notifying it that the operation has been aborted, using *biasThief*.notify(). If the operation has already completed, then the notification is consumed in the main loop and ignored. If the operation is waiting for a telemetry packet buffer, the notification is consumed by the call to requestEvent(), and the transmission of the current map, and maps from subsequent FEPs is aborted.

37.5 Class BiasThief

Documentation:

This class is responsible for copying the pixel bias map values from the Front End Processors, and trickling these maps to telemetry.

Export Control:	Public
Cardinality:	n

Hierarchy:

Superclasses:	**Task**

Implementation Uses:

Tf_Data_Te_Bias_Map
Tf_Data_Cc_Bias_Map
TaskMonitor *taskMonitor*
HuffmanMap *huffmanMap*

Public Interface:

Operations:

```
BiasThief()
abort()
biasReady()
goTaskEntry()
selectMode()
setFepInfo()
```

Protected Interface:

Operations:

```
checkMonitor()
getBuffer()
setupTeForm()
trickleCcBias()
trickleTeBias()
```

Private Interface:

Constants:	PIXELS_PER_ROW = 1024 locations
	BUFFER_TIMEOUT = 1 second
	MAXROWS_PER_PACKET = 10 rows

Has-A Relationships:

unsigned *timestamp*: This is a copy of the start time of the bias run.

unsigned *blockid*: This is a copy of the parameter block id used to start the run.

unsigned *modetype*: This is a copy of the type of bias map being telemetered. 0 indicates Timed-exposure, and 1 indicates Continuous Clocking.

Boolean *abortFlag*: This indicates whether the bias has been aborted or not. This flag is cleared by calls to biasReady() and is asserted by calls to abort().

static HuffmanTable *biasHuffman*: This is the Huffman Compression table object used by the Bias Thief to compress bias map data into telemetry packet buffers.

const unsigned *buffer_timeout*: This is the maximum number of timer ticks to wait for a telemetry buffer before checking for task monitor queries, or abort requests (Acis::TICKS_PER_SECOND).

const unsigned *maxrows*: This is the maximum number of rows that should be packed into a single telemetry buffer (10 TBD).

const unsigned *pixels_per_row*: This is the number of pixels locations in 1 row in the bias map (1024).

struct FepInfo *fepInfo*[6]: This is an array of information structures used to configure the bias telemetry operation for each FEP. The structure is as follows:

const unsigned short* *base*: Points to FEP's pixel bias map. 0 if FEP unused.

CcdId *ccd*: Speicifes which CCD the FEP is processing

unsigned *rowpixels*: Number of pixels in 1 bias map row

unsigned *rowcnt*: Number of rows in bias map

unsigned *scale*: Number of CCD rows summed on-chip into image row

unsigned *biasoffset*[4]: Pixel bias map offsets for each quadrant

unsigned *compress*: Compression table selection for the FEP

Concurrency:	Active
Persistence:	Persistent

37.5.1 BiasThief()

Public member of: **BiasThief**

Arguments:

unsigned *taskid*

Documentation:

This is the constructor for the Bias Thief task. *taskid* is the RTX identifier for the task. This function passes *taskid* to its parent's constructor, `Task`(), and initializes the constants used by the class.

Concurrency: Sequential

37.5.2 abort()

Public member of: **BiasThief**

Return Class: **void**

Documentation:

This function causes the bias thief to abort its current bias telemetry processing. This function sets *abortFlag* to *BoolTrue* and calls `notify`() to signal the task that an abort has been requested.

Concurrency: Synchronous

37.5.3 biasReady()

Public member of: **BiasThief**

Return Class: **void**

Documentation:

This function indicates that the bias maps for all of the enabled Front End Processors are ready to be telemetered. This function sets *abortFlag* to *BoolFalse*, and then uses notify() to inform the task that a bias map is ready to be sent.

Preconditions:

The client must first call selectMode(), and must then call setFepInfo() for each FEP in use. This is only required after a reset, and if the current mode or FEP parameters change. (Although it is not required by the class, it is strongly recommended to call these functions for each run).

Concurrency: Synchronous

37.5.4 checkMonitor()

Protected member of: **BiasThief**

Return Class: **Boolean**

Documentation:

This function responds to queries from the task monitor, and detects whether or not the current operation has been aborted. If the operation has been aborted, the function returns *BoolFalse*. If not, it returns *BoolTrue*.

Semantics:

This function uses requestEvent() to poll for the task monitor query or an abort signal. If the task monitor has issued a query, this function uses *taskMonitor*.respond() to respond to query. If an abort signal has been issued, the function returns *BoolFalse*, otherwise, it returns *BoolTrue*.

Concurrency: Synchronous

37.5.5 getBuffer()

Protected member of: **BiasThief**

Return Class: **Boolean**

Arguments:

TlmForm& *form*

Documentation:

This function waits for and allocates a buffer for the telemetry format, *form*, while responding to queries from the *taskMonitor*. If successful, the function returns *BoolTrue*. If the operation is aborted, the function returns *BoolFalse*.

Semantics:

This function consists of a loop which acquires a telemetry buffer for *form*. by passing *buffer_timeout* to *form*.waitForBuffer(). If a buffer is allocated, the function returns. If the wait times out, the function calls checkMonitor() to respond to any task monitor queries, and to detect an abort of the bias telemetry operation. If aborted, the function returns immediately. If not aborted, the loop iterates.

Concurrency: Synchronous

37.5.6 goTaskEntry()

Public member of: **BiasThief**

Return Class: **void**

Documentation:

This function contains the main loop of the bias thief task.

Semantics.

This function consists of an infinite loop. At the top of the loop, the function waits for and consumes start, abort and task monitor query signals. Abort signals are discarded. If a task monitor query signal is received, `goTaskEntry`() responds using *`taskMonitor`*.`respond`(). If a start signal is received, and *`abortFlag`* is *`BoolFalse`*, `goTaskEntry`() enters a loop which sends the bias maps for each FEP in the system. (NOTE: If *`abortFlag`* is *`BoolTrue`*, the abort request occurred after the start request, and no bias telemetry operation should be attempted). Within the loop, if the base address for a given FEP is zero, then the corresponding FEP's bias is not sent, and the loop skips to the next FEP (see `selectMode`() and `setFepInfo`()). If *`modetype`* is 0, then a Timed Exposure bias map is being sent, and the function calls `trickleTeBias`() to send the map. Otherwise, it is a Continuous Clocking bias, and `goTaskEntry`() calls `trickleCcBias`() instead. If either function returns *`BoolFalse`*, then an abort request has been received, and the bias telemetry operation from subsequent FEPs is aborted. Otherwise, the process repeats for each used FEP.

Concurrency: Synchronous

37.5.7 selectMode()

Public member of: **BiasThief**

Return Class: **void**

Arguments:

unsigned *mode*
unsigned *starttime*
unsigned *parameterId*

Documentation:

This function selects whether or not to dump Timed-Exposure bias values, or Continuous Clocking bias values, and configures the start time of the bias run, and the parameter block id used to produce the map. If *mode* is 0, Timed Exposure is used. If *mode* is 1, Continuous Clocking is used. *starttime* indicates the latched ACIS science timestamp at the start of the bias run, *parameterId* is the id from within the parameter block used to configure the run.

Preconditions:

A bias telemetry operation must not already be in progress.

Semantics:

This function saves the passed parameters in its instance variables, and then flags all FEPs as unused by zeroing the bias base address for each FEP entry in the *fepInfo*[] array.

Postconditions:

No FEPs are configured to be used. For each FEP to be used, `setFepInfo`() must be called to set the bias map telemetry parameters for the corresponding FEP.

Concurrency: Synchronous

37.5.8 setFepInfo()

Public member of: **BiasThief**

Return Class: **void**

Arguments:

FepId *fepid*
const unsigned short* *base*
CcdId *ccd*
unsigned *rowpixels*
unsigned *rowcnt*
const unsigned *biasoffset[4]*
unsigned *compress*
unsigned *scale*

Documentation:

This function sets up the bias thief to steal pixel bias values from a particular FEP, indicated by *fepid*. *base* is the base address, within the FEP, of the bias map. *ccd* indicates which CCD produce the map. *rowpixels* is the number of map values in each row, and *rowcnt* is the number of rows from the map to telemeter. The *biasoffsets* array contains the pixel initial overclocks for each video chain. *table* specifies which compression table to use (ignored if continuous clocking is being performed). *scale* is the number of CCD rows in each image row (i.e. in 2x2 summing, there are two CCD rows summed into 1 image row).

Preconditions:

A bias telemetry operation must not already be in progress, and `selectMode`() must have been called.

Semantics:

This function saves the passed parameters in the *fepInfo*[] entry indexed by *fepid*.

Postconditions:

Once started, the bias map from *fepid* will be telemetered.

Concurrency: Synchronous

37.5.9 setupTeForm()

Protected member of: **BiasThief**

Return Class: **void**

Arguments:

Tf_Data_Te_Bias_Map& *form*
unsigned *packetNum*
unsigned *pixelrow*
FepId *fepid*

Documentation:

This function sets up the Timed Exposure telemetry form, *form*. *packetNum* is the packet number in the series for this FEP, and *pixelrow* is the row number, in image coordinates, of the first row in the packet. Since data is packed last row to first, subsequent rows in the packet have decrementing positions. *fepid* is the id of the FEP whose bias map is being sent.

Preconditions:

The form must have allocated a telemetry packet buffer.

Semantics:

This function uses the *form* to store the passed information into the telemetry packet buffer, and zeros the bias data length using *form*.setEmpty().

Concurrency: Synchronous

37.5.10 trickleCcBias()

Protected member of: **BiasThief**

Return Class: **Boolean**

Arguments:

FepId *fepid*

Documentation:

This function trickles the Continuous Clocking bias map from the FEP indicated by *fepid*. If successful, the function returns *BoolTrue*. If it is aborted, it returns *BoolFalse*.

Semantics:

This function first calls `yield()` to allow other tasks of the same priority to run. It then calls `checkMonitor()` to respond to any task monitor queries, and to check for any abort requests. If aborted, `trickleCcBias()` returns immediately. If not aborted, it proceeds to send the bias. It first passes a NULL table pointer to *biasHuffman*.`loadTable()` to configure the data compression algorithm to bit-pack the data, without compressing it, and *biasHuffman*.`reset()` to initialize the state of the packing algorithm. It then creates a bias telemetry form, *form*, and calls `getBuffer()` to obtain a telemetry packet buffer. If `getBuffer()` indicates an abort, the function returns immediately. Once a buffer has been obtained, the function zeros the bias data length of the buffer, and uses the form to set the start time, parameter block id, CCD Id, FEP Id, and initial overclocks into the telemetry buffer. It then uses *form*.`get_Data_Address()` and form.`get_Data_Avail()` to get the bias data buffer address and length within the telemetry packet buffer. It then passes these to *biasHuffman*.`packData()` to pack the one continuous clocking bias map row into the telemetry packet buffer. It calls *form*.`set_Data_Written()` to set the bias data word count in the buffer, and then uses *form*.`post()` to post the packet buffer to telemetry.

Concurrency: Synchronous

37.5.11 trickleTeBias()

Protected member of: **BiasThief**

Return Class: **Boolean**

Arguments:

FepId *fepid*

Documentation:

This function trickles the Timed Exposure bias map from the FEP indicated by *fepid*. If successful, the function returns *BoolTrue*. If it is aborted, then it returns *BoolFalse*.

Semantics:

This function uses *huffmanMap*.getTable() to obtain the table pointer for the FEP's compression selection, and passes this pointer to *biasHuffman*.loadTable(). It then initializes some local variables and enters its row processing loop. Rows are processed in reverse order, from the end of the bias map to the beginning.

On each iteration, the function calls yield() to allow other tasks of the same priority to run, and then checkMonitor() to respond to task monitor queries and detect abort requests. If aborted, the function returns immediately. The loop checks to see of the telemetry form, *form*, has a buffer using *form*.hasBuffer(), and if not, calls getBuffer() to allocate a buffer, setupTeForm() to initialize its contents, and *biasHuffman*.reset() to reset the compression state.

It then calls *biasHuffman*.packData() to pack one row of bias data to the end of the telemetry buffer. If the entire row fit, it updates its row information. If the row does not fit into the buffer, or if the last row in the map has been packed, or if the maximum number of rows per packet have been put into the telemetry buffer, the function uses *form* to store the total number words written, and the total number of rows written into the telemetry buffer. It then uses *form*.post() to post the buffer to telemetry.

NOTE: If the *form* has a telemetry buffer when an abort is detected, the destructor for the form will release the buffer back into its pool. This prevents the abort causing buffers to be "lost."

Concurrency: Synchronous