CSR	MASSACHUSETTS INSTITUTE OF TECHNOLOGY CENTER FOR SPACE RESEARCH CAMBRIDGE, MASSACHUSETTS 02139	
REVISION LOG	**TITLE: Software Detailed Design Interrupt Control Classes**	**DOC. NO. 36-53206 Rev. A**

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
01	5/4/95	36-236	all	Initial version. Incorporated comments from initial review.	
A	4/23/96	36-602	all	Initial controlled release. Removed mention of FEP and DEA as sources of interrupts.	[illegible] 5/23/96

6.0 Interrupt Control Classes (36-53206 A)

6.1 Purpose

The purpose of the collection of interrupt control classes is to manage interrupts from the various Back End Processor devices. For a description of the R3000 and Mongoose interrupt structure and registers, see Section 4.1 and Section 4.2 . For a list of Back End Processor interrupt causes, priorities and their timing requirements, see Section 4.3.13 .

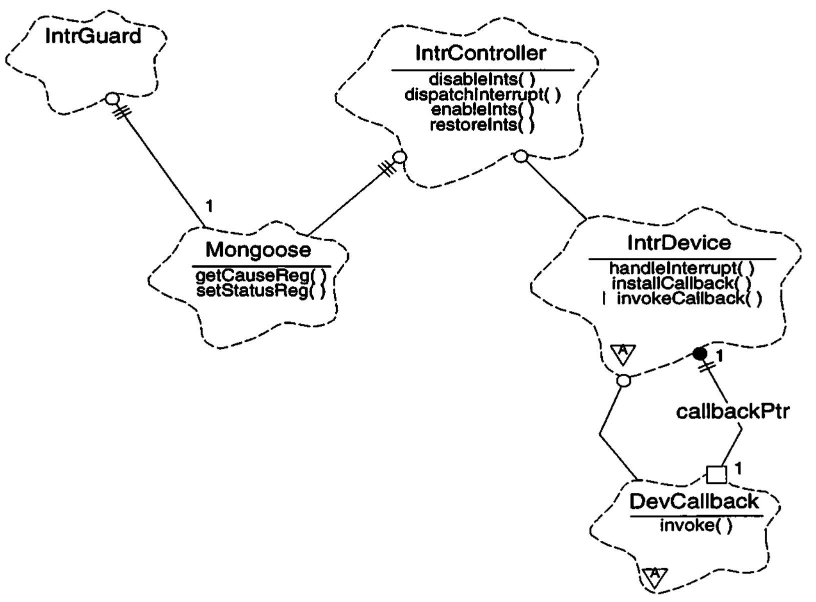
6.2 Uses

Use 1:: Dispatch control to a device when the associated hardware interrupt is asserted
Use 2:: Provide the ability to disable interrupts when executing critical sections of code
Use 3:: Allow clients of interruptible devices to obtain control during interrupt processing

6.3 Organization

This collection of classes consists of (a) an **`IntrController`** class, which manages prioritized interrupts and dispatches control to the interrupting device, (b) an **`IntrGuard`** class, whose construction and destruction delimit periods during which interrupts are disabled, (c) an abstract **`IntrDevice`** class, which generalizes the common interface to all interrupting devices, and (d) an abstract **`DevCallback`** class, which provides the common interface to all classes which can be installed to be "called-back" during interrupt processing. The **`IntrController`** class is a top level class, and uses the **`Mongoose`** class to provide access to the R3000 and Mongoose interrupt control and status registers. Not shown in the figure is the low-level R3000 assembly language code which passes control to the **`IntrController`** during interrupt processing.

FIGURE 12. Interrupt Controller and Device Relationships



IntrController - The **IntrController** class is responsible for top-level interrupt handling (dispatchInterrupt), and for enabling (enableInts), disabling (disableInts) interrupts, and for restoring a previous interrupt-enable state (restoreInts). It uses the **Mongoose** class to manipulate the R3000 Co-processor 0's Status Register (setStatusReg), read Co-processor 0's Cause Register (getCauseReg), and to read and write to the Mongoose Control/Status Interface's (CSI) Extended Mask and Cause registers (*mongoose.regs->xmask* and *mongoose.regs->xcause*, not shown in Figure 12).

IntrDevice - This is an abstract class which defines the common interface to all types of interruptible devices. It is used by the **IntrController** to dispatch control to interruptible devices (handleInterrupt()), and by client code to install callback functions (installCallback), which are invoked during a devices's interrupt processing (invokeCallback() and **DevCallback**::invoke()). Child classes of **IntrDevice**, may use their parent's protected method, invokeCallback() to invoke the installed callback instance.

DevCallback- This is an abstract class which defines the common interface to all classes which can be installed as a device callback (via **IntrDevice**::installCallback()). The **IntrDevice** class uses the **DevCallback** member function invoke() to pass control to the callback instance during the device's interrupt processing.

IntrGuard - This class provides a "safe" mechanism by which client code can temporarily disable interrupts during critical sections of code. The constructor for **IntrGuard** uses **Mongoose**::setStatusReg() to read the current interrupt enable state and disable interrupts. The read state is stored in a private variable within the **IntrGuard** instance. When the destructor for the class is invoked, it uses the same **Mongoose** function to restore the saved interrupt enable state. By declaring an instance of **IntrGuard** within a local scope (i.e. a function or sub-block within a function) interrupts are disabled from the point of declaration until the code leaves the scope of the block. By relying on the compiler to generate the destructor when a function returns or a local block is exited, it ensures that the original interrupt state is restored.

`Mongoose` - This class is used by the **`IntrController`** class to manage the R3000 Co-processor 0's Status and Cause registers, and the Mongoose Extended Interrupt Mask and Cause registers. Refer to Section 5.0 for a description of this class.

6.4 R3000 and Mongoose Interrupt Description

6.4.1 R3000 Interrupt Status and Cause Registers

The R3000 manages interrupts through its System Co-processor (Co-processor 0). This co-processor contains a Status Register and Cause Register (see Section 4.1). The Status Register controls interrupt enables and disables, and contains (in addition to many other control bits) 8 interrupt mask bits. Two of these bits correspond to the two R3000 software interrupts, and the other 6 correspond to hardware interrupts level-triggered lines wired to the R3000. In addition to various other status information, the co-processor's Cause register contains an Exception cause code and 8 interrupt cause bits, corresponding to the mask bits in the Status Register.

The ACIS instrument software only deals with the Interrupt Exception Cause. All other Exception codes are considered fatal errors, and during pre-flight debugging, will cause a fatal system error and reboot.

6.4.2 Mongoose Extended Interrupt Mask and Cause Registers

In addition to the R3000 interrupts, the Mongoose Microcontroller adds another suite of interrupt causes, including a Watchdog Timer Interrupt, General Purpose Timer Interrupt, DMA Interrupt, UART Receive Interrupt, UART Transmit Interrupt and three external edge triggered interrupts, and a collection of additional error exceptions (see Section 4.2). These interrupts are ORed to the R3000 hardware interrupt number 3. The device interrupts (DMA, Timers, and UART) can be individually masked via a memory-mapped register in the Mongoose's CSI block, the Extended Interrupt Mask register. The cause of a particular Mongoose Interrupt is indicated by the Extended Cause register, in the same CSI block.

6.4.3 Low-level Interrupt Processing

When the R3000 is interrupted, it saves the interrupt return address in its Co-processor 0's Exception Program Counter register, disables interrupts, places the processor into "Kernel" mode and executes its Exception Vector code located at processor address 0x80000080. This code branches to the main low-level interrupt handler, written in assembler. This handler then saves a few registers, invokes the ***Nucleus RTX*** function `SKD_Interrupt_Context_Save()`, which saves all of the R3000 registers, including the R3000 Co-processor's Exception Program Counter and Status Register, and some information on the task being interrupted. The assembler handler then calls a "C/C++" function, `intr_handler()`. This function then invokes the **`IntrController`**`::dispatchInterrupt()` function. Once **`IntrController`** and `intr_handler()` return, the assembly handler calls `SKD_Interrupt_Context_Restore()` to restore control to a task.

6.4.4 Interrupt Priorities

The R3000 does not provide built-in facilities for nesting prioritized interrupts. It treats this as a software responsibility. The recommended technique (see "The MIPS Programmer's Handbook," Section 4) for implementing prioritized nested interrupts uses a 256 entry table, indexed by the 8 interrupt cause bits from the Co-processor's Cause register. During initialization, the software sets up the table to map the 8 possible causes to a single priority selection. A second table, indexed by priority, selects an interrupt mask to use. During interrupt processing, while interrupts are disabled, the software uses the Cause bits to select the priority, and then uses priority to select a mask which disables all lower-priority interrupt causes. It then writes the mask to the Co-processor's Status Register, and re-enables interrupts. This will allow higher priority interrupts to occur while the software is processing the current interrupt level.

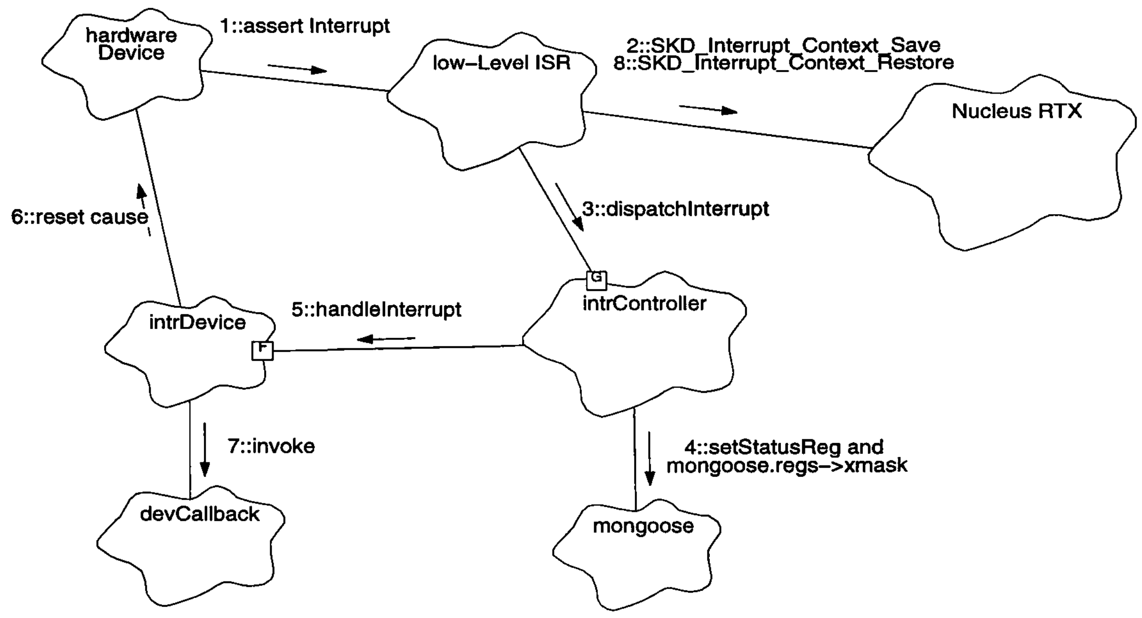
Given that the Mongoose provides another set of interrupts (see Section 4.2), located in a different register, and masked using a different mask, it is difficult for the ACIS software to use exactly the same scheme for prioritizing interrupts. Instead, the **IntrController**::`dispatchInterrupt()` function tests for each of the known interrupt causes separately, and selects both an R3000 mask and Mongoose Extended Interrupt mask which disables lower-priority device interrupts. The function then masks both the R3000 and Mongoose causes, enables interrupts, and transfers control to the interrupt device, using **IntrDevice**::`handleInterrupt()`. Once the device returns, the dispatch function disables interrupts and restores the previous R3000 and Mongoose interrupt masks. For a list of Back End hardware interrupts and priorities, see Section 4.3.13 .

6.5 Scenarios

6.5.1 Use 1::Dispatch Control to Device on Hardware Interrupt

Figure 13 illustrates the overall interrupt handling scenario.

FIGURE 13. Interrupt Handling Scenario



1. A hardware device asserts its interrupt line, causing the R3000 to execute its Exception Vector code, which then branches to the instrument software's low-level Interrupt Service Routine (ISR).
2. The low-level Interrupt Service Routine calls ***Nucleus*** to save the register and task context (`SKD_Interrupt_Context_Save`())
3. The low-level ISR then invokes `intr_handler`() (not shown) which in-turn invokes *`intrController`*.`dispatchInterrupt`().
4. `dispatchInterrupt`() gets the R3000 Co-processor 0's Cause Register and Mongoose Extended Cause (not shown) and then tests each cause in order of priority. Once the cause is determined, it selects the corresponding interrupt device instance to invoke, and determines which higher priority interrupts to enable. It then sets the Mongoose Extended Interrupt Mask register and R3000 Status register to mask off this interrupt cause and all lower priority interrupt causes. It then re-enables interrupts.
5. `dispatchInterrupt` () then invokes the selected device's **`IntrDevice`**::`handleInterrupt`() member function. (NOTE: Each subclass of **`IntrDevice`** is required to provide its own implementation of this function).
6. The member function then resets the cause of the interrupt in the hardware device, and performs any low-level hardware-specific maintenance operations.

7. The member function then invokes the installed callback instances **DevCallback**::invoke() function to handle any higher level device-specific operations. (NOTE:: Each subclass of **DevCallback** is required to implement its own version of this invoke()).
8. Once all interrupt processing is complete (i.e. **DevCallback**::invoke(), **IntrDevice**::handleInterrupt(), **IntrController**::dispatchInterrupt() return, the low-level ISR calls ***Nucleus RTX*** SKD_Interrupt_Context_Restore() to restore the context of the next task to run (or the same task if no context switch occurred as a result of the interrupt).

6.5.2 Use 2:: Provide Interrupt Disable/Restore for Critical Code Sections

In order to allow client code to perform a set of operations without worrying about one or more of the interrupt handlers modifying things behind its back, the system provides an **IntrGuard** class provides the ability to temporarily disable interrupts. This system relies on the C++ behavior of invoking a class constructor when an instance of that class is declared within a function or block within a function, and invoking its destructor when the function returns, or the block is ended.

The following C++ code fragment illustrates how to use the **IntrGuard** within a function.

```
// ---- Variable shared by both foo() and an interrupt handler ----
volatile unsigned sharedVariable;

void foo ()
{
    // By declaring guard, interrupt state is saved,  and interrupts are disabled
    // (The private, const variable guard.oldState contains the previous
    // interrupt enable state)
    IntrGuard guard;

    // --- Read variable, increment and write back
    sharedVariable++;
    if (sharedVariable < 10)
    {
        return;// -- Interrupt restored by guard destructor
    }
    sharedVariable = 0;
    return;     // --- Interrupt state restore by guard destructor
}
```

Another example shows its use within a block inside a function:

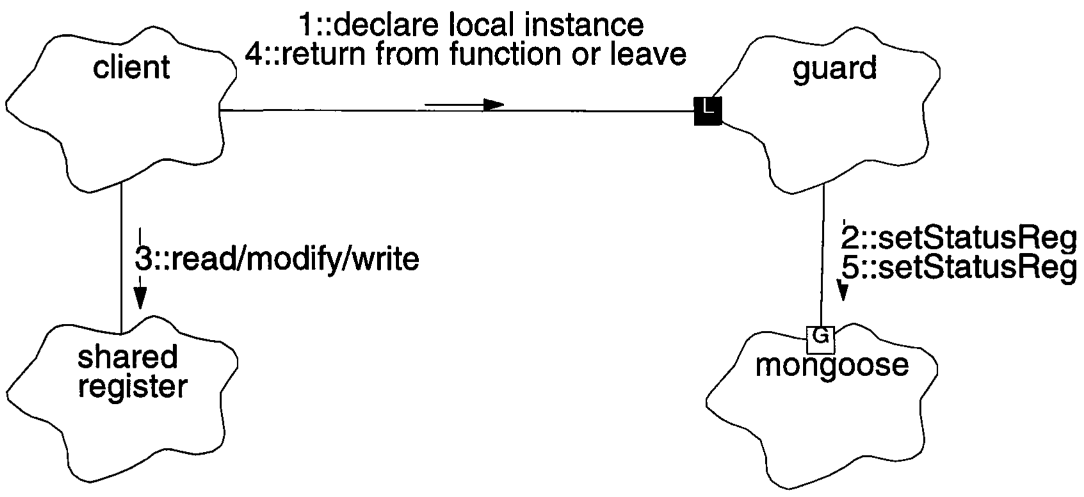
```
// ---- Variable shared by both bar() and an interrupt handler ----
volatile unsigned sharedVariable;
unsigned nonSharedVariable;

void bar()
{
    nonSharedVariable++;
    {
        // Constructor saves old state and disable interrupts
        IntrGuard guard;
        sharedVariable++;// Safely read/modify/write
    } // Leaving block invokes destructor and restores interrupts

    // Modifying sharedVariable outside block may cause problems
    // due to contention with interrupt handler.
}
```

Figure 14 illustrates the overall use of **IntrGuard**.

FIGURE 14. Performing block interrupt disables and restores



1. Client code declares *guard*, a local instance of an **IntrGuard**, causing its constructor to be invoked (**IntrGuard**::IntrGuard()).
2. The constructor, **IntrGuard**::IntrGuard() invokes **Mongoose**::setStatus-Reg() to get the old contents of the R3000 Co-processor 0's Status Register, and to write an interrupt disable value to the register.disable interrupts. **IntrGuard**::Intr-Guard() uses the returned value to initialize its const (read-only) *oldState* instance variable.
3. The client code modifies shared variable or register at will, without worrying about interrupt handlers modifying the same variable/register.

4. Upon completion of the protected code, the client code either returns from the protected function, or leaves the protected block of code. Upon leaving a function or block which declared the **IntrGuard** instance, the guard's destructor is invoked (**Intr-Guard**::~IntrGuard()).
5. The guard's destructor, ~IntrGuard() passes the saved interrupt enable state, old-State, to **Mongoose**::setStatusReg() which restores the contents of the R3000 Co-processor 0's Status Register.

6.5.3 Use 2:: Pass Control to Client Code during Interrupt Processing

In order to support higher level operations during interrupt processing, the **IntrDevice** class contains a pointer to an installed **DevCallback** instance. Client applications install the callback during initialization using **IntrDevice**::installCallback(). When an interrupt handler is invoked, the handler is responsible for invoking the callback, **DevCallback**::invoke(). **IntrDevice** provides a function which does this, **IntrDevice**::invokeCallback(), which in general covers most needs. Figure 12, "Interrupt Controller and Device Relationships," on page 160 illustrates how and when the callback is invoked.

6.6 Class IntrDevice

Documentation:

The **IntrDevice** class is an abstraction of all interruptible devices. It serves to provide a set of common interfaces to all such devices. All sub-classes of this class MUST implement the own version of the following: `handleInterrupt ()`

Export Control:	Public
Cardinality:	n

Hierarchy:

Superclasses:	**none**

Public Uses:

DevCallback

Public Interface:

Operations:	`handleInterrupt()`
	`installCallback()`

Protected Interface:

Operations:	`invokeCallback()`

Private Interface:

Has-A Relationships:

DevCallback* *callbackPtr*: This is a pointer to an instance of **DevCallback**. This is used by clients to obtain control whenever a device interrupt occurs.

Concurrency:	Synchronous
Persistence:	Persistent

6.6.1 handleInterrupt()

Public member of: **IntrDevice**

Return Class: **void**

Documentation:

This member function handles any device-specific interrupt operations. This function MUST be implemented by all leaf subclasses of **IntrDevice**.

Semantics:

Handle the device specific interrupt. Usually, subclass implementations of this function reset the device-specific cause of the interrupt and then invoke **IntrDevice**::invokeCallback()

Time complexity: < 10ms

Concurrency: Synchronous

6.6.2 installCallback()

Public member of: **IntrDevice**

Return Class: **void**

Arguments:

DevCallback* *callback*

Documentation:

This function is used by a client to install an **DevCallback** instance, pointed to by *callback*, to be invoked whenever a device interrupt occurs.

Preconditions:

Another *callback* must not have been previously installed (i.e. *callbackPtr* must be NULL)

Semantics:

Store *callback* in *callbackPtr*.

Concurrency: Sequential

6.6.3 invokeCallback()

Protected member of: **`IntrDevice`**

Return Class: **`void`**

Documentation:

This function invokes the installed callback function: **`DevCallback`**::`invoke`(). If no callback is installed, this function has no effect.

Semantics:

If no callback installed, do nothing, otherwise, invoke the installed callback, *`callbackPtr`*->`invoke`().

Time complexity: < 10ms

Concurrency: Synchronous

6.7 Class DevCallback

Documentation:

The **DevCallback** class is an abstract class which defines a common interface to all callback functions invoked by device-specific interrupt handlers. All subclasses of this class MUST implement their own versions of the following functions: `invoke(`**`IntrDevice`**`*)`

Export Control:	Public
Cardinality:	n
Hierarchy:	
Superclasses:	**none**
Public Interface:	
Operations:	`invoke()`
Concurrency:	Synchronous
Persistence:	Persistent

6.7.1 invoke()

Public member of: **`DevCallback`**

Return Class: **`void`**

Arguments:

`IntrDevice*` *`device`*

Documentation:

This function is invoked by a device-specific interrupt handler. This function then performs whatever client specific operations are needed. It is intended that this function be implemented by the various sub-classes of this class.

Semantics:

Perform client-specific device interrupt operations

Concurrency: Synchronous

6.8 Class IntrController

Documentation:

IntrController is responsible for dispatching control to various devices when an interrupt occurs.

Export Control:	Public
Cardinality:	1
Hierarchy:	
Superclasses:	**none**
Public Uses:	**IntrDevice**
Implementation Uses:	**Mongoose** **Dma** **Watchdog** **Timer** **CmdDevice** **TlmDevice**
Public Interface:	

Operations:	`disableInts()`
	`dispatchInterrupt()`
	`enableInts()`
	`restoreInts()`

Concurrency:	Synchronous
Persistence:	Persistent

6.8.1 disableInts()

Public member of: **`IntrController`**

Return Class: **`unsigned`**

Documentation:

Shutdown all interrupts, and return previous interrupt state.

Semantics:

Use *mongoose*.`setStatusReg()` to disable interrupts while obtaining the previous interrupt state.

Concurrency: Synchronous

6.8.2 dispatchInterrupt()

Public member of:	**`IntrController`**
Return Class:	**`void`**

Arguments:	**`unsigned`** *`icause`*
	`unsigned` *`xcause`*

Documentation:

This function dispatches control to each device with a pending interrupt. Each device is serviced in prioritized order, with higher priority devices having the interrupts re-enabled.

Semantics:

Use *`icause`* and *`xcause`* to determine which device is requesting service. While interrupts are disabled, unmask all higher priority interrupt causes on both the R3000 and the Mongoose XMask and re-enable interrupts. Then invoke the interrupting device `handleInterrupt()` function. Once the function returns, disable interrupts and restore the previous Mongoose and R3000 interrupt mask states. Then re-read the R3000 and Mongoose interrupt cause registers, and handle the next cause (prevents unnecessary context saves and restores).

Time complexity: < 10ms

Concurrency: Synchronous

6.8.3 enableInts()

Public member of: **`IntrController`**

Return Class: **`unsigned`**

Documentation:

This function enables interrupts. It returns the previous interrupt enable/disable state.

Semantics:

Use *mongoose*.`setStatusReg`() to enable interrupts while retrieving the previous interrupt state.

Concurrency: Synchronous

6.8.4 restoreInts()

Public member of: **`IntrController`**

Return Class: **`void`**

Arguments:

`unsigned` *`oldState`*

Documentation:

Restore a previous interrupt enable/disable state. *`oldState`* is the value returned from `disableInts`() or `enableInts`().

Semantics:

Use *mongoose*.`setStatusReg`() to restore the passed interrupt state.

Concurrency: Synchronous

6.9 Class IntrGuard

Documentation:

This class is used within the scope of a block of code to disable interrupts for the duration of the block. Its constructor saves the current interrupt state and disables interrupts. Its destructor, invoked at the end of a given code block, restores the saved interrupt state.

Export Control:	Public
Cardinality:	n

Hierarchy:

Superclasses:	**none**

Implementation Uses:

Mongoose

Public Interface:

Operations:	`IntrGuard()`
	`~IntrGuard()`

Private Interface:

Has-A Relationships:

const unsigned *oldState*: This variable is set during the instance's constructor and reflects the interrupt state at the time the guard was created. The variable is used upon destruction to restore the interrupt state.

Concurrency:	Synchronous
Persistence:	Transient

6.9.1 IntrGuard()

Public member of:	**IntrGuard**
Return Class:	**IntrGuard**

Documentation:

The constructor for **IntrGuard** disables interrupts while saving the previous interrupt state in *oldState*.

Semantics:

Use *mongoose*.setStatusReg() to get the old interrupt state while disabling interrupt. The previous state is stored by initializing *oldState*. Since *oldState* is read-only, the entire operation of this constructor occurs in the initialization statement portion of the constructor.

Concurrency: Synchronous

6.9.2 ~IntrGuard()

Public member of:	**IntrGuard**
Return Class:	**void**

Documentation:

The destructor for **IntrGuard** restores the interrupt state prior to when the instance was constructed (see IntrGuard()).

Semantics:

Use *mongoose*.setStatusReg() to restore the state indicated by *oldState*.

Concurrency: Synchronous