CSR

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

REVISION LOG	**TITLE:** Software Detailed Design FEP and FEP Interrupt Devices	**DOC. NO.** 36-53210

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
—	3/27/95	—	—	initial version for code walkthrough	
A	5/4/95	36-234	all	Incorperate review comments	5/22/95

10.0 FEP Devices and FEP Interrupt Device (36-53210 A)

10.1 Purpose

The purpose of the Front End Processor (FEP) Device and Front End Processor Interrupt Device classes are to provide access to Back End's Front End Processor Interface logic and interrupt.

10.2 Uses

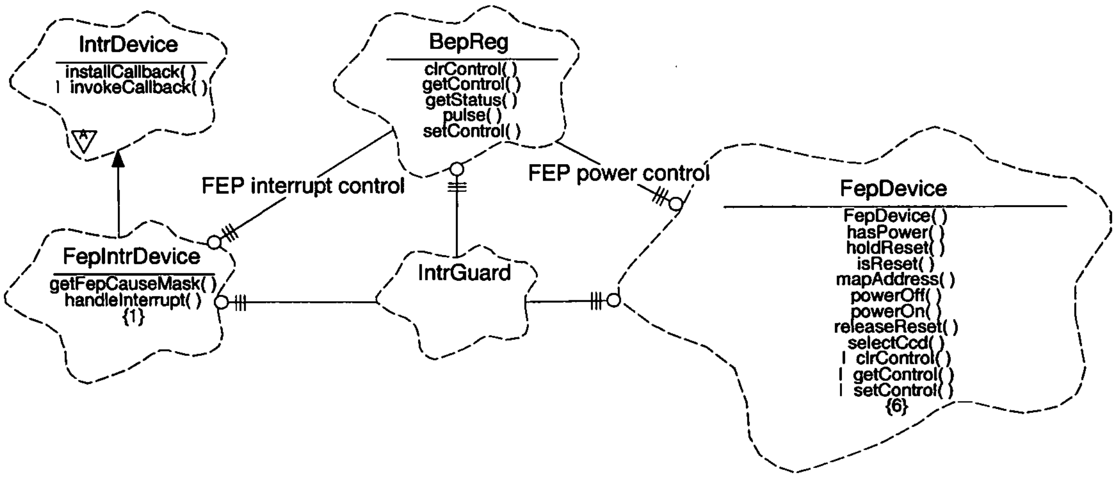
The FEP Device class, `FepDevice`, and FEP Interrupt class, `FepIntrDevice`, provide the following features:

Use 1:: Handle the interrupt caused by one or more of the active FEPs
Use 2:: Manage the reset lines of each FEP
Use 3:: Map FEP shared memory addresses into BEP address space
Use 4:: Enable and disable power to each of the FEPs
Use 5:: Select which CCD's data is processed by each FEP

10.3 Organization

The Back End Processor manages the Front End Processor hardware using 6 FEP Device class instances, and 1 FEP Interrupt Device class instance. Figure 21 illustrates the relationships used by these classes.

FIGURE 21. FEP Device and Interrupt Device Classes



`FepIntrDevice` - This class is a subclass of `IntrDevice` and is responsible for handling interrupts from one or more of the Front End Processors (`handleInterrupt`()), and for providing client access to which Front End Processor's are requesting service (`getFepCauseMask`()). In order to allow client callback functions to process Front End interrupts in tight loops, there is one FEP interrupt (and class instance) on the Back End Processor. This class uses the **BepReg** class to access the Front End Processor interrupt cause and reset bits (**BepReg**::getStatus(), **BepReg**::pulse()) This class also uses the **IntrGuard** class to temporarily disable interrupts during read/modify/writes of shared registers or instance variables.

FepDevice - This class is responsible for managing the Back End hardware interface logic to Front End Processors. There is one instance of this class for each Front End Processor within ACIS. During start-up, each **FepDevice** instance is associated with one of the Front End Processors by passing to its constructor the corresponding Front End Processor Identifier (**FepId**). From then on, that instance is responsible for managing that FEP's hardware interface (other than interrupts). This class provides functions to map Front End Processor shared memory addresses to the Back End Processor address space (mapAddress), to control and query the state of a Front End Processor's reset line (isReset, holdReset, releaseReset), to query and control the power to a Front End Processor (hasPower, powerOff, powerOn), and to select from which CCD to acquire data (selectCcd). Internally, this class uses the **BepReg** class to gain access to the FEP power control bits (**BegReg**::getControl(), **BepReg**::setControl(), and **BepReg**::clrControl()). This class uses three similar functions, getControl() setControl() and clrControl(), to manage the BEP to FEP Control Register residing on each Front End Processor. It uses **IntrGuard** to disable interrupts when performing read/modify/write operations on these registers.

NOTE: In order to simplify access to these devices, the system provides an array of pointers to these devices, *fepDevice*[], which is indexed by the Front End Processor identifier.

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 **FepIntrDevice**, may use their parent's protected method, invokeCallback(), to invoke the installed callback instance. See Section 6.0 for more detail.

BepReg- This class represents the lowest level hardware access to the features provided by the Back End hardware control, status, and pulse registers. See Section 5.0 for more detail.

IntrGuard - This class is responsible for temporarily disabling interrupts. Its constructor saves the current interrupt state, and disables interrupts. Its destructor restores the previously saved interrupt state. For more detail, see Section 6.0 .

10.4 BEP to FEP Hardware Interface Overview

On ACIS, the active Back End Processor (BEP) is responsible for managing 6 Front End Processors (FEP). Each of these 6 FEPs is responsible for processing science data from one CCD. The following describe certain portions of the Back End/Front End hardware interfaces. For more detail, see Section 4.9 . For details on the software protocols used between the FEPs and the BEP, see Section 4.10 .

10.4.1 Power Control

The Back End Processor is responsible for turning on and off the power to each of the FEPs. The BEP software uses 6 FEP power control bits in the Back End's Control Register, one bit for each FEP. When a bit is set to 1 in the control register, the power to the corresponding FEP is turned on. When the bit is 0, that FEP is powered off.

10.4.2 FEP to BEP Interrupt

Each FEP can cause an interrupt on the Back End Processor. The FEP interrupt lines are combined together to form a single FEP to BEP interrupt. The Back End's Status Register contains 6 bits which each contain a latched version of a given FEP to BEP interrupt line. If any of these bits is set, then a FEP to BEP interrupt is generated on the Back End. The Back End can individually reset each of these latches with a corresponding bit in the Back End's pulse register.

10.4.3 Shared Memory

A portion of each FEP's memory, including their respective reset vectors and microboot control words, appears in the Back End's address space, and can be read from and written to directly by the Back End. When the Back End needs to start a FEP, it holds the FEP's reset line, loads any needed code and data into the FEP via the shared memory interface (including the code at the FEPs reset vector and microboot control word) and releases the FEP's reset line. The FEP will then use the microboot control word to setup its cache address mapping, and start executing the loaded code starting from its reset vector. Once running, the Back End can communicate with each of the FEPs through their respective shared memory (using an agreed upon software interface, see the FEP Manager, Section 17.0 , and the interface descriptions in Section 4.9 and Section 4.10).

10.4.4 FEP Reset Control and Watchdog Timer

Each Front End Processor contains a BEP to FEP control register. This register is writable by the Back End via the shared memory interface, and can only be read by software running on the FEP. This register contains a reset control bit, and a separate reset status bit. The Back End can control the state of the FEP's reset line using the control bit. The current state of the FEP's reset line is indicated by a status bit in this register.

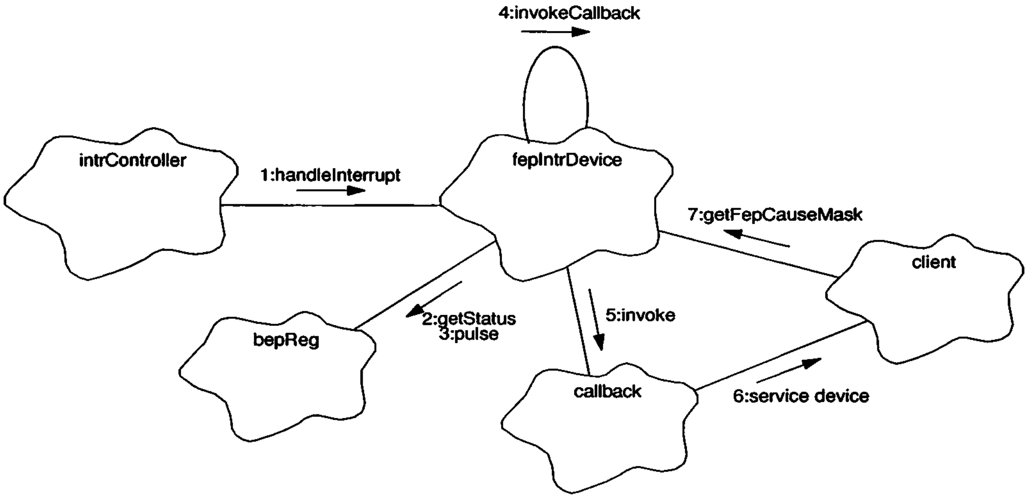
In addition to being controlled by the Back End, a given FEP's reset line is also under the control of the FEPs watchdog timer logic. If a given FEP's watchdog timer expires, the FEP's reset line is asserted and held, and a FEP to BEP interrupt is generated. The FEP's reset line remains asserted until explicitly released by the Back End. This allows the Back End to detect crashes occurring on the FEP and take appropriate action.

10.5 Scenarios

10.5.1 Use1: Handle the interrupt caused by one or more of the active FEPs

Figure 22 illustrates the handling of an FEP interrupt.

FIGURE 22. Handle FEP Interrupt

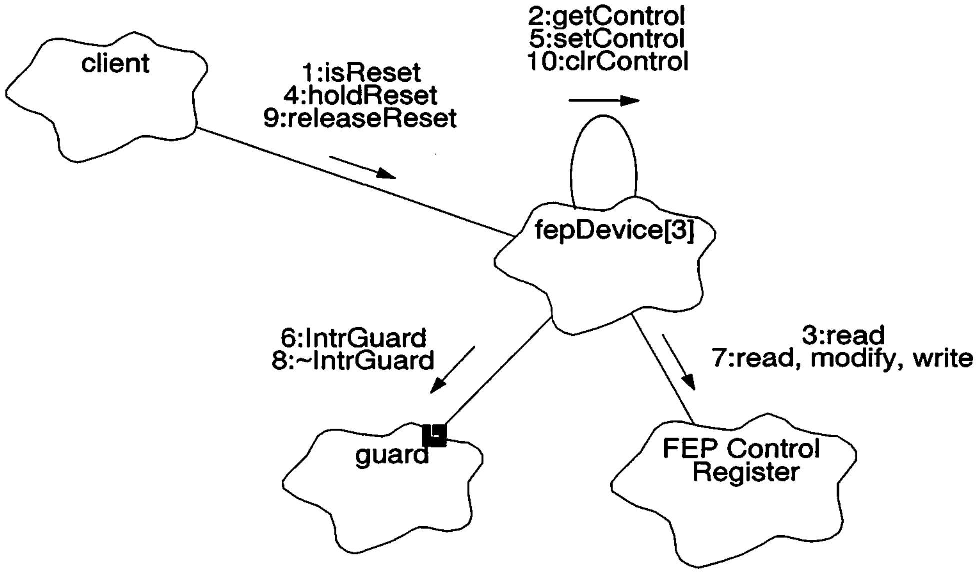


1. One or more of the Front End Processors generate a Back End Interrupt. The interrupt controller instance determines the cause of the interrupt, enables higher priority interrupts (not shown) and invokes the FEP Interrupt Device instance to deal with the interrupt, using *fepIntrDevice*.handleInterrupt().
2. handleInterrupt() uses *bepReg*.getStatus() to read the Back End's status register. It masks and shifts the returned value to get a bit-field containing which FEPs are currently requesting interrupt service from the BEP. handleInterrupt() saves the mask in its private *saveMask* instance variable.
3. It then clears the latched FEP interrupt cause bits using *bepReg*.pulse(). At this point, *saveMask* contains the list of FEPs which will be serviced by this interrupt invocation. The handling of subsequent FEP interrupt requests will be deferred until after this handler invocation returns.
4. handleInterrupt() calls its inherited function, **IntrDevice**::invokeCallback() to pass control to the installed FEP Interrupt callback instance.
5. invokeCallback() then invokes the installed callback instance's invoke() function.
6. *callback*.invoke() then passes control to a client object (service device).
7. The client then obtains the mask of FEPs requesting service, using *fepIntrDevice*.getFepCauseMask(), which returns *saveMask*, and proceeds to respond to each of the requesting FEP's needs.

10.5.2 Use 2: Manage the reset lines of each FEP

Figure 23 illustrates queries, assertions and de-assertions of an FEP reset line.

FIGURE 23. FEP Reset line query, assertion and de-assertion



1. The *client* queries FEP number 3's reset state, using *fepDevice*[3]->isReset().
2. *fepDevice*[3]->isReset() calls *fepDevice*[3]->getControl() and tests the returned word's reset status bit to determine if the Front Processor is reset. isReset() then returns *BoolTrue* if the processor is in a reset state, or *BoolFalse* if it is not.
3. *fepDevice*[3]->getControl() reads the associated FEP's control register from the shared memory interface.
4. The client calls *fepDevice*[3]->holdReset() to place the FEP into a reset state.
5. *fepDevice*[3]->holdReset() calls its private function setControl() to assert the reset bit in the FEP control register.
6. setControl() first disables interrupts by constructing an **IntrGuard** instance, *guard*.
7. setControl() reads the FEP control register, ORs in the reset bit, and writes it back to the memory-mapped register.
8. setControl() then returns, leaving the scope which created *guard*. *guard*'s destructor (~IntrGuard()) then restores the previous interrupt state.
9. The client calls *fepDevice*[3]->releaseReset() to restart the FEP.
10. releaseReset() calls the internal function, clrControl(). clrControl() then uses **IntrGuard** to disable interrupts, reads the control register, clears its reset bit, writes the result back to the control register, and returns, restoring the previous interrupt state (not illustrated).

10.5.3 Use 3: Map FEP shared memory addresses into BEP address space

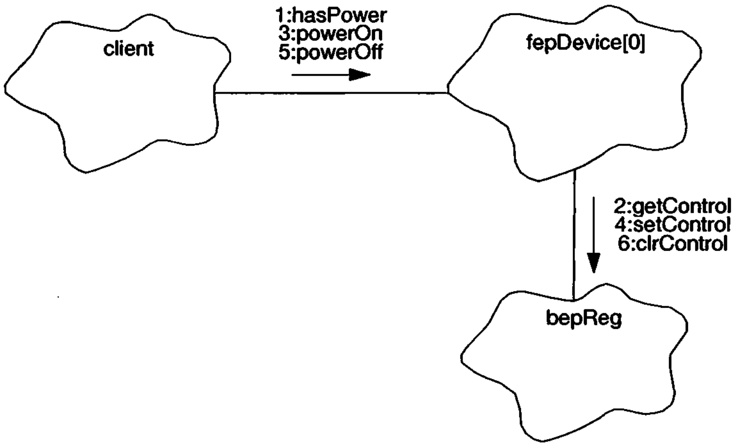
After start-up, when an `FepDevice` instance is constructed, it is permanently mapped to a particular Front End Processor. When a client needs to map an address, specified in terms of the Front End Processor's virtual address space, into the corresponding shared memory address in the Back End Processor, it calls the particular FEP's `mapAddress()` member function, passing the starting address and maximum number of words to be accessed. If the specified starting address and range is within a FEP's shared memory space, `mapAddress()` returns the corresponding address in the Back End's virtual address space. If the starting address or ending address is not within the shared memory address space, `mapAddress()` returns 0.

NOTE: The BEP shared memory address space assigned to a particular FEP does not necessarily map to contiguous FEP memory blocks.

10.5.4 Use 4: Enable and disable power to each of the FEPs

Figure 24 illustrates power queries, and power on, and power off commands to a Front End Processor.

FIGURE 24. Querying FEP power, and turning an FEP on and off



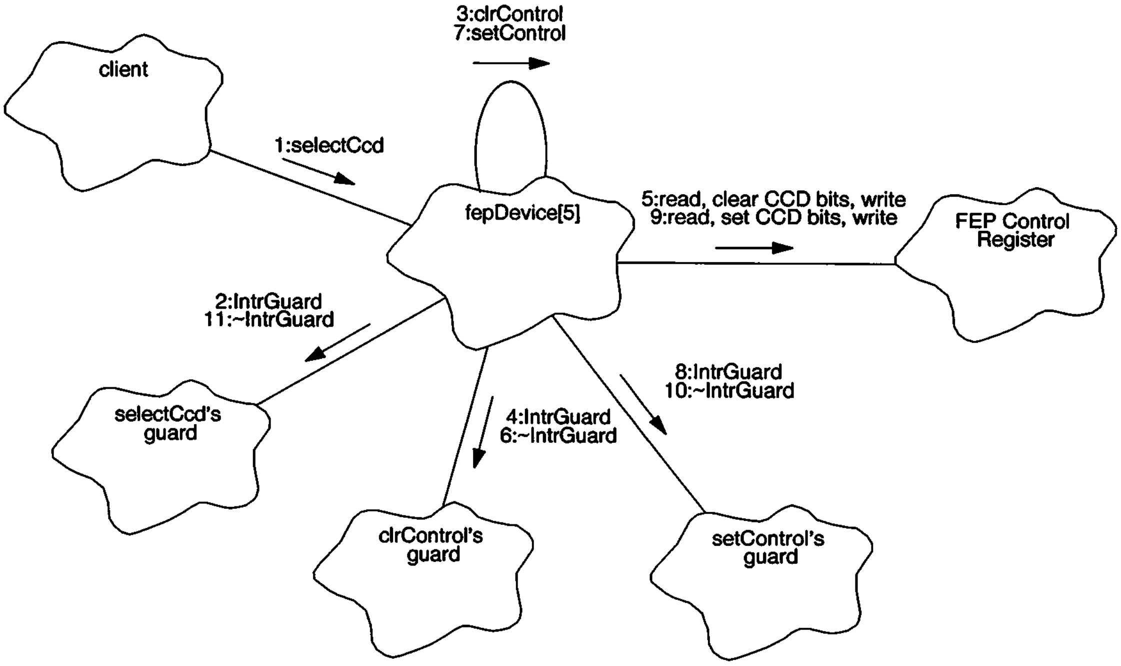
1. The client queries the current state of FEP 0's power, using *`fepDevice`*[0]->`hasPower()`.
2. `hasPower()` calls *`bepReg`*.`getControl()` to read the current state of the Back End's Control register. If the power bit corresponding to FEP 0 is set, `hasPower()` returns *`BoolTrue`*, indicating the FEP is on. Otherwise, it returns *`BoolFalse`*, indicating that FEP 0 is currently off.
3. The client calls *`fepDevice`*[0].`powerOn()` to turn on FEP 0.
4. `powerOn()` calls *`bepReg`*.`setControl()` to assert the power control bit for FEP 0, which causes power to be supplied to FEP 0.
5. The client calls *`fepDevice`*[0]->`powerOff()` to turn off power to FEP 0.

6. powerOff() calls *bepReg*.clrControl() to clear the power bit for FEP 0. Once this bit is cleared, FEP 0 is no longer powered on (NOTE: Any state information maintained in FEP 0, including its pixel bias map values, is now lost).

10.5.5 Use 5: Select CCD to process

Figure 25 illustrates how a client configures a FEP to process data from a particular CCD.

FIGURE 25. Assigning CCD Selection



1. The client tells FEP 5 to accept data from CCD I2 (Imaging CCD 2) only, by calling *fepDevice*[5].selectCcd().
2. selectCcd() temporarily disables interrupts by declaring an **IntrGuard** instance, *guard*.
3. selectCcd() then uses clrControl() to zero all of the CCD selection bits in the FEP's control register.
4. clrControl() declares another local **IntrGuard** instance, which saves the already disabled interrupt state.
5. clrControl() then reads the FEP Control Register, clears the CCD selection bits, and writes the result back to the control register.
6. clrControl() returns, and its local guard's destructor, ~IntrGuard(), is invoked. The destructor then restores the disabled interrupt state (i.e. interrupts remain disabled).
7. selectCcd() shifts the passed CCD identifier to the appropriate bit position within the control register, and uses setControl() to set the 1's contained in CCD selection into the control register (NOTE: clrControl() has already cleared the others).

8. setControl() declares another local **IntrGuard** instance, which also saves the already disabled interrupt state.
9. setControl() then reads the FEP Control Register, sets the CCD selection bits, and writes the result back to the control register.
10. setControl() returns, causing ~IntrGuard() to restore the disabled interrupt state.
11. selectCcd() returns, causing its ~IntrGuard() to restore the interrupt state that was active when the *client* first invoked selectCcd().

10.6 Class FepIntrDevice

Documentation:

This class is responsible for handling the FEPs to BEP interrupt. Since any active FEP can cause this single interrupt, **`FepIntrDevice`** provides the capability to query which FEP(s) caused a particular interrupt.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`IntrDevice`**

Implementation Uses:

`BepReg`
`IntrGuard`

Public Interface:

Operations:

`getFepCauseMask()`
`handleInterrupt()`

Private Interface:

Has-A Relationships:

`Boolean` *`inHandler`*: This variable indicates whether or not we are currently processing a FEP interrupt. If *`inHandler`* is *`BoolFalse`*, then the device is not inside its `handleInterrupt`() routine. If *`inHandler`* is *`BoolTrue`*, then `handleInterrupt`() has been called and is in the process of dealing with an FEP interrupt.

`unsigned` *`saveMask`*: This contains the FEP interrupt mask, saved prior to resetting the FEP interrupt cause bits by `handleInterrupt`(). This mask indicates which FEPs were requesting service at the time the interrupt handler was invoked (and which cause bits were subsequently cleared by the handler).

Concurrency: Synchronous

Persistence: Persistent

10.6.1 getFepCauseMask()

Public member of: `FepIntrDevice`

Return Class: `unsigned`

Documentation:

This function returns a value indicating which Front End Processors are requesting interrupt service. The least significant 6-bits in the return value each correspond to one FEP, where bit 0 corresponds to FEP_0, bit 1 to FEP_1 and so on. The remaining bits in the returned value are unused, and will be set to 0. If a particular FEP bit is 1, the corresponding FEP has asserted the FEP to BEP interrupt request. If a bit is 0, the corresponding FEP was not requesting service (or was not handling an FEP interrupt) at the time the interrupt was called.

Semantics:

If in the process of handling an interrupt, *inHandler* is *BoolTrue*, return the value saved when the handler was invoked, *saveMask* (i.e. prior to when the interrupts were reset). If not in interrupt processing, obtain the value directly from the FEP Status register.

Concurrency: Synchronous

10.6.2 handleInterrupt()

Public member of: `FepIntrDevice`

Return Class: `void`

Documentation:

This function handles interrupts from one or more of the Front End Processors. This function clears the interrupt cause and invokes the installed callback function, using the inherited `IntrDevice::invokeCallback()`.

Semantics:

Get the FEP interrupt cause mask, using *bepReg*.`getStatus()`. Shift, mask and store in *saveMask*. Set *inHandler* to *BoolTrue*. Clear the FEP interrupt causes (via *bepReg*.`pulse()`). Invoke the callback, `invokeCallback()`. Set *inHandler* to *BoolFalse* and return.

Concurrency: Synchronous

10.7 Class FepDevice

Documentation:

This class provides an interface to allow the Back End Processor access to the hardware for a single Front End Processor.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **none**

Implementation Uses:

BepReg
IntrGuard

Public Interface:

Operations:

```
FepDevice()
hasPower()
holdReset()
isReset()
mapAddress()
powerOff()
powerOn()
releaseReset()
selectCcd()
```

Protected Interface:

Has-A Relationships:

const FepId *fepId*:: This represents which Front End Processor is associated with this instance, set when the instance is constructed.

volatile unsigned* *sharedBase*::This contains the base address in shared memory for this FEP instance.

Operations:

```
clrControl()
getControl()
setControl()
```

Concurrency: Synchronous

10.7.1 FepDevice()

Public member of: **`FepDevice`**

Arguments:

`FepId` *`fepid`*

Documentation:

This function initializes the state of the FEP device and uses *`fepid`* to relate the constructed instance with a particular physical Front End Processor. It initializes *`fepId`* to the passed *`fepid`*. The body of the constructor uses *`fepId`* to select which shared memory base address to use, and stores the selected base address in *`sharedBase`*.

Concurrency: Sequential

10.7.2 clrControl()

Protected member of: **FepDevice**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function clears the bits, designated by 1's in the mask argument, in the Front End Processor Control Register associated with this **FepDevice** instance. 0's in *mask* have no effect.

Preconditions:

For sensible results, the FEP must be powered on.

Semantics:

Declare **IntrGuard** instance, *guard*, to save interrupt state and disable interrupts. Read the BEP to FEP control register from shared memory (*sharedBase*[CTL_OFFSET]), bitwise-AND the inverse of *mask* to clear the specified bits, and write the result back out to the control register. Once this function returns, *guard*'s destructor will be invoked, and will restore the previous interrupt state.

Concurrency: Synchronous

10.7.3 getControl()

Protected member of: **`FepDevice`**

Return Class: **`unsigned`**

Documentation:

This function returns the current contents of the Front End Processor's (the one associated with this instance) control register.

Preconditions:

For sensible results, the FEP must be powered on.

Semantics:

Read and return the FEP's control register (*`sharedBase`*[CTL_OFFSET]) from shared memory.

Concurrency: Synchronous

10.7.4 hasPower()

Public member of: **`FepDevice`**

Return Class: **`Boolean`**

Documentation:

This function tests to see if the Front End is turned on. It returns *`BoolTrue`* if so, and *`BoolFalse`* if the FEP is powered off.

Semantics:

Use *`bepReg`*`.getControl()` to read the current value of the BEP control register. Test the bit corresponding to the associated FEP. If 1, return *`BoolTrue`*, if 0, return *`BoolFalse`*.

Concurrency: Synchronous

10.7.5 holdReset()

Public member of: `FepDevice`

Return Class: `void`

Documentation:

Assert and hold the Front End Processor's reset line, using `FepDevice::setControl()`.

Preconditions:

For sensible results, the FEP must be powered on.

Concurrency: Synchronous

10.7.6 isReset()

Public member of: `FepDevice`

Return Class: `Boolean`

Documentation:

This function tests the FEP reset line. It returns *BoolTrue* if the FEP reset line is asserted, and *BoolFalse* if the FEP is running.

Preconditions:

For sensible results, the FEP must be powered on.

Semantics:

Use `FepDevice::getControl()` to read the FEP's control register, and tests the reset status bit. If 1, return *BoolTrue*, else return *BoolFalse*.

Concurrency: Sequential

10.7.7 mapAddress()

Public member of: `FepDevice`

Return Class: `volatile unsigned*`

Arguments:

`volatile const unsigned*` *fepaddr*
`unsigned` *wordcnt*

Documentation:

This function tests to see if the FEP buffer pointed to by *fepaddr* (in FEP address space), and whose size is *wordcnt*, is accessible via the shared-memory interface to the FEP. If so, it returns the corresponding shared memory address. If not, it returns 0.

Semantics:

Verify that *fepaddr* is greater than or equal to the start of a FEP's shared memory address space, and that *fepaddr* + *wordcnt* - 1 is within the last FEP shared memory address slot. If the address range is legal, return *sharedBase* + *fepaddr*. Otherwise, return 0.

NOTE: Once the shared memory addresses and ranges are defined, this implementation may change.

Postconditions:

To produce sensible results, the client is responsible for ensuring that the FEP is powered on before reading or writing to the returned address.

Concurrency: Synchronous

10.7.8 powerOff()

Public member of: `FepDevice`

Return Class: `void`

Documentation:

Disable power to the Front End Processor, using *bepReg*.`clrControl`().

Postconditions:

Power is removed from the FEP and any information stored in FEP memory is lost.

Concurrency: Synchronous

10.7.9 powerOn()

Public member of: `FepDevice`

Return Class: `void`

Documentation:

This function enables power to the Front End Processor, using *bepReg*.`setControl`().

Postconditions:

Power is applied to the FEP. If the power was off prior to this call, the FEP will come up in a reset state, and any information stored in FEP memory prior to this call is lost. If power was already applied to the FEP prior to this call, asserting this bit has no effect.

Concurrency: Synchronous

10.7.10 releaseReset()

Public member of: `FepDevice`

Return Class: `void`

Documentation:

De-assert the Front End reset line, using `FepDevice::clrControl()`, allowing the FEP to boot and run.

Preconditions:

The FEP must be powered on, and the BEP must have initialized the FEP's microboot control word and stored executable code at the FEPs reset vector location.

Postconditions:

If the FEP's reset line was asserted prior to this call, it will configure its memory according to the microboot control word, and start executing from its reset vector. If the FEP's reset line was not asserted prior to this call (i.e. the FEP is already running), de-asserting this bit has no effect.

Concurrency: Synchronous

10.7.11 selectCcd()

Public member of: `FepDevice`

Return Class: `void`

Arguments:

`CcdId` *`ccdid`*

Documentation:

This function selects which CCD is processed by the associated Front End Processor. *`ccdid`* identifies which CCD's data is processed by this FEP.

Preconditions:

For sensible results, the FEP must be powered on.

Semantics:

Disable interrupts by declaring **`IntrGuard`** instance, *`guard`*. Use `clrControl()` to zero all CCD bits in the FEP's control register. Then shift *`ccdid`* to the appropriate bit position and pass to `setControl()`, effectively storing *`ccdid`* without modifying other bits in the control register. Once this function returns, *`guard`*'s destructor will restore the previous interrupt state.

Postconditions:

This FEP will accept image data only from the CCD specified by *`ccdid`*.

Concurrency: Synchronous

10.7.12 setControl()

Protected member of: **FepDevice**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function sets the bits, designated by 1's in the *mask* argument, in the Front End Processor Control Register associated with this **FepDevice** instance. 0's in *mask* have no effect.

Preconditions:

For sensible results, the FEP must be powered on.

Semantics:

Declare **IntrGuard** instance, *guard*, to save interrupt state and disable interrupts. Read the BEP to FEP control register from shared memory (*sharedBase*[CTL_OFFSET]), bitwise-OR *mask* to clear the specified bits, and write the result back out to the control register. Once this function returns, *guard*'s destructor will be invoked, and will restore the previous interrupt state.

Concurrency: Synchronous