CSR

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

REVISION LOG

TITLE: Software Detailed Design
Mongoose and Back End Registers

DOC. NO.
36-53205 Rev. B

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
-	1/17/95	-	-	Design walkthrough	
A	5/1/95	36-216	all	Comment incorporation	
B	4/26/96	36-606	all	Added Leds and BootMode classes Design updates.	5/6/96

5.0 Mongoose and Back End Registers (36-53205 B)

5.1 Purpose

The purpose of the **Mongoose** and **BepReg** (Back End Registers) classes is to provide low-level access to the R3000 System Coprocessor (Coprocessor 0) Registers, the Mongoose Command/Status Interface (CSI) Registers, and the ACIS-specific Back End Processor Registers. See Section 4.1 , Section 4.2 , and Section 4.3 for descriptions of the R3000 core processor, the Mongoose extensions to the R3000 core, and the Back End Processor hardware, respectively.

In addition to these classes, this section also describes the **Leds** and **BootMode** utility classes. The **Leds** class is responsible for providing a layer of abstraction to the software-accessible discrete telemetry bits. The **BootMode** class is responsible for providing the cause of the most recent reset of the Back End Processor.

5.2 Uses

The **Mongoose** class provides the following features:

Use 1:: Read the contents of each of the R3000 System Coprocessor Registers
Use 2:: Write the contents of the R3000 System Coprocessor Status Register
Use 3:: Provide memory-mapped access to all of the Mongoose CSI Registers
Use 4:: Set and clear sets of bits in the Mongoose Configuration Register
Use 5:: Set the DMA mode value in the Mongoose Configuration Register
Use 6:: Set and clear sets of bits in the Mongoose Extended Interrupt Mask Register
Use 7:: Clear latched extended interrupts by writing to the Extended Cause Register
Use 8:: Test user addresses against Instruction Cache and Data Cache boundaries.
Use 9:: Copy information to and from the Instruction Cache

The **BepReg** class provides the following additional features:
Use 10:: Set and clear sets of bits in the Back End's Control Register
Use 11:: Write a value to the software discrete telemetry bits (LEDs) in the Back End's Control Register
Use 12:: Read the contents of the Back End's Status Register
Use 13:: Write sets of bits to the Back End's Pulse Register
Use 14:: Provide memory-mapped access all of the Back End Registers

The **Leds** class provides the following feature:
Use 15:: Set the software discrete telemetry (LED) values

The **BootMode** class provides the following feature:
Use 16:: Indicate the cause of the most recent reset of the Back End Processor

5.3 Organization

With the exception of the **IntrGuard** class (see Section 6.0), the **Mongoose** and **BepReg** classes are stand-alone classes, and do not rely on any higher level class definitions. The only relationship is that the **BepReg** and **Mongoose** class use the **IntrGuard** class to temporarily disable interrupts when performing atomic operations, such as a read-modify-write of a shared register. The **IntrGuard** class, then uses the **Mongoose** class to access the R3000 Status Register needed to perform the interrupt disable and enables. The **Leds** uses the **BepReg** class to set the discrete telemetry code and **BootMode** class uses the **BepReg** class to obtain the contents of the hardware Status Register.

5.4 Class Instances and Register Bit-field Definitions

5.4.1 Class Instances

Access to the Mongoose registers is provided by a global **Mongoose** class instance, named *mongoose* and the BEP register class is accessed via a single global instance *bepReg*.

Since the Mongoose CSI registers are packed into adjacent memory locations, and this will not change over the development cycle, the *mongoose* instance provides a pointer to a structure which directly overlays the CSI block. This provides the lowest level device code direct access to the Mongoose's device control registers.

The Back End Registers are currently NOT packed into adjacent memory addresses, and therefore, this approach is not appropriate. Instead, the Back End Registers module currently provides all of its functions as in-line functions to address of the various BEP registers and atomically manage the control, status and pulse registers.

5.4.2 Register Bit-field Definitions

The R3000 register and bit definitions can be found in the "MIPS Programmer's Handbook, Section A.4 Registers." The Mongoose register and bit definitions can be found in the "Mongoose ASIC Microcontroller Programming Guide, Section 9.0 Command/Status Interface (CSI) Registers." The Back End Processor registers are listed and described in the "DPA Hardware Specification and System Description" in the sections pertaining to the Back End Processors Overall Description and subsequent subsections.

Both the **Mongoose** and **BepReg** classes provide enumerations which define the various bits in each register they support. The bit-definition names are qualified by the name of the class defining them. This provides in-code documentation as to where to find the actual definition. For example, if the **Mongoose** class defines the R3000 status register bit, SR_IEC (Status register Interrupt Enable Current), users of this definition (i.e. all classes other than the **Mongoose** class) must refer to the bit by qualifying with the **Mongoose**

class definition, "**Mongoose**::SR_IEC." Rather than list the many definitions in this section, refer to Section 4.3 for a detailed description of these bits; the constants are defined in mongoose.H and bepreg.H.

5.5 Scenarios

5.5.1 Mongoose Class Scenarios

5.5.1.1 Use 1: Read the R3000 System Coprocessor Registers

The **Mongoose** class provides functions which read the R3000 System Coprocessor registers using the "mfc0" (Move from Coprocessor 0) assembler instruction (NOTE: The **Mongoose** class uses a small set of assembler functions to access these registers. These support functions are provided by filesstartup/asm_startup.S). This provides the client code with the contents of the following registers:

- Status Register (SR) - This register contains the key R3000 interrupt and processor control bits. This register is primarily used by the Interrupt Controller class (see Section 6.0), and by code which needs to perform atomic operations. It is accessed by the `getStatusReg()` member function.
- Cause Register (CAUSE) - This register contains information pertaining to the cause of an interrupt. This register is primarily used by the Interrupt Controller class. It is accessed by the `getCauseReg()` member function.
- Bad Virtual Address Register (BVADDR) - This register contains the address which caused the last address exception. It is accessed by the `getBadVaddrReg()` member function.
- Exception Program Counter Register (EPC) - This register contains the address to return to after handling the current exception or interrupt. This register is currently only used by the assembly language low-level interrupt handling code. It is accessed by the `getEpcReg()` member function.

5.5.1.2 Use 2: Write the R3000 Status Register

The R3000 Status Register controls interrupt enables, and masks individual interrupt lines on the R3000 core. This, in conjunction with the Mongoose's Extended Interrupt Mask register, is used to mask and unmask individual interrupt causes, as globally enable and disable interrupts. Client functions modify this register using the `setStatusReg()` member function.

In general, most low-level client functions want to protect themselves from interrupts while read-modify-writing a shared register. The following C++ code fragment illustrates a safe way of accomplishing this with the **Mongoose** class:

```
unsigned oldStatusRegister; // Saved contents of Status Register
// ---- Disable interrupts, saving previous contents ----
oldStatusRegister = mongoose->setStatusReg(Mongoose::SR_DISABLE_INTS);
// code to read/modify and write the shared register
// ---- Restore original Status Register contents ----
mongoose->setStatusReg (oldStatusRegister);
```

5.5.1.3 Use 3: Memory-map access to Mongoose Registers

Since the Mongoose CSI registers are memory-mapped into a contiguous block in memory, the **Mongoose** class provides access to these registers in the form of a pointer to a data structure. The registers are accessed by referring to the register by name through the *regs* register pointer, contained within the global **Mongoose** class instance *mongoose*. The following code fragment illustrates how a client might set the contents of the DMA destination address register:

```
mongoose.regs->dmadst = unsigned(dstptr);
```

5.5.1.4 Use 4: Set and clear bits in Configuration Register

The Mongoose Configuration Registers controls several different devices. As such, access to this register's bits must be shared by more than one client device. To make this sharing easier, the **Mongoose** class provides functions which atomically set (`setCfgBits()`) and clear (`clrCfgBits()`) bits in the register.

5.5.1.5 Use 5: Set the DMA mode value

This function, `setDmaMode()`, is a special case of "Use 4: Set and clear bits in Configuration Register." Since setting the DMA mode value in the Configuration involves both setting and clearing bits, this ability is rolled into a separate function specifically for dealing with the DMA mode value. Typically, the DMA code uses this function to start and stop DMA activity.

5.5.1.6 Use 6: Set and clear bits in Extended Interrupt Mask

Just as for the Configuration Register, the Mongoose Extended Interrupt Mask Register controls several different devices. As such, access to this register's bits must be shared by more than one client device. To make this sharing easier, the **Mongoose** class provides functions which atomically set and clear bits in the register, `setXmaskBits()` and `clrXmaskBits()` respectively.

5.5.1.7 Use 7: Clear latched extended interrupts

Extended interrupts on the Mongoose are latched. In order to clear a latch for a particular interrupt, one must write a 1 into the corresponding bit in the Extended Cause Register. The **Mongoose** class provides a function which provides this service, `clrXcauseBits()`.

5.5.1.8 Use 8: Test addresses against cache boundaries

Since the Mongoose Instruction Cache requires special access, and the Mongoose DMA controller only supports certain types of transfers, the **Mongoose** class provides func-

tions which test client addresses against the Instruction and Data cache address boundaries (`isIcache()`, `isDcache()` `isUncached()`).

5.5.1.9 Use 9: Copy to and from I-cache

In order to read and write information to and from the Mongoose Instruction cache, one must manipulate the Instruction Cache Address and Data registers. In order to keep block operations to and from I-cache reasonably fast, the **`Mongoose`** class provides block copy functions to and from I-cache (`icacheWrite()`, `icacheRead()`).

5.5.2 BepReg Class Scenarios

5.5.2.1 Use 10: Set and clear bits in BEP Control Register

The BEP's Control Register affects several different devices. As such, access to this register's bits must be shared by more than one client device. To make this sharing easier, the **`BepReg`** class provides functions which atomically set and clear bits in the register (`setControl()`, `clrControl()`).

5.5.2.2 Use 11: Write LEDs

This function is a special case of "Use 10: Set and clear bits in BEP Control Register." Since setting the LED value in the Control Registers involves both setting and clearing bits, this ability is rolled into a separate function specifically for dealing with the LED value (**`BepReg`**`::showLeds()`).

5.5.2.3 Use 12: Read BEP Status Register

In order to provide straight-forward access to the BEP's Status Register, the **`BepReg`** class defines a function which reads and returns the current value of the BEP Status Register (`getStatus()`).

5.5.2.4 Use 13: Pulse bits in BEP Pulse Register

In order to provide straight-forward access to the BEP's Pulse Register, the **`BepReg`** class defines a function which writes a caller-supplied value to the BEP Pulse Register (`pulse()`).

5.5.2.5 Use 14: Memory-map access to BEP Registers

Since most of the **`BepReg`** class functions require access to physical hardware locations, the **`BepReg`** class also provides functions which return type-safe pointers to all BEP registers. These include the following:

- Control Register (read/write)- Contains various device control bits (`ctlReg`)
- Status Register (read only)- This contains various device status bits (`statReg`)
- Pulse Register (write only) - This clears various latched device interrupts and controls (`pulseReg`)
- Command FIFO (read only) - This provides 16-bit command words from the RCTU (`cmdFifoReg`)
- DEA Command Register (write only) - This writes commands to the DEA (`deaCmdReg`)
- DEA Status Register (read only) - This reads status information from the DEA (`deaStatReg`)
- Downlink Controller Start Address (read/write) - This specifies the start address for a telemetry packet transfer (`dtcStartReg`).
- Downlink Controller End Address (read/write) - This specifies the ending address of a telemetry packet transfer (`dtcEndReg`).
- Downlink Controller Address Count (read only) - This specifies the current address of the ongoing telemetry packet transfer(dtcAddrCntReg).
- S/C Counter - Latched (read only) - This contains the timestamp of the last command sent to the DEA.
- S/C Counter - Running (read only) - This contains a running value of the S/C counter.

5.5.3 Leds and BootMode Class Scenarios

5.5.3.1 Use 15: Set the software discrete telemetry values

In order to set the software discrete telemetry (LED) bits to a particular pattern, the client passes the desired pattern to *`leds`*`.show()`, which forwards the passed value to *`bepReg`*`.showLeds()` (see Section 5.5.2.2).

(NOTE: This class allows the **`BepReg`** class to remain an internal member to the ***`Devices`*** class category. The goal is to have the **`BepReg`** class be used directly only by the various device classes.).

5.5.3.2 Use 16: Indicate the cause of the most recent reset

To determine if the most recent reset was caused by a power-on command, the client calls *`bootMode`*`.isPowerOn()`. `isPowerOn()` calls *`bepReg`*`.getStatus()` and tests the power-on status bit. If the most recent reset was caused by a power-on command, `isPowerOn()` returns *`BoolTrue`*. If not, it returns *`BoolFalse`*.

To determine if the most recent reset was caused by the watchdog timer, the client calls *`bootMode`*`.isWatchdog()`. `isWatchdog()` calls *`bepReg`*`.getStatus()` and tests

the watchdog-reset status bit. If the most recent reset was caused by a timeout of the watchdog timer, `isWatchdog()` returns *`BoolTrue`*. If not, it returns *`BoolFalse`*.

5.6 Class Mongoose

Documentation:

This class provides the lowest-level interface to the Mongoose Processor.

NOTE: Some member functions are sequential, therefore access to these functions must be coordinated between active threads to avoid contention. This drives the "Guarded" concurrency attribute listed below. Refer to the "Mongoose Programming Guide" and the "MIPS Programmer's Handbook" for detailed descriptions of the hardware registers provided by the Mongoose Microcontroller.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:

```
clrCfgBits()
clrXCauseBits()
clrXMaskBits()
delay()
getBadVaddrReg()
getCauseReg()
getEpcReg()
getStatusReg()
getXCauseReg()
icacheRead()
icacheWrite()
isDcache()
isIcache()
isUncached()
setCfgBits()
setDmaMode()
setStatusReg()
setXMaskBits()
```

Concurrency: Guarded

Persistence: Persistent

5.6.1 clrCfgBits()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function clears bits, designated by 1's in the *mask* argument, in the Configuration Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.6.2 clrXCauseBits()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *xcause*

Documentation:

This function writes *xcause* to the Mongoose Extended Interrupt Cause Register. This has the effect of clearing any extended interrupts corresponding to 1's in the *xcause* argument. Interrupts corresponding to 0's are not affected.

Concurrency: Synchronous

5.6.3 clrXMaskBits()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function clears the bits, indicated by 1's in the *mask* argument, in the Mongoose's Extended Interrupt Mask Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.6.4 delay()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *useconds*

Documentation:

This function consists of a busy-loop which iterates for at least *useconds* microseconds. The loop executes without disabling interrupts or executive context switching, so the actual delay introduced by calling this function may be longer.

Preconditions:

useconds must be <= $(2^{32} - 1)$/ITERATIONS_PER_USEC where ITERATIONS_PER_USEC is 5 (TBD)

Concurrency: Synchronous

5.6.5 getBadVaddrReg()

Public member of: **Mongoose**

Return Class: **void***

Documentation:

This function returns the contents of the R3000 Coprocessor 0's Bad Virtual Address Register, using `asm_getBadVaddrReg()`.

Concurrency: Synchronous

5.6.6 getCauseReg()

Public member of: **Mongoose**

Return Class: **unsigned**

Documentation:

This function returns the contents of the R3000 Coprocessor 0's Interrupt Cause Register, using `asm_getCauseReg()`.

Concurrency: Synchronous

5.6.7 getEpcReg()

Public member of: **Mongoose**

Return Class: **unsigned***

Documentation:

This function returns the contents of the R3000 Coprocessor 0's Exception Program Counter Register, using `asm_getEpcReg()`.

NOTE: The Exception Program Counter is overwritten by nested interrupts, hence the Sequential concurrency qualifier.

Concurrency: Sequential

5.6.8 getStatusReg()

Public member of: **Mongoose**

Return Class: **unsigned**

Documentation:

Read and return the contents of the R3000 Coprocessor 0's Status Register, using `asm_getStatusReg()`.

Concurrency: Synchronous

5.6.9 getXCauseReg()

Public member of: **Mongoose**

Return Class: **unsigned**

Documentation:

This function returns the contents of the Mongoose Extended Cause Register.

Concurrency: Synchronous

5.6.10 icacheRead()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

const unsigned* *srcaddr*
unsigned* *dstaddr*
unsigned *wordcnt*

Documentation:

Copy *wordcnt* 32-bit words from the Mongoose Instruction cache, located at *srcaddr* into the buffer pointed to by *dstaddr*.

NOTE: This function does not disable interrupts while reading and writing the I-cache address and data registers, hence the "Sequential" concurrency qualifier.

Concurrency: Sequential

5.6.11 icacheWrite()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

const unsigned* *srcaddr*
unsigned* *dstaddr*
unsigned *wordcnt*

Documentation:

This function writes *wordcnt* words from the data address *srcaddr* to *dstaddr* within Instruction cache.

NOTE: This function does not disable interrupts while reading and writing the I-cache address and data registers, hence the "Sequential" concurrency qualifier.

Concurrency: Sequential

5.6.12 isDcache()

Public member of: **Mongoose**

Return Class: **Boolean**

Arguments:

```
const void* addr
```

Documentation:

This function tests the passed pointer, *addr*, against the boundaries of the Data Cache.The function returns *BoolTrue* if the pointer is within D-cache and returns *BoolFalse* if it is not.

5.6.13 isIcache()

Public member of: **Mongoose**

Return Class: **Boolean**

Arguments:

```
const void* addr
```

Documentation:

This function tests the passed pointer, *addr*, against the address boundaries of the Instruction Cache. If the pointer is within I-cache, the function returns *BoolTrue*, else it returns *BoolFalse*.

5.6.14 isUncached()

Public member of: **Mongoose**

Return Class: **Boolean**

Arguments:

```
const void* addr
```

Documentation:

This function tests the passed pointer, *addr*, against the start of non-cached memory. It returns *BoolTrue* if the pointer is beyond cache address space, else *BoolFalse*.

5.6.15 setCfgBits()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function sets bits, designated by 1's in the *mask* argument, in the Mongoose Configuration Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.6.16 setDmaMode()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *mode*

Documentation:

This function writes the mode value, specified by *mode*, into the Mongoose Configuration Register. Only the DMA Mode bits are affected.

Concurrency: Synchronous

5.6.17 setStatusReg()

Public member of: **Mongoose**

Return Class: **unsigned**

Arguments:

unsigned *value*

Documentation:

This function writes *value* to the contents of the R3000 Coprocessor 0's Status Register, and returns the old contents of the register, using `asm_setStatusReg()`

Concurrency: Synchronous

5.6.18 setXMaskBits()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function sets the bits, indicated by 1's in the *mask* argument, in the Mongooses Extended Interrupt Mask Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.7 Class BepReg

Documentation:

This class provides the lowest-level interface to the Back End Processor registers.

NOTE: Some member functions are sequential, therefore access to these functions must be coordinated between active threads to avoid contention. This drives the "Guarded" concurrency attribute listed below.

Refer to the "DPA Functional Description and Requirements" for detailed descriptions of the hardware registers provided by the Back End Processor.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:

```
clrControl()
cmdFifoReg()
ctlReg()
dtcAddrCntReg()
dtcEndReg()
dtcStartReg()
deaCmdReg()
deaStatReg()
getControl()
getStatus()
pulseReg()
pulse()
scCntLatTimeReg()
scCntRunTimeReg()
setControl()
showLeds()
statReg()
```

Concurrency: Guarded

Persistence: Persistent

5.7.1 clrControl()

Public member of: **BepReg**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function clears bits, designated by 1's in the *mask* argument, in the BEP Control Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.7.2 cmdFifoReg()

Public member of: **BepReg**

Return Class: **volatile const unsigned short***

Documentation:

This function returns a pointer to the BEP's Command FIFO. The FIFO is 16-bits wide. 32-bit fetches yield the next value in the FIFO in the least-significant 16-bits and garbage in the upper 16-bits. The pointer itself never changes. The "const" directive indicates that the register cannot be modified by the software. Since the hardware changes the contents of the register with each read, a "volatile" directive is needed to ensure that the compiler's code-optimizer does not remove code which reads from the FIFO (for example, by caching the read value in a CPU register).

Concurrency: Synchronous

5.7.3 ctlReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Control Register.

Concurrency: Synchronous

5.7.4 dtcAddrCntReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Downlink Controller Address Count Register.

Concurrency: Synchronous

5.7.5 dtcEndReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Downlink Controller End Address Register.

Concurrency: Synchronous

5.7.6 dtcStartReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Downlink Controller Start Address Register.

Concurrency: Synchronous

5.7.7 deaCmdReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Detector Electronics Assembly Command Register.

Concurrency: Synchronous

5.7.8 deaStatReg()

Public member of: **BepReg**

Return Class: **volatile const unsigned***

Documentation:

This function returns the constant address of the read-only ("const" directive) BEP Detector Electronics Assembly Status Register. The value of this register can change without being written to, hence the "volatile" directive.

Concurrency: Synchronous

5.7.9 getControl()

Public member of: **BepReg**

Return Class: **unsigned**

Documentation:

This function returns the current value contained in the BEP's Control Register.

Concurrency: Synchronous

5.7.10 getStatus()

Public member of: **BepReg**

Return Class: **unsigned**

Documentation:

This function returns the current value contained in the BEP's Status Register.

Concurrency: Synchronous

5.7.11 pulseReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Pulse Register.

Concurrency: Synchronous

5.7.12 pulse()

Public member of: **BepReg**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function writes *mask* into the BEP's Pulse Register. This has the effect of pulsing the bits indicated by 1's in the *mask* argument. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.7.13 scCntLatTimeReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the S/C Latched Counter.

Concurrency: Synchronous

5.7.14 scCntRunTimeReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the S/C Running Counter.

Concurrency: Synchronous

5.7.15 setControl()

Public member of: **BepReg**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function sets bits, designated by 1's in the *mask* argument, in the BEP Control Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.7.16 showLeds()

Public member of: **BepReg**

Return Class: **void**

Arguments:

unsigned *value*

Documentation:

This function sets the LED bits in the BEP's Control Register to the argument *value*. The least-significant four bits of *value* correspond to the four LED bits in the BEP Control Register. All other bits in *value* are ignored. Only the LED bits in the Control Register are modified by this function.

Concurrency: Synchronous

5.7.17 statReg()

Public member of: **BepReg**

Return Class: **volatile const unsigned***

Documentation:

This function returns the constant address of the read-only ("const" directive) BEP Status Register. The value of this register can change without being written to, hence the "volatile" directive.

Concurrency: Synchronous

5.8 Class Leds

Documentation:

This class is responsible for writing LED values to the software discrete telemetry bi-levels.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

BepReg

Public Interface:

Operations: `show()`

Concurrency: Synchronous

Persistence: Transient

5.8.1 show()

Public member of: **Leds**

Return Class: **void**

Arguments:

unsigned *value*

Documentation:

This function writes value to the LED discrete telemetry bi-levels, by passing *value* to *bepReg*.showLeds(). *value* must range from 0 to 15, inclusive.

Concurrency: Synchronous

5.9 Class BootMode

Documentation:

This class is responsible for determining the type of boot the BEP came up from.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

BepReg

Public Interface:

Operations:
```
isPowerOn()
isWatchdog()
```

Concurrency: Sequential

Persistence: Transient

5.9.1 isPowerOn()

Public member of: **BootMode**

Return Class: **Boolean**

Documentation:

This function determines if the BEP was reset due to a power-on reset. If so, the function returns *BoolTrue*, otherwise, it returns *BoolFalse*.

Semantics:

Call *bepReg*.getStatus() and test for a commanded or watchdog reset. If neither is set, then the BEP came up due to a power-on, and return *BoolTrue*. If either the Command Reset or Watchdog Reset status bits are set, then return *BoolFalse*.

Concurrency: Synchronous

5.9.2 isWatchdog()

Public member of: **BootMode**

Return Class: **Boolean**

Documentation:

This function determines if the BEP was last reset due to the watchdog timer. It returns *BoolTrue* if so, and *BoolFalse* if the watchdog did not cause the last reset.

Semantics:

Call *bepReg*.getStatus() and test the Watchdog reset status bit. If set, then the BEP was reset by the Watchdog timer. Return *BoolTrue*. If the Watchdog reset status bit is not set, then return *BoolFalse*.

Concurrency: Synchronous