CSR

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

REVISION LOG

TITLE: DEA HOUSEKEEPER

DOC. NO. 36-53221 ~~CE~~

| Revision | Date (mm/dd/yy) | ECO No. | Page(s) Affected | Reason | Approval |
|---|---|---|---|---|---|
| 01 | 09/22/95 | 36-357 | ALL | Initial Release | |

# 4.0 DEA Housekeeper (36-53221 01)

## 4.1 Purpose

The DEA Housekeeper function provides the capability to acquire the contents of a specific set of DEA registers at a prescribed interval rate.

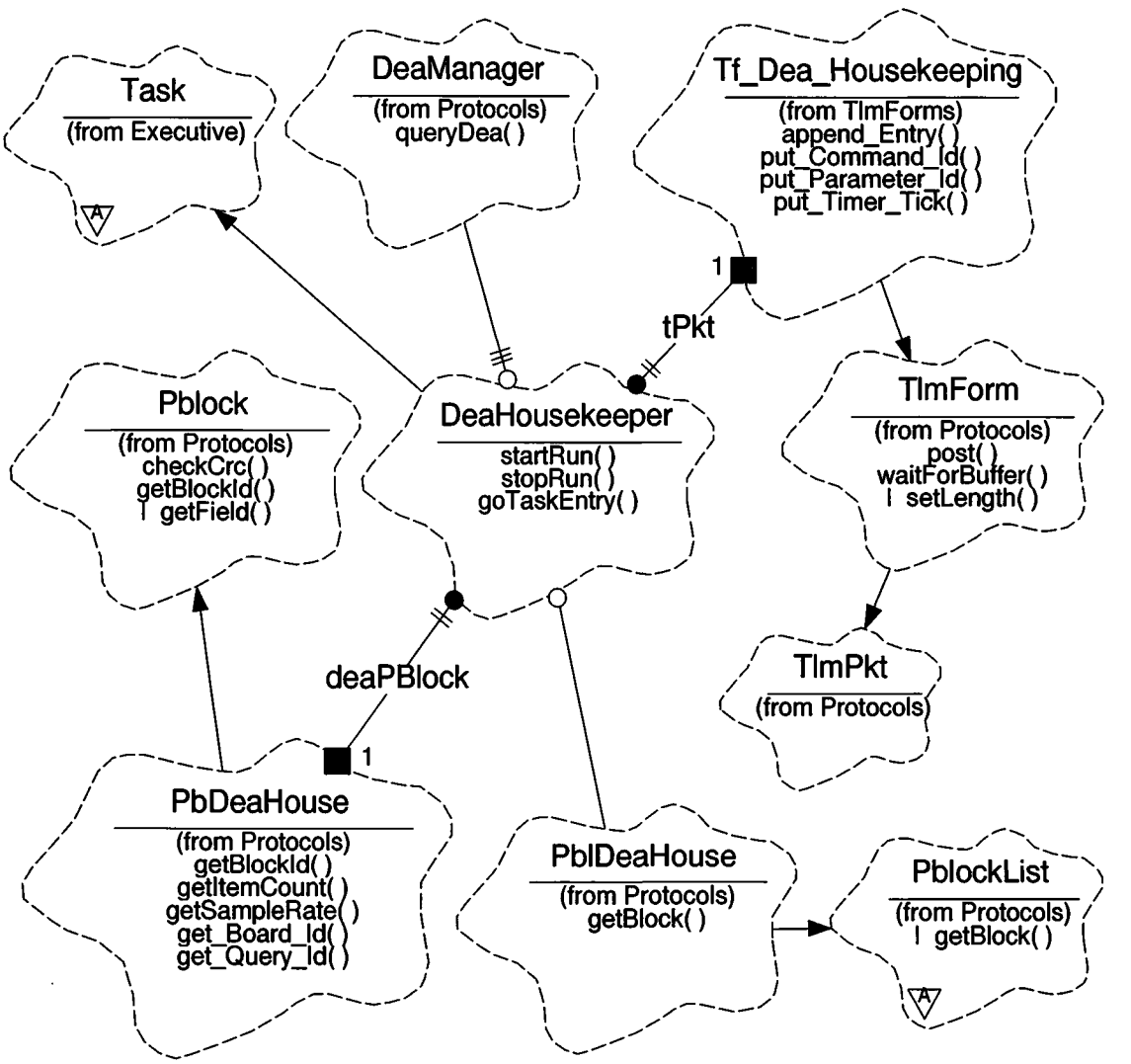
## 4.2 Uses

Use 1:: A method of acquiring the selected DEA register values.

## 4.3 Organization

Figure 1 illustrates the relationship between the classes used by the DEA Housekeeper.

**FIGURE 1. DEA Housekeeping Class Relationships**



The DEA Housekeeper uses the ***Executive*** and ***Protocols*** class categories.

**DeaHousekeeper** - This class is a subclass of ***Executive***::**Task**. It is responsible for accumulating and delivering DEA Housekeeping data as specified in the referenced parameter block, from the receipt of a start command until receipt of a stop command.

**Tf_Dea_Housekeeping** - This class encapsulates the representation of a telemetry packet. It is a subclass of ***Protocols***::**TlmForm**.

***Pblock*** - This class is responsible for manipulating the contents of a parameter block; extracting fields and verifying the block. It is a subclass of ***protocols***.

**PbDeaHouse** - This class encapsulates the representation of a parameter block. It is a subclass of ***Protocols***::**Pblock**.

**DeaManager** - This class is responsible for interaction with the DEAs, when requested to acquire specified housekeeping data. It is a subclass of ***Protocols***.

**PblDeaHouse** - This class is responsible for loading, maintaining, and delivering the sets of parameter blocks. It is a subclass of ***Protocols***::**pBlockList**.

**TaskMonitor** - This class (not shown) is a subclass of ***Executive***::**Task**. It interrogates each thread in turn, verifying that it is functioning. Failure to respond will cause a watchdog reset.

`notify` and `waitForEvent` and `requestEvent` - These are functions of ***Executive***::**Task** inherited by the DEA Housekeeper. They provide the connectivity between the clients request via the housekeeper public functions and the threads main process.

**SystemClock** - This class (not shown) provides the BEP tick count which is included in the telemetry Packet. It is a subclass of ***Executive***.

**Note**: --- The New IP&CL generated names for associated commands and functions are: `CmdPkt_Load_Dea_Block`, `CmdPkt_Start_Dea`, `CmdPkt_Stop_Dea`, and **Tf_Dea_Housekeeping**.

The current thinking is that the entire command which loads the DEA Parameter Block will be stored into the block - TBD.

The block will contain: a command packet header, a DEA housekeeping header, and a set of target identifiers. The command packet header consisting of: message length (U16), packet identifier (U16), and an op-Code (U16). The DEA housekeeping header contains, the slot index (U16), the parameter block id (U32), the block CRC (U16), and the data sampling rate (U32). This is followed by a series of one word (U16) target identifiers which indicate the board Id (bits8-11) and the register Id (bits 0-7), (bits12-15 unused). The number of target entries is derived from the total length less the two headers - TBD.

## 4.4 Scenario

The `startRun` command consists of a command packet header and the slot index from which the block will be extracted for the run. The `stopRun` command consists of a command packet header - TBD. There are three flags to control the processing; *`startFlag`*, *`workingFlag`*, and *`stopFlag`*. The start request will obtain a verified parameter block, set *`startFlag`* and `notify` the thread, `goTaskEntry`. When the idle thread receives the notification it will establish housekeeping according to the block delivered, clear *`startFlag`*, set *`workingFlag`*, and proceed until the set *`stopFlag`* is detected, when it will return to the idle state with all flags cleared. A stop request will verify the current block and set the stop flag, and notify the thread.

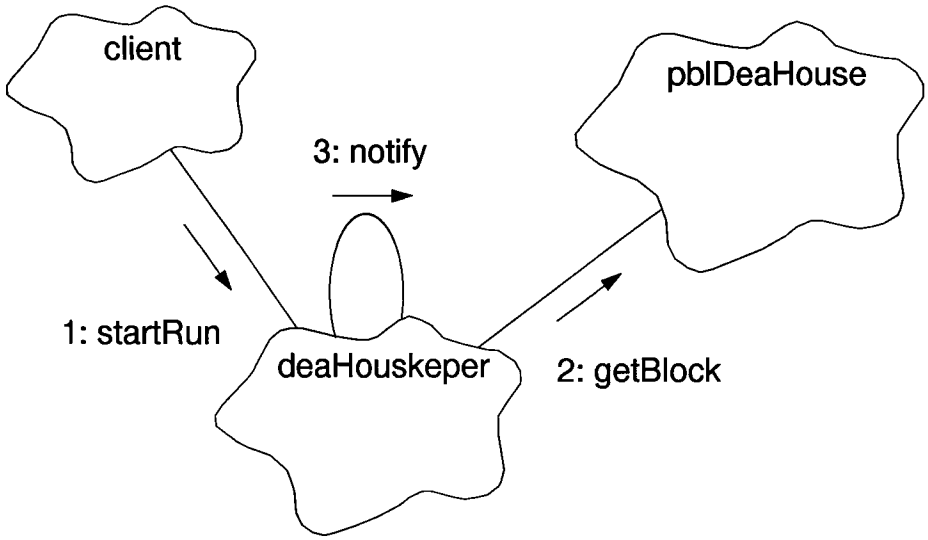
### 4.4.1 USE 1::A method of acquiring the selected DEA register values

#### 4.4.1.1 Start DEA Housekeeping

Figure 2 illustrates the initiation of the DEA housekeeping process.

1. To begin a DEA housekeeping run, the client must provide the slot index from which the parameter block will be extracted for the run, and the command Id from the packet header which is to be echoed in the telemetry identifying the instigating agent for the action. Then it must engage this tasks public function `startRun` to start DEA housekeeping.

**FIGURE 2. Start DEA Housekeeping**




2. When a client requests that a run be started, an instance of **`PBlock`** is created. `startRun` will acquire a copy of the designated block using **`PblDeaHouse`**::`getBlock` which accesses its inherited **`PblockList`**::`getBlock` which will lock the block against incursion by another

process during copying. On failure to get the lock within a prescribed time, it returns false. After copying the block, `getBlock` performs a CRC check on the copy, returning false if it is not validated. If a block is not delivered, the process returns CMDRESULT_BAD_LOAD. The **`PBlock`** instance holds the copy.

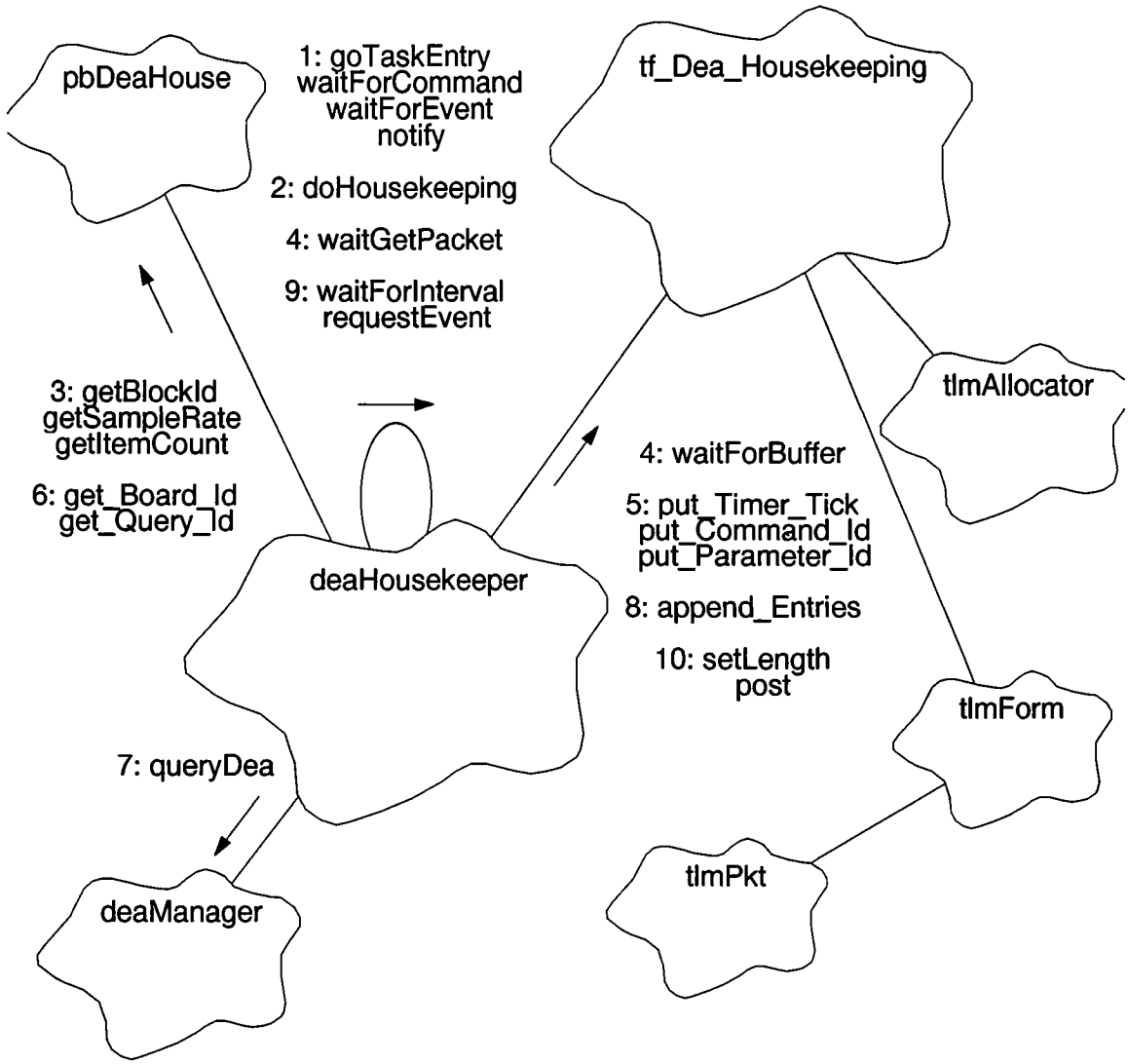
3. The current state of the thread is checked. Should the thread be executing a prior run request, a missed stop command is assumed. In that case the interface will set both the stop and start flags to generate termination of the current run and initiation of the requested run. It will set the return argument to CMDRESULT_ALREADY. In this event, the last data-point in the current packet may reference the new parameter block. The client (via command response echo?) has the responsibility to convey this possibility. If the current state of the thread is stopped, the process will set the *`startFlag`* and set the return argument to CMDRESULT_OK. In either case, the `startRun` function will send a `notify` request to the thread before returning.

#### 4.4.1.2 Acquiring DEA Housekeeping Data

Figure 3 illustrates the acquisition of the DEA housekeeping process.

1. The main thread of this task, `goTaskEntry`, is initiated during BEP start-up. It consists of a forever loop in which the task idles in `waitForCommand` using the inherited `waitForEvent` while waiting to call **`TaskMonitor`**`::respond` to answer the **`TaskMonitor`**`::query` (neither shown) or to the `notify` sent by the `startRun` or `stopRun` public functions indicating that a request has been received and awaits proper action. Receipt of a `startRun` request causes the task to enter a loop in which it calls (or recalls) `doHousekeeping` whenever the *`startFlag`* is set (or reset).
2. In `doHousekeeping`, the process sets the *`workingFlag`* and clears the *`startFlag`*.
3. The DEA header values, block Id and sample rate, are extracted from the parameter block using `getBlockId` and `getSampleRate`, and the item count is returned by `getItemCount`.
4. It then enters a loop which it exits only when a stop run request flag is detected. It creates an instance of the DEA telemetry packet, **`Tf_Dea_Housekeeping`**. Then it enters `waitGetPacket` to obtain a packet buffer using the packet instance's `waitForBuffer` which it inherits through **`TlmForm`**. Obtaining the buffer, it will return.
5. Getting the time reference from **`systemClock`**`::currentTime` (not shown), `doHousekeeping` will install that and the command and block Ids into the DEA Header of the packet using functions provided by the packet instance. They are: `put_timer_Tick`, `put_Command_Id`, and `put_Parameter_Id`.
6. The target indicators, the board index and the register index, are extracted from the parameter block using `get_Board_Id`, and `get_Query_Id` respectively. TBD
7. The function drops into a loop in which it obtains and unpacks the DEA board Id and the register reference. It uses the **`deaManager`**`::queryDea` to obtain the targeted register value, providing a default value if there is no response. The loop is cycled for *`itemCount`* times.

**FIGURE 3. Acquisition of Housekeeping Data**



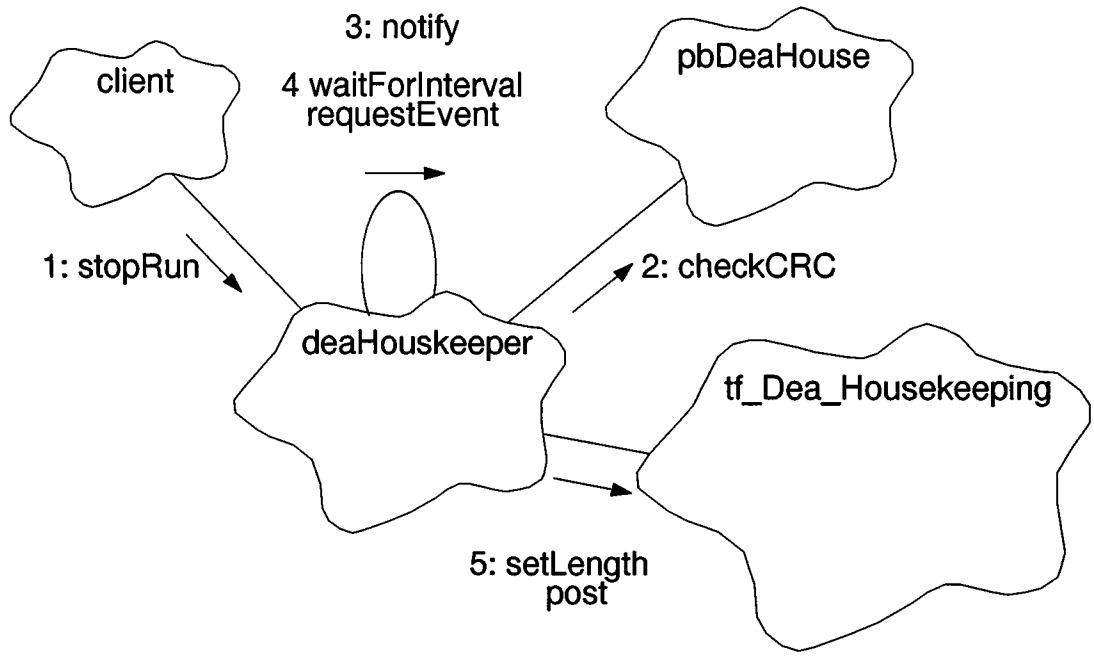

8. It then uses the packet instance function `append_Entries` to load the target Identifiers, *`boardIndex`* and *`queryIndex`*, and the register response *`value`* into the packet.
9. The process then tests the *`stopFlag`*, exiting the loop if it is set, else it uses `waitForInterval` to check for notifications from the **`TaskMonitor`** or from this tasks public functions using the inherited function `requestEvent`. It then waits the requisite interval, *`sampleRate`*, before processing the next data point.
10. Having completed the data set, or having been told to stop, thus truncating the data set, the function will pass the data length to the packet instance using the inherited function `setLength`, and forward the packet using the inherited function `post`. If a stop has been requested it will clear the *`workingFlag`* and the *`stopFlag`* before returning.

### 4.4.1.3 Stop DEA Housekeeping

Figure 4 illustrates the termination of the DEA housekeeping process.

1. The stop run request is handled by the *stopRun* public function called by the client.
2. It will use checkCRC to verify the current parameter block returning CMDRESULT_BAD_CRC on failure. If the thread is not processing data, the interface assumes that it is stopped and will return CMDRESULT_ALREADY, else it will return CMDRESULT_OK.
3. Regardless of the outcome of these checks, the interface will notify the thread to stop the run. If goTaskEntry is waiting for a command, it will receive the notification from the stopRun function. The stop run request is ignored by goTaskEntry, since, if one is received, the process is already stopped, and no action is required.

**FIGURE 4. Stop DEA Housekeeping**




4. If housekeeping data is being collected, doHousekeeping will receive the request, and act upon it. In the course of its processing, after the acquisition of each data value, doHousekeeping enters waitForInterval which uses requestEvent to check pending notifications from **TaskMonitor** and from the public functions startRun and stopRun. Receiving a notification from either public function will cause a return without delaying for the *sampleRate* interval.
5. The *stopFlag* will be set; since, if a start was requested the current run must be stopped, and if a stop was received a stop is appropriate. The function will terminate the run and finalize using setLength and deliver the packet using post.

## 4.5 Class DeaHousekeeper

Documentation:

The DEA Housekeeper is responsible for modulated acquisition and periodic telemetry of engineering information from the DEA subsystem. It is started and stopped by command.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`Task`**

Public Uses:

**`PblockList`**
**`PblDeaHouse`**

Implementation Uses:

**`DeaManager`**

Public Interface:

Operations:
`DeaHousekeeper()`
`goTaskEntry()`
`startRun()`
`stopRun()`

Protected Interface:

Has-A Relationships:

**`unsigned`** *`cmdGndId`*: The value the ground uses to identify a command. It is echoed in packets generated in response to the command.

**`unsigned`** *`sampleRate`*: Initially holds the number of seconds which should elapse between requests for housekeeping data. It is multiplied by TICKS_PER_SECOND to be used by `sleep()`.

Operations:

```
doHousekeeping()
waitForCommand()
waitForInterval()
waitGetPacket()
```

Private Interface:

Has-A Relationships:

**PbDeaHouse** *deaPBlock:* An instance of a DEA parameter block.

**Boolean** *startFlag*: This instance variable conveys the action requested by the client, from the public binding function to the housekeeper. it will retain the state of the request to begin housekeeping.

**Boolean** *stopFlag*: This instance variable conveys the action requested by the client, from the public binding function to the housekeeper. It will reflect the state of the request to terminate housekeeping.

**Tf_Dea_Housekeeping** *tPkt*: An instance of a DEA telemetry packet.

**Boolean** *workingFlag*: This instance variable contains the state of the housekeeper processing.

Concurrency: Active

Persistence: Persistent

### 4.5.1 goTaskEntry()

Public member of: `DeaHousekeeper`

Return Class: `void`

Documentation:

This function is the main entry point of the task.

Semantics:

This thread is idle unless commanded to start data acquisition according to parameters stored in a designated block. A start run notification will initiate housekeeping by calling doHousekeeping. Housekeeping data will continue to be obtained until the housekeeper is commanded to stop. A stop notification received by this function is ignored: because it is stopped.

Concurrency: Sequential

### 4.5.2 startRun()

Public member of: **DeaHousekeeper**

Return Class: **CmdResult**

Arguments:

**unsigned** *slotIndex*
**unsigned** *cmdGndId*

Documentation:

This function acquires and verifies the designated block (CRC check) and instructs the DEA Housekeeper task to start acquiring and sending information from the DEA. *slotIndex* is the index (acis location) of the DEA Housekeeping parameter block to use for the run. *cmdGndId* is the ground designation of the command sent to instigate this action. It is included in response telemetry

Semantics:

This function will create an instance of a parameter block then extract a copy of the designated block using **pblDeaHouse**::getBlock. A faulty attempt caused by inability to lock the parameter access semaphore, or a bad CRC check of the block, will cause a return with state CMDRESULT_BAD_LOAD. With a valid parameter block, the threads state is tested, if it is running, the stop and start flags are set and the return state is set to CMDRESULT_ALREADY, but if it is stopped, the start flag is set and the return state set to CMDRESULT_OK. The function will use notify to inform the thread of the request before returning.

Concurrency: Sequential

### 4.5.3 stopRun()

Public member of: **DeaHousekeeper**

Return Class: **CmdResult**

Documentation:

This function instructs the DEA housekeeper to stop acquiring and sending information from the DEA. The last data obtained will be posted.

Semantics:

This function will check the current block CRC using **deaPBlock**::checkCrc and set the return state CMDRESULT_BAD_CRC if it is invalid. It will set the state CMDRESULT_ALREADY if processing state is stopped, otherwise it will set the state to CMDRESULT_OK. The thread will be informed using notify.

Concurrency: Sequential

### 4.5.4 waitForCommand()

Protected member of: **DeaHousekeeper**

Return Class: **void**

Documentation:

This function waits until the task is commanded to start (or to stop) acquiring housekeeping values while responding to **TaskMonitor::**query.

Semantics:

This function consists of a loop in which it waits; using waitForEvent for notification of a **TaskMonitor**::query which initiates **TaskMonitor**::respond, or a request from one of its public functions which allows it to leave the loop and return.

. Concurrency: Guarded

### 4.5.5 doHousekeeping()

Protected member of: `DeaHousekeeper`

Return Class: `void`

Documentation:

This function periodically acquires and telemeters DEA housekeeping values. The parameter block to use was loaded by the public *`startRun`* function. This function runs until commanded to stop.

Semantics:

The status flags are adjusted. The parameter block entries for the science identifier, the count of housekeeping items, and the rate at which housekeeping data is acquired are obtained using `getBlockId`, `getItemCount`, and `getSampleRate`. The main process loop is entered and will be cycled until a stop run directive is detected. A **`Tf_Dea_Housekeeping`** packet instance is passed to `waitGetPacket` to be associated with a packet buffer. The packet functions will load data into the packet. `currentTime` is used to obtain the BEP tick count. The tick count, the command identity which was received from the client, and the science block identity are installed in the packet using `put_Timer_Tick`, `put_Command_Id`, and `put_Parameter_Id`. A housekeeping data loop is entered during which the target board and register indices will be extracted from the block using `get_Board_Id` and `get_Query_Id`. Then a response value from the target requested from **`deaManager`**::`queryDea` or a default value will be installed in the packet using `append_Entries`. If a stop request is detected, the loop will be terminated. `waitForInterval` is used to check for commands and for queries from the **`TaskMonitor`**. When the housekeeping requests have been fulfilled, or the run terminated, the packet length is set with `setLength` and, accessing `post`, the packet is forwarded for transmission. If a stop request is detected, the status flags will be adjusted prior to returning to the main thread.

Concurrency: Guarded

### 4.5.6 waitForInterval()

Protected member of: **DeaHousekeeper**

Return Class: **void**

Documentation:

This function waits for *sampleRate* seconds. It is used to space out engineering value acquisitions while responding to queries from the **TaskMonitor** or for a command to stop housekeeping.

Semantics:

This function uses *requestEvent* to determine if the **TaskMonitor** has interrogated this process or if a public function has sent a notify to this process of a received command. If so, it uses **TaskMonitor**::respond to answer the **TaskMonitor** or returns so its caller may address the command. With no pending directive, it will sleep for the designated interval, and return.

Concurrency: Guarded

### 4.5.7 DeaHousekeeper()

Public member of: **DeaHousekeeper**

Arguments:

**unsigned** *taskId*

Documentation:

The identity of this task is contained in *taskId*. This constructor initializes instance variables.

Concurrency: Sequential

### 4.5.8 waitGetPacket()

Private member of: **DeaHousekeeper**

Return Class: **void**

Arguments:

**TlmForm** & *pkt*

Documentation:

This function waits until a packet buffer becomes available while checking for **TaskMonitor** notifications. The buffer, in due course, will contain the data of the packer instance *pkt*.

Semantics:

This function checks for the **TaskMonitor**::query, then waits for a fresh packet for a timed duration using waitForBuffer. On failure to obtain a packet in that time, it loops to check and to wait again. When a packet buffer is obtained, it will return.

Concurrency: Guarded