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

REVISION LOG	**TITLE: Software Detailed Design DEA Management Class**	**DOC. NO. 36-53218 Rev. A**

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
A	4/10/96	36-577	all	Initial version. Incorporated comments from initial review.	[illegible] 4/22/96

26.0 DEA Management Class (36-53218 A)

26.1 Purpose

The purpose of the Detector Electronics Assembly (DEA) Management class is to manage access to the DEA CCD-controller and system boards.

26.2 Uses

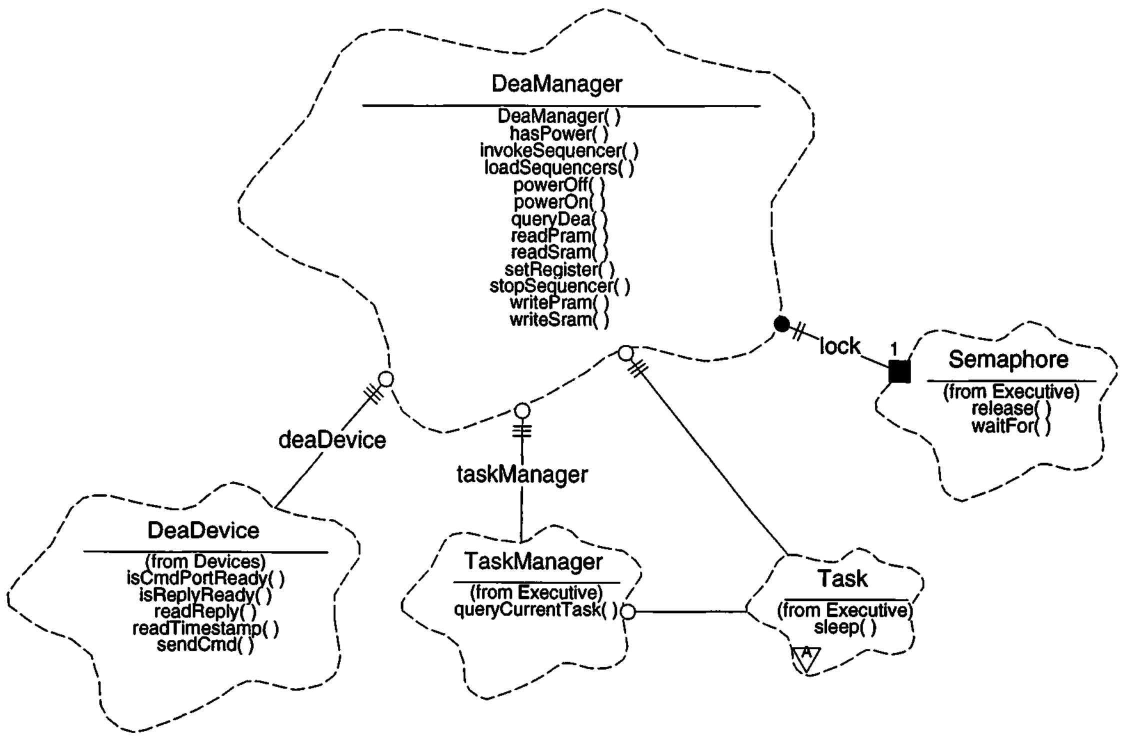
The following lists the use of the DEA Manager:

Use 1:: Load values into a CCD-controller's Program or Sequencer RAM
Use 2:: Read values stored into a CCD-controller's Program or Sequencer RAM
Use 3:: Start the CCD-controller sequencers
Use 4:: Stop the CCD-controller sequencers
Use 5:: Set the value of one of the DEA Registers or Digital-to-Analog converters
Use 6:: Obtain the current value of one of the DEA Registers
Use 7:: Enable and disable power to a single DEA CCD-controller board

26.3 Organization

Figure 104 illustrates the class relationships used by the Detector Electronics Assembly Manager class, **DeaManager**.

FIGURE 104. DeaManager Class Relationships



DeaManager - The **DeaManager** class is responsible for managing access to the Detector Electronics Assembly. It provides functions which read and write the contents of a CCD Controller board's Program and Sequencer RAM (`readPram`, `readSram`, `writePram`, `writeSram`), and to load complete images into the controller's RAM (`loadSequencers`). It provides functions to set control and digital-to-analog register values in one of the DEA boards and to query the values in the control and housekeeping registers (`setRegister`, `queryDea`). It provides functions to start and stop all of the DEA sequencers (`invokeSequencer`, `stopSequencer`). It also provides functions to individually power CCD-controller boards on and off and to determine if it has enabled power to a particular board (`powerOn`, `powerOff`, `hasPower`).

Semaphore - This class is supplied by the ***Executive*** class category. This class represents a resource lock or flag. Its implementation uses the underlying resource facilities of the RTX. The **DeaManager** contains an instance of this class, called *`lock`*, and uses this class to arbitrate access to the DEA interface. Prior to using the interface, the **DeaManager** attempts wait for exclusive access to the interface and reserve the semaphore (`waitFor`). Once the **DeaManager** has completed its action, it releases the semaphore (`release`).

DeaDevice - This class is supplied by the ***Devices*** class category. It is responsible for managing the physical interface between the Back End Processor and the Detector Electronics Assembly. The **DeaManager** uses this class to issue commands to the DEA and to retrieve status back from the DEA. This class provides functions which reset the DEA's interface board (`reset()` not shown), which indicate whether or not the command port to the DEA is ready to accept another command (`isCmdPortReady`), whether or not a status word has been received from the DEA (`isReplyRead`). It provides functions to write a command to be sent the DEA (`sendCmd`) and read status information sent back (`readReply`). The DeaDevice also provides a function which reads and returns the version of the science time-stamp, latched when the last command was issued to the DEA (`readTimestamp`).

TaskManager - This class is supplied by the ***Executive*** class category. It is responsible for managing the collection of tasks running within the Back End Processor. The **DeaManager** uses this class to obtain a pointer to the currently active task (`queryCurrentTask`).

Task - This class is supplied by the ***Executive*** class category. It represents and controls an active running task. The **DeaManager** uses this class to cause the current task to relinquish control to other tasks for a period of time (`sleep`).

26.4 DEA Manager Design Issues

26.4.1 Command Timing

This section describes how much time the DEA Manager must allow for commands to be transmitted to the DEA, and how much time it must allow for commands to be executed by the DEA. When issuing back-to-back commands, the DEA Manager must wait until the first command has been transmitted to the DEA and executed by the DEA before issuing a second command. If it sends the second command too close to the first, the DEA will ignore the second. Except for commands which demand a response from the DEA, there is no physical handshaking between the Back End Processor and the DEA. The command spacing requirements must be handled using time-delays.

The Back End Processor sends 24-bit commands to the DEA via a 1Mbps serial interface. It takes approximately 24us after the Back End writes a command word to the serial interface and the time at which it has been received by the DEA.

The time taken to execute a command varies from command to command. At a minimum, each command (including board and address selection) takes the DEA 5us (TBD) to execute.

The following table lists the various command types and their respective execution times:

TABLE 24. DEA Command Timing

Command	Register Type	Execution Time	Reply Time	Total Time (including transmission)
Select Card	N/A	9us TBD	N/A	33us
Write Address	N/A	9us TBD	N/A	33us
Write Data	PRAM/SRAM	9us TBD	N/A	33us
Write Data	A/D Control Register	200us TBD	N/A	224us
Write Data	Sequencer Control	19us TBD	N/A	43us
Read Data	PRAM/SRAM	14us TBD	24us (TBD)	62us
Read Data	Housekeeping	56.5us TBD	45.5us (TBD)	121us
Read Data	Control Register	9us TBD	45.5us (TBD)	78.5us

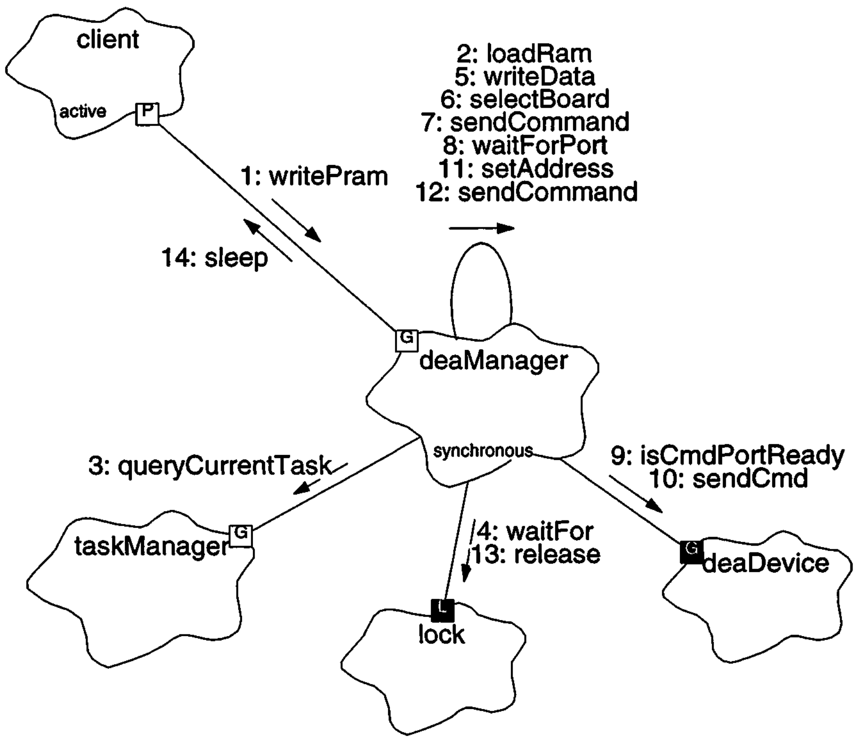
Assuming that the bulk of commanding to the DEA is to load Program and Sequencer RAM (PRAM and SRAM), the DEA Manager takes three approaches to command timing, fast commanding, and slow commanding. For commands which select boards, addresses and load RAM, the DEA Manager attempts to issue the commands as quickly as possible. For other commands which do not cause a response from the DEA, the DEA Manager adds the maximum delay between each command. For commands which request a response from the DEA, the DEA Manager blocks until the response has been received.

26.5 Scenarios

26.5.1 Use 1: Load Program or Sequencer RAM

Figure 105 illustrates the sequence of actions when a client instructs the **DeaManager** to load a section of Program RAM (PRAM). The scenario to load Sequencer RAM (SRAM) is the same except for the initial call to `writeSram()`.

FIGURE 105. Load Program RAM Scenario



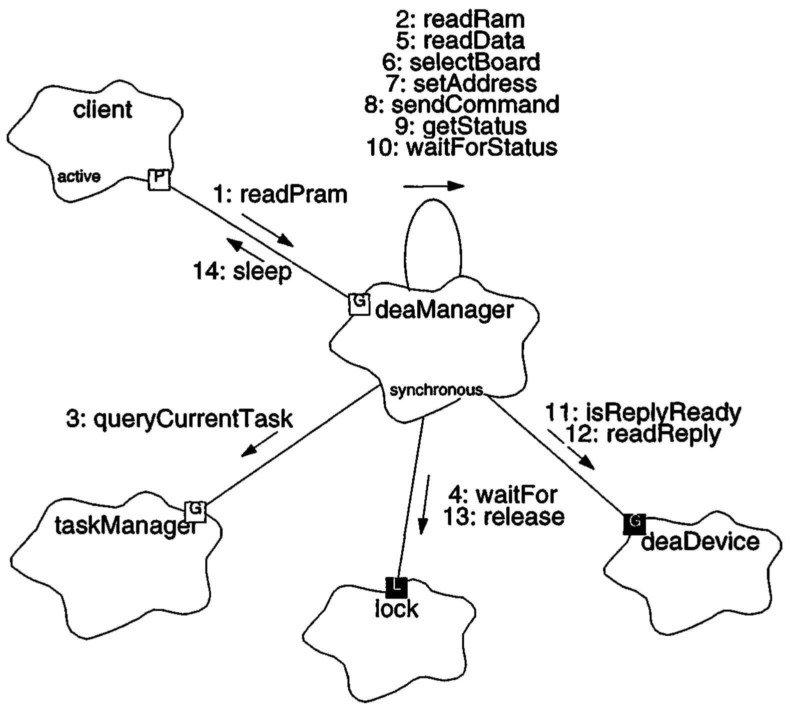
1. The *client* tells the *deaManager* to load a section of PRAM using *deaManager*.`writePram()`.
2. The *deaManager* maps the requested PRAM address into an absolute DEA address, and calls `loadRam()` to perform the load.
3. `loadRam()` gets and saves a pointer to the running task using *taskManager*.`queryCurrentTask()`.
4. `loadRam()` enters a loop where it loads each word into the DEA PRAM. The loop terminates once all words have been loaded, or on an error. The loop starts by symbolically obtaining exclusive access to the DEA by calling *lock*.`waitFor()`.
5. `loadRam()` then checks an internal flag indicating if the sequencer is running or if the board is off (`hasPower()` not shown), and if so, releases the lock (*lock*.`release()` not shown) and returns an error. If the sequencer is not running, the `loadRam()` calls `writeData()` to write one word into the DEA board's RAM. For this scenario, assume that the sequencer is not running.

6. writeData() calls selectBoard() to ensure that the targeted DEA board has been selected.
7. selectBoard() checks the desired board against the currently selected board, and if different, issues a command to select the desired board using sendCommand(). For the purposes of this scenario, assume that selectBoard() needs to issue the command.
8. sendCommand() waits for the DEA command port to complete any transmissions in progress by calling waitForPort().
9. waitForPort() consists of a loop which polls the status of the DEA command port using *deaDevice*.isCmdPortReady(). If the loop's count expires, waitForPort() returns an error. If the command port finishes its transmission before the loop count terminates, waitForPort() returns that the port is ready. The loop count is chosen such that the time to exhaust the count is longer than the time to send one command to the DEA. For this scenario, assume that the previous command transmission completes.
10. Once the port is ready, sendCommand() performs a short busy-loop to allow the previous command to execute on the DEA, and then writes the new command to the DEA command port using *deaDevice*.sendCmd().
11. Once the desired board has been selected, selectBoard() returns to writeData(). writeData() then sets the destination address for the memory write using setAddress(). setAddress(), in turn, calls sendCommand() to issue the command (not shown).
12. writeData() then calls sendCommand() directly to write the data value into the DEA RAM, and then returns to loadRam().
13. loadRam()'s write loop then releases its hold on the DEA port, using *lock*.release().
14. loadRam() then checks how many words it has written since pausing to allow other tasks to run. If the number of words written exceeds the class-specific limit (defined by **DeaManager**::DEATIME_RAM_ITERATIONS), loadRam(), using its saved task pointer, tells the current task to sleep for a few tenths of a second, using *client*->sleep(). loadRam()'s loop then repeats from step 4 until all of the data words have been loaded into the DEA board's RAM.

26.5.2 Use 2: Read Program or Sequencer RAM

Figure 106 illustrates the sequence of actions when a client instructs the **DeaManager** to load a section of Program RAM (PRAM). The scenario to read Sequencer RAM (SRAM) is the same except for the initial call to `readSram()`.

FIGURE 106. Read Program RAM Scenario



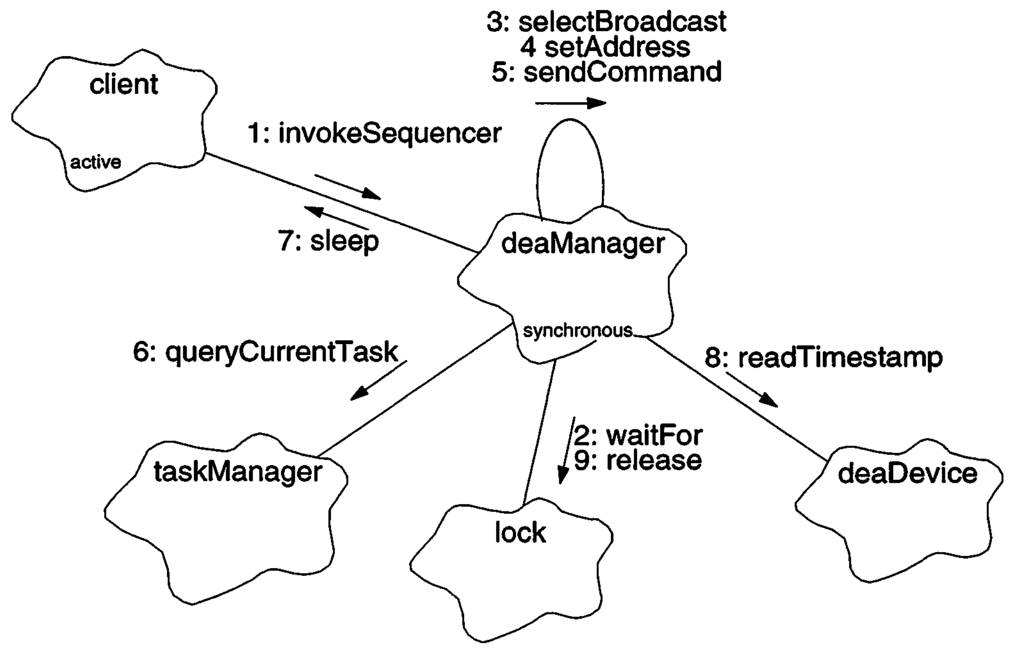
1. The *`client`* tells the *`deaManager`* to read section of PRAM using *`deaManager`*`.readPram()`.
2. `readPram()` converts the SRAM index into a DEA address, and calls `readRam()` to perform the read.
3. `readRam()` obtains and saves a pointer to the currently running task, using *`taskManager`*`.queryCurrentTask()`.
4. `readRam()` then enters its acquisition loop. The loop terminates once all of the requested words have been read, or when an error is encountered. The body of the loop starts by symbolically obtaining exclusive access to the DEA, using *`lock`*`.waitFor()`.
5. `readRam()` then checks to see if sequencer is running or if the board is off. If so, it unlocks the DEA and returns an error. If not, it proceeds to read one word from the DEA using `readData()`. Assume for this scenario that the sequencer is not running.
6. `readData()` selects the desired board using `selectBoard()` (see Section 26.5.1 for more detail).
7. `readData()` then sets the desired read address using `setAddress()`.

8. readData() then issues a data read command to the DEA using sendCommand().
9. Once the command has been issued, readData() obtains the returned data word using getStatus().
10. getStatus() waits for the returned value to arrive using waitForStatus().
11. waitForStatus() consists of a busy-loop which polls the DEA status port using *deaDevice*.isReplyReady(). If the loop's count expires before the status is returned, waitForStatus() returns an error. The busy-loop count is chosen to account for the execution time to read a value plus the transmission time of the value from the DEA to the BEP. Assume for this scenario that the reply arrives before the polling loop's count expires.
12. getStatus() then reads the status port using *deaDevice*.readReply(), passing the value back to its caller.
13. Once readData() has read and stored the RAM word into the destination buffer, readRam() releases the symbolic DEA lock using *lock*.release().
14. readRam() then checks how words it has written since pausing to allow other tasks to run. If the number of words read exceeds the class-specific limit (defined by **DeaManager**::DEATIME_RAM_ITERATIONS), readRam(), using its saved task pointer, tells the current task to sleep for a few tenths of a second, using *client*->sleep(). readRam()'s loop then repeats from step 4 until all of the data words have been read from the DEA board's RAM.

26.5.3 Use 3: Start the CCD-controller sequencers

Figure 107 illustrates the sequence of actions when a client instructs the **DeaManager** to start the CCD sequencers.

FIGURE 107. Start Sequencer Scenario

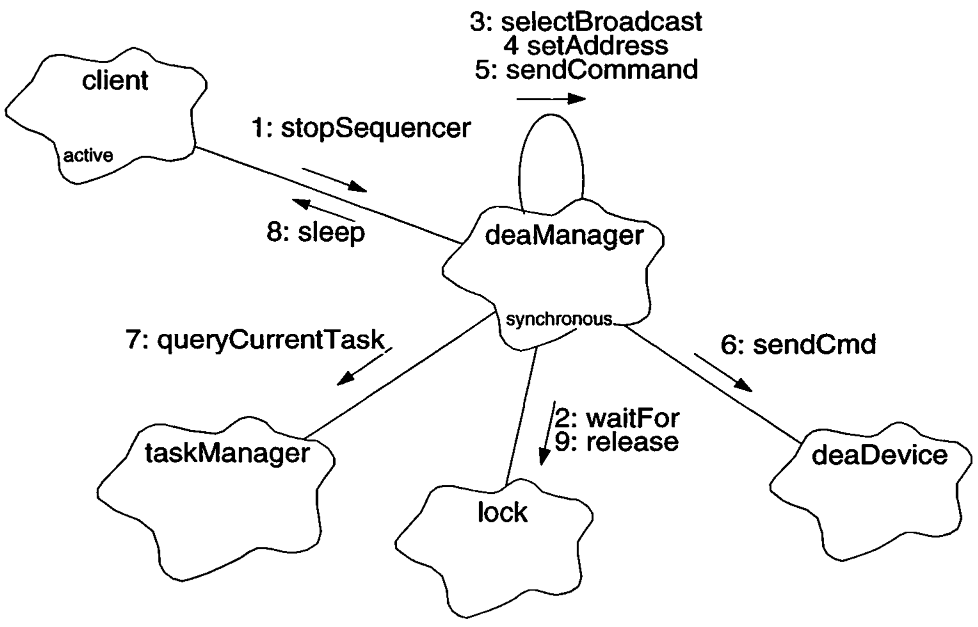


1. The `client` issues a command to start the DEA CCD sequencers, using *deaManager*.`invokeSequencer()`.
2. `invokeSequencer()` obtains exclusive access to the DEA interface, using *lock*.`waitFor()`.
3. `invokeSequencer()` enables all DEA CCD-controller boards to listen for commands using `selectBroadcast()`.
4. `invokeSequencer()` selects the sequencer control register on all of the enabled boards using `setAddress()`.
5. `invokeSequencer()` then forms and writes a sequencer enable word to the selected register on all of the enabled boards using `sendCommand()`. The sequencers will start clocking within 10us after the command is transmitted to the DEA.
6. `invokeSequencer()` obtains a pointer to the currently running task using *taskManager*.`queryCurrentTask()`.
7. `invokeSequencer()` instructs the current task to sleep for a tenth of a second to ensure that the latched time-stamp is stable, using *client*->`sleep()`.
8. `invokeSequencer()` then obtains a copy of the latched time-stamp using *deaDevice*.`readTimestamp()`.
9. `invokeSequencer()` releases the lock, using *lock*.`release()` and passed the read time-stamp to the calling *client*.

26.5.4 Use 4: Stop the CCD-controller sequencers

Figure 108 illustrates the sequence of actions when a client instructs the **DeaManager** to halt the CCD sequencers.

FIGURE 108. Stop Sequencer Scenario

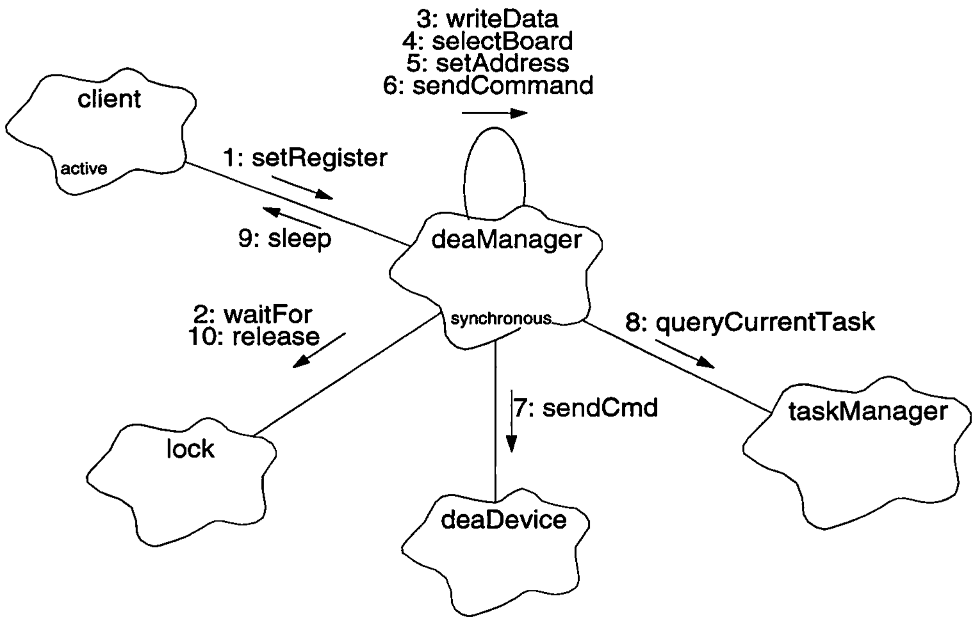


1. The `client` issues a command to halt the DEA CCD sequencers, using *deaManager*.`stopSequencer()`.
2. `stopSequencer()` obtains exclusive access to the DEA interface, using *lock*.`waitFor()`.
3. `stopSequencer()` enables all DEA CCD-controller boards to listen for commands using `selectBroadcast()`.
4. `stopSequencer()` selects the sequencer control register on all of the enabled boards using `setAddress()`.
5. `stopSequencer()` then forms and writes a sequencer disable word to the selected register on all of the enabled boards using `sendCommand()`. The sequencers will stop clocking once the command is transmitted to the DEA.
6. After ensuring that the command port is ready, `sendCommand()` issues the command to the DEA boards using *deaDevice*.`sendCmd()`.
7. `stopSequencer()` obtains a pointer to the currently running task using *taskManager*.`queryCurrentTask()`.
8. `stopSequencer()` instructs the current task to sleep for a tenth of a second to ensure that the command has been executed by the DEA, using *client*->`sleep()`.
9. `stopSequencer()` releases the lock, using *lock*.`release()` and returns to the calling *client*.

26.5.5 Use 5: Set DEA Register

Figure 109 illustrates the sequence of actions when a client instructs the **DeaManager** to write a value to one of the DEA board registers.

FIGURE 109. Set DEA Register Scenario

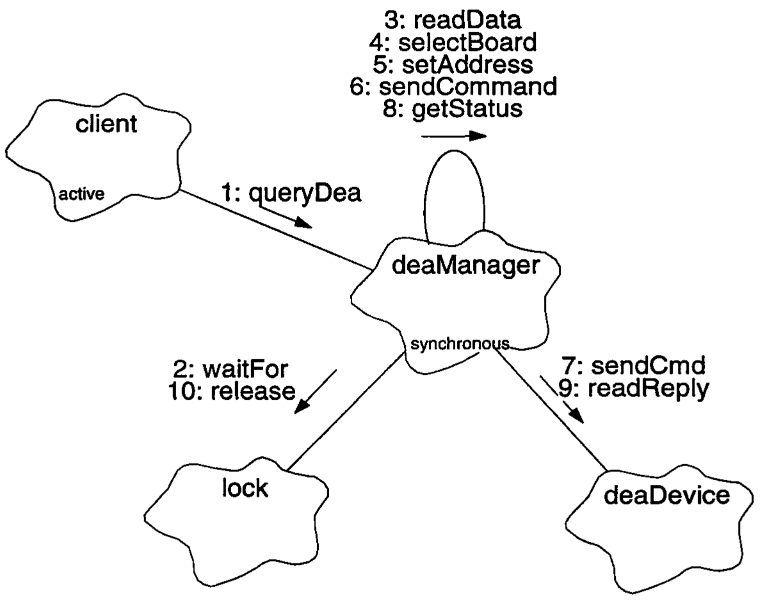


1. The *client* issues a command to write a value to one of the DEA boards, using *deaManager*.setRegister().
2. setRegister() obtains exclusive access to the DEA interface, using *lock*.waitFor(). It then checks if the board has power (hasPower() not shown), and if not, it releases the lock and returns an error.
3. setRegister() writes the data to the selected board register by calling writeData().
4. writeData() selects the desired DEA board using selectBoard().
5. writeData() selects the desired register using setAddress().
6. writeData() issues the command to write the data to the register using sendCommand().
7. After waiting for the command port to become available, sendCommand() transmits the command to the DEA using *deaDevice*.sendCmd().
8. Once the command has been issued, setRegister() gets a pointer to the currently running task using *taskManager*.queryCurrentTask().
9. setRegister() tells the task to sleep for several tenths of a second to ensure that the DEA has time to act on the register command using *client*->sleep().
10. setRegister() releases the lock using *lock*.release(), and returns to the calling *client*.

26.5.6 Use 6: Read DEA Register

Figure 110 illustrates the sequence of actions when a client instructs the **DeaManager** to read a value from one of the DEA board registers.

FIGURE 110. Query DEA Register Scenario

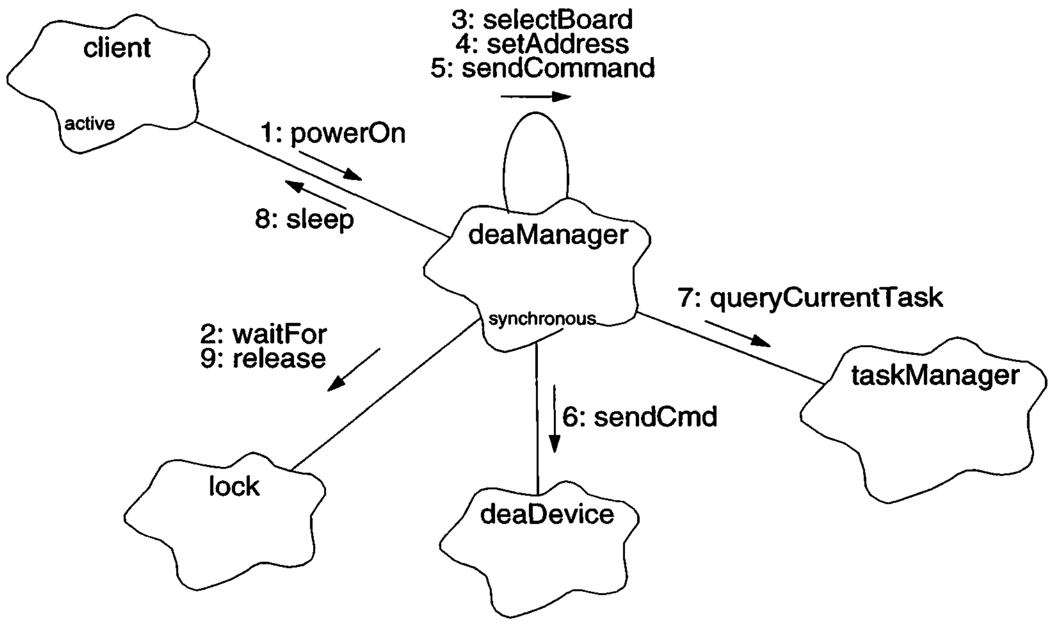


1. The `client` issues a command to read value from one of the DEA boards, using *deaManager*.`queryDea()`.
2. `queryDea()` obtains exclusive access to the DEA interface, using *lock*.`waitFor()`. It then checks if the board has power (`hasPower()` not shown), and if not, it releases the lock and returns an error.
3. `queryDea()` reads the value from the desired board and register calling `readData()`.
4. `readData()` selects the desired DEA board using `selectBoard()`.
5. `readData()` selects the desired register using `setAddress()`.
6. `readData()` issues the command to read the data from the register using `sendCommand()`.
7. After waiting for the command port to become available, `sendCommand()` transmits the command to the DEA using *deaDevice*.`sendCmd()`.
8. `readData()` then gets the response to the query using `getStatus()`.
9. `getStatus()` waits for the reply to be received by the Back End, and reads the value using *deaDevice*.`readReply()`.
10. `setRegister()` releases the lock using *lock*.`release()`, and returns to the calling *client*.

26.5.7 Use 7: Enable and disable power to a single DEA board

Figure 111 illustrates the sequence of actions when a client instructs the **DeaManager** to power on a single DEA CCD-controller board. The steps involved in powering off a single board are similar, except that a power-off command is sent to the DEA Interface board. TBD: How to detect and update system configurations on power-on?

FIGURE 111. CCD-Controller Power On Scenario



1. The `client` issues a command to enable power to one of the DEA CCD-controller boards, using *deaManager*.`powerOn()`.
2. `powerOn()` obtains exclusive access to the DEA interface using *lock*.`waitFor()`.
3. `powerOn()` issues a command to select the DEA Interface board on the DEA using `selectBoard()`.
4. `powerOn()` issues a command to select the power-control register on the DEA Interface board using `setAddress()`.
5. `powerOn()` adds the selected board to a mask of boards which have power, and issues a command to write the mask to the power control register using `sendCommand()`.
6. `sendCommand()` waits for the command port to complete its previous transfer, and then issues the command to the DEA using *deaDevice*.`sendCmd()`.
7. `powerOn()` obtains a pointer to the currently running task using *taskManager*.`queryCurrentTask()`.
8. `powerOn()` instructs the task to sleep for 1 second (TBD) using *client*->`sleep()`.
9. `powerOn()` releases its lock on the DEA interface using *lock*.`release()` and returns to the calling `client`.

26.6 Class DeaManager

Documentation:

The **DeaManager** class performs high level input/output operations to the Detector Electronics Assembly. It is capable of loading sequencer memory, setting register levels, and querying housekeeping ports.

Export Control:	Public
Cardinality:	n

Hierarchy:

Superclasses:	**none**

Implementation Uses:

DeaDevice *deaDevice*
TaskManager *taskManager*
Task

Public Interface:

Operations:

```
DeaManager()
hasPower()
invokeSequencer()
loadSequencers()
powerOff()
powerOn()
queryDea()
readPram()
readSram()
setRegister()
stopSequencer()
writePram()
writeSram()
```

Protected Interface:

Operations:	`getStatus()`
	`loadRam()`
	`readData()`
	`readRam()`
	`selectBoard()`
	`selectBroadcast()`
	`sendCommand()`
	`setAddress()`
	`waitForPort()`
	`waitForStatus()`
	`writeData()`

Private Interface:

Timing Constants:	DEATIME_RAM_ITERATIONS = 1000 per sleep
	DEATIME_RAM_SLEEP = 2 ticks (0.2 seconds)
	DEATIME_LOCK_WAIT = 5 ticks (0.5 seconds)
	DEATIME_REG_SLEEP = 2 ticks (0.2 seconds)
	DEATIME_SEQ_SLEEP = 2 ticks (0.2 seconds)
	DEATIME_CMD_ITERATIONS = 100 (~50us)
	DEATIME_CMD_WAITLOOP = 500 (~250us)
	DEATIME_STAT_WAITLOOP = 1500 (~750us)
	DEATIME_POWER_ON_SLEEP = 10 (1 second)
	DEATIME_POWER_OFF_SLEEP = 10 (1 second)

Has-A Relationships:

`unsigned` *`curboard`*: This state variable indicates which DEA board was last written to and selected to send responses. If broadcast mode was last selected, or the last select command failed, this variable contains the value DEAID_UNKNOWN = 15.

`Boolean` *`sequencerActive`*: This instance variable indicates whether or not the sequencers have been commanded to run or not.

`Semaphore` *`lock`*: This is used to symbolically indicate exclusive access to the DEA interface.

`unsigned` *`powermask`*: This variable maintains a list of currently powered DEA Boards. Bit 0 corresponds to board 0, bit 1 to board 1, etc. If a bit is set to 1, a command has been sent to turn the power on the board. If a bit is set to 0, the board should be off.

Concurrency:	Synchronous
Persistence:	Persistent

26.6.1 DeaManager()

Public member of: **DeaManager**

Arguments:

unsigned *semid*

Documentation:

This is the constructor for the DEA Manager. *semid* is the Nucleus RTX semaphore id to use for the class's *lock* variable. The function initializes *lock*, sets *curboard* to DEAID_UNKNOWN, deasserts *sequencerActive*, and zeroes *powermask*. The body of the constructor calls *deaDevice*.reset() to reset the DEA interface board, and disable power to the DEA's CCD controller boards.

Concurrency: Sequential

26.6.2 getStatus()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

unsigned& *status*

Documentation:

This function busy-waits until a status is received from the DEA, and stores the read word into the status parameter. If the status word is read successfully, the function returns *BoolTrue*. If the wait expires before a status word is ready, the function returns *BoolFalse*.

Semantics:

Call waitForStatus() to synch. with the DEA. Then call *deaDevice*.readReply() to read the status word. If waitForStatus() times out, return *BoolFalse*, otherwise return *BoolTrue*.

Concurrency: Guarded

26.6.3 hasPower()

Public member of:	**DeaManager**
Return Class:	**Boolean**
Arguments:	
	DeaBoardId *boardid*

Documentation:

This function returns whether or not the DEA Manager has enabled power to the board, indicated by *boardid*, by examining the private *powermask* variable. If the board has been enabled, the function returns *BoolTrue*. If not, it returns *BoolFalse*.

Concurrency:	Guarded

26.6.4 invokeSequencer()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

unsigned& *starttime*

Documentation:

This function starts the DEA CCD Controller Sequencers. *starttime* is a copy of the science timestamp, latched when the command to start was issued. If the command is sent successfully, the function returns *BoolTrue*. If an error occurs, it returns *BoolFalse*.

Semantics:

Obtain exclusive access to the DEA interface using *lock*.waitFor(). Select all CCD-controllers as listeners by calling selectBroadcast(). Select the sequencer control register using setAddress() and form and issue the sequencer start command using sendCommand(). Obtain the calling task using *taskManager*.queryCurrentTask() and sleep for DEATIME_SEQ_SLEEP timer ticks (0.1 second) using *curtask*->sleep() to allow the command to complete execution. Read the latched time-stamp using *deaDevice*.readTimestamp() and place the result into *starttime*. Assert the *sequencerActive* flag, and release access to the DEA interface using *lock*.release().

Concurrency: Guarded

26.6.5 loadRam()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*
unsigned *addr*
const unsigned short* *srcbuf*
unsigned *count*

Documentation:

This function loads *count* words pointed to by *srcbuf* into the location specified by *addr* on the board specified by *boardid*. If the load succeeds, the function returns *BoolTrue*, otherwise it returns *BoolFalse*.

Semantics:

Call *taskManager*.queryCurrentTask() to get pointer to calling task. Set iteration counter to DEATIME_RAM_ITERATIONS and enter loop, which terminates when there are no more words to store, or on error. On each iteration, call *lock*.waitFor() to obtain exclusive access to the DEA. Then check *sequencerActive*, and generate an error if the sequencers are running. Call hasPower() to make sure board was turned on, and generate error if not. Call writeData() to store the word into DEA RAM and release the lock using *lock*.release() once the function returns. Then check the iteration counter and call *curtask*->sleep() and reset the counter if it has expired, otherwise, decrement the counter.

Concurrency: Guarded

26.6.6 loadSequencers()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

const DeaSequenceLoad& *image*

Documentation:

This function loads the sequencer images, specified by *image*, into the indicated CCD controller boards. If the load is successful, the function returns *BoolTrue*. If the load fails, the function returns *BoolFalse*.

Semantics:

Iterate through each section contained within *image*. On each section iteration, iterate through each selected controller board. If a board is selected, use `writePram()` to load PRAM sections into the boards, and `writeSram()` to load SRAM sections.

Concurrency: Guarded

26.6.7 powerOff()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*

Documentation:

This function issues a command to turn off power to one of the DEA's CCD Controller boards. The function returns *BoolTrue* if successful, and *BoolFalse* if the command transmission fails.

Semantics:

Lock access to the DEA using *lock*.waitFor(). Call selectBoard() to select the DEA's interface board, and call setAddress() to select the power control register on that board. Clear the bit in *powermask* corresponding to *boardid*, and write *powermask* to the selected control register. Once the command has been issued, get a pointer to the current task using *taskManager*.queryCurrentTask(), and sleep for DEATIME_POWER_OFF_SLEEP using *curtask*->sleep(). Once the power-off time has elapsed, release control the DEA interface using *lock*.release().

Concurrency: Guarded

26.6.8 powerOn()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*

Documentation:

This function issues a command to enable power to the CCD controller board indicated by *boardid*. If the command is sent successfully, the function returns *BoolTrue*. If the command fails, it returns *BoolFalse*.

Semantics:

Lock access to the DEA using *lock*.`waitFor()`. Call `selectBoard()` to select the DEA's interface board, and call `setAddress()` to select the power control register on that board. Set the bit in *powermask* corresponding to *boardid*, and write *powermask* to the selected control register. Once the command has been issued, get a pointer to the current task using *taskManager*.`queryCurrentTask()`, and sleep for DEATIME_POWER_ON_SLEEP using *curtask*->`sleep()`. Once the power-on time has elapsed, release control the DEA interface using *lock*.`release()`.

Concurrency: Guarded

26.6.9 queryDea()

Public member of:	**DeaManager**
Return Class:	**Boolean**

Arguments:

DeaBoardId *boardid*
unsigned *queryid*
unsigned& *value*

Documentation:

This function queries the housekeeping port or PRAM location, addressed by *queryid*, on the DEA board indicated by *boardid*, and places the returned item into *value*. If the query command and response is successful, the function returns *BoolTrue*, otherwise it returns *BoolFalse*.

Semantics:

Lock access to the DEA using *lock*.waitFor(). Call hasPower() to ensure that the desired board has been powered on. If the board is not on, return *BoolFalse*. Convert the *queryid* from a register index into a DEA address, and pass it, and the reference to *value* to readData(), which reads the requested register and places the result into *value*. Once the requested register has been read, release control the DEA interface using *lock*.release().

Concurrency: Guarded

26.6.10 readData()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*
unsigned *addr*
unsigned& *value*

Documentation:

This function reads the field located at addr on DEA Board *boardid* and stores the result into value. If successful, the routine returns *BoolTrue*. If the read fails, it returns *BoolFalse*.

Semantics:

Call selectBoard() to select the board to read from. Call setAddress() to select the address to read. Call sendCommand() to issue the read command. Call getStatus() to read the response to the query.

Concurrency: Guarded

26.6.11 readPram()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *pramid*
unsigned *index*
unsigned short* *dstbuf*
unsigned *valcnt*

Documentation:

This function reads *valcnt* words of PRAM from the board specified by *pramid*, and starting from the location indicated by *index* into the array pointed to by *dstbuf*. If successful, the function returns *BoolTrue*. If a query fails, the function returns *BoolFalse*. This function adjusts index to the corresponding DEA address and calls readRam() to perform the read.

Concurrency: Guarded

26.6.12 readRam()

Protected member of:	**DeaManager**
Return Class:	**Boolean**
Arguments:	**DeaBoardId** *boardid* **unsigned** *addr* **unsigned short*** *dstbuf* **unsigned** *count*

Documentation:

This function reads *count* words from the DEA RAM located at *addr* on board *boardid* into the buffer *dstbuf*. If successful, the function returns *BoolTrue*, otherwise it returns *BoolFalse*.

Semantics:

Call *taskManager*.queryCurrentTask() to get pointer to calling task. Set iteration counter to DEATIME_RAM_ITERATIONS and enter loop, which terminates when there are no more words to read, or on error. On each iteration, call *lock*.waitFor() to obtain exclusive access to the DEA. Then check *sequencerActive*, and generate an error if the sequencers are running. Call hasPower() to make sure board was turned on, and generate error if not. Call readData() to fetch the word from DEA RAM. Release the lock using *lock*.release() once the function returns. Store the returned value into the destination buffer. Then check the iteration counter and call *curtask*->sleep() and reset the counter if it has expired, otherwise, decrement the counter.

Concurrency: Guarded

26.6.13 readSram()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *sramid*
unsigned *index*
unsigned short* *dstbuf*
unsigned *valcnt*

Documentation:

This function reads `valcnt` words of DRAM from the board specified by *sramid*, and starting from the location indicated by `index` into the array pointed to by *dstbuf*. If successful, the function returns *BoolTrue*. If a query fails, the function returns *BoolFalse*. This function adjusts index to the corresponding DEA address and calls `readRam()` to perform the read.

Concurrency: Guarded

26.6.14 selectBoard()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*

Documentation:

This function selects *boardid* as the current board being written to and being queried. If the specified board has already been selected, or if the selection command succeeds, the function returns *BoolTrue*. Otherwise, the function returns *BoolFalse*.

Semantics:

Compare *boardid* to *curboard*. If different, form command, set the write mask bit corresponding to *boardid*, and the read select field to the passed *boardid* and issue the command using `sendCommand()`. If `sendCommand()` succeeds, copy *boardid* to *curboard* and return *BoolTrue*. If it fails, set *curboard* to DEAID_UNKNOWN and return *BoolFalse*.

Concurrency: Guarded

26.6.15 selectBroadcast()

Protected member of: **DeaManager**

Return Class: **Boolean**

Documentation:

> This function issues a command which selects all boards to listen to the next command. If the command is successful, the function returns *BoolTrue*, otherwise, it returns *BoolFalse*.

Semantics:

> Set *curboard* to *powermask*, enabling all powered on boards as listeners, form command and set the read select to board DEAID_UNKNOWN. Issue the command using sendCommand(). If sendCommand() succeeds, return *BoolTrue*. If it fails, return *BoolFalse*.

Concurrency: Guarded

26.6.16 sendCommand()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

unsigned *command*

Documentation:

> This function sends a command word, command, to the DEA. If the word is sent, the function returns *BoolTrue*. If the wait for the command port expires, the function returns *BoolFalse*.

Semantics:

> Call waitForPort() to wait until the command port is ready. Then execute a small loop DEATIME_CMD_INTERATIONS times to ensure a delay between the previous command and the next. Then pass *command* to *deaDevice*.sendCmd(). If waitForPort() fails, return *BoolFalse*, otherwise return *BoolTrue*.

Concurrency: Guarded

26.6.17 setAddress()

Protected member of: **`DeaManager`**

Return Class: **`Boolean`**

Arguments:

`unsigned` *`address`*

Documentation:

This function issues a command to copy *`address`* into current address register on the listening DEA board. If the command succeeds, the function returns *`BoolTrue`*. On error, it returns *`BoolFalse`*.

Semantics:

Form and issues the write address command using `sendCommand()`. If the command is sent successfully, the function returns *`BoolTrue`*. If the command fails, it returns *`BoolFalse`*.

Concurrency: Guarded

26.6.18 setRegister()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*
unsigned *regaddr*
unsigned *regval*

Documentation:

This function writes *regval* to the register located on the DEA board, *boardid*, at the register address, *regaddr*. This function returns *BoolTrue* if the command is successfully sent, and *BoolFalse* if it fails to send the value. After issuing the command, the function sleeps the calling task for at least 0.2 seconds to allow the command to be executed.

Semantics:

Obtain exclusive access to the DEA interface using *lock*.waitFor(). Check to ensure that the board has power, using hasPower(). Adjust the *regaddr* to map to the DEA register addresses and call writeData() to write the value to the specified board register. If it succeeds, get the current task, using *taskManager*.queryCurrentTask(), and sleep for DEATIME_REG_SLEEP (~0.2 seconds) to allow the command to be executed, release the lock using *lock*.release() and return *BoolTrue*. If the it fails, return *BoolFalse*.

Concurrency: Guarded

26.6.19 stopSequencer()

Public member of:	**DeaManager**
Return Class:	**Boolean**

Documentation:

This function issues a command to stop the DEA CCD Controller Sequencers. If the command is successful, the function returns *BoolTrue*. If an error is detected, the function returns *BoolFalse*.

Semantics:

Obtain exclusive access to the DEA interface using *lock*.waitFor(). Select all CCD-controllers as listeners by calling selectBroadcast(). Select the sequencer control register using setAddress() and form and issue the stop command using sendCommand(). Obtain the calling task using *taskManager*.queryCurrentTask() and sleep for DEATIME_SEQ_SLEEP timer ticks (0.1 second) using *curtask*->sleep() to allow the command to complete execution. Clear the *sequencerActive* flag, and release access to the DEA interface using *lock*.release().

Concurrency: Guarded

26.6.20 waitForPort()

Protected member of: **DeaManager**

Return Class: **Boolean**

Documentation:

This function busy waits until the DEA serial command port is ready to send a word of data. If the wait time exceeds the serial transfer time (i.e. about 24us), the function returns *BoolFalse*, otherwise, it returns *BoolTrue*.

Semantics:

Iterate no more than DEATIME_CMD_WAITLOOP times, calling *deaDevice*.isCmdPortReady(). If the call returns *BoolTrue*, the port is ready for another command and return *BoolTrue*. Otherwise, continue iterating. If the loop iterations complete, the command port timed-out, and return *BoolFalse*.

Concurrency: Guarded

26.6.21 waitForStatus()

Protected member of: **DeaManager**

Return Class: **Boolean**

Documentation:

This function busy-waits until a status word is received by the DEA status port. If a status word is ready, the function returns *BoolTrue*. If the wait expires without a status word being received, the function returns *BoolFalse*.

Semantics:

Iterate DEATIME_STAT_WAITLOOP times or until *deaDevice*.isReplyReady() returns *BoolTrue*. If the device indicates a reply is ready, return *BoolTrue*. If the loop completes without a status word available, return *BoolFalse*.

Concurrency: Guarded

26.6.22 writeData()

Protected member of:	**DeaManager**
Return Class:	**Boolean**
Arguments:	**DeaBoardId** *boardid* **unsigned** *addr* **unsigned** *value*

Documentation:

This function writes `value` to the register located on the DEA board, *boardid*, at the register address, *addr*. This function returns *BoolTrue* if the command is successfully sent, and *BoolFalse* if it fails to send the value.

Semantics:

Call `selectBoard()` to ensure the appropriate board has been enabled. Then call `setAddress()` to select the register to write to, Then form and issue the write command using `sendCommand()` to send the value to the selected board. If all of the steps succeed, return *BoolTrue*, otherwise return *BoolFalse*.

Concurrency: Guarded

26.6.23 writePram()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *pramid*
unsigned *index*
const unsigned short* *srcbuf*
unsigned *valcnt*

Documentation:

This function writes *valcnt* words of PRAM from the source buffer pointed to by *srcbuf*, into the board specified by *pramid*, and starting at the location indicated by *index* into the array pointed to by *dstbuf*. If successful, the function returns *BoolTrue*. If a write fails, the function returns *BoolFalse*. This function maps *index* to the appropriate DEA address and uses `loadRam()` to perform the load.

Concurrency: Guarded

26.6.24 writeSram()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *sramid*
unsigned *index*
const unsigned short* *srcbuf*
unsigned *valcnt*

Documentation:

This function writes `valcnt` words of SRAM from the source buffer pointed to by *srcbuf*, into the board specified by *sramid*, and starting at the location indicated by *index* into the array pointed to by *dstbuf*. If successful, the function returns *BoolTrue*. If a write fails, the function returns *BoolFalse*. This function maps *index* to the appropriate DEA address and uses `loadRam()` to perform the load.

Concurrency: Guarded