CSR	MASSACHUSETTS INSTITUTE OF TECHNOLOGY CENTER FOR SPACE RESEARCH CAMBRIDGE, MASSACHUSETTS 02139	
REVISION LOG	**TITLE:** Software Detailed Design Mongoose Devices	**DOC. NO.** 36-53207

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
—	1/23/95	—	—	Initial version for design walkthrough	
A	5/3/95	36-230	all	Incorporate comments Added Timer & Watchdog devices	[illegible] 5/16/95

7.0 Mongoose Devices (36-53207 A)

7.1 Purpose

The Mongoose devices consists of a Direct-Memory-Access (DMA) Controller, General Purpose Timer, and a Watchdog Timer. The purpose of the Mongoose Device classes are to provide access to the each of these devices. Refer to TBD for a description of the hardware interfaces to these devices.

7.2 Uses

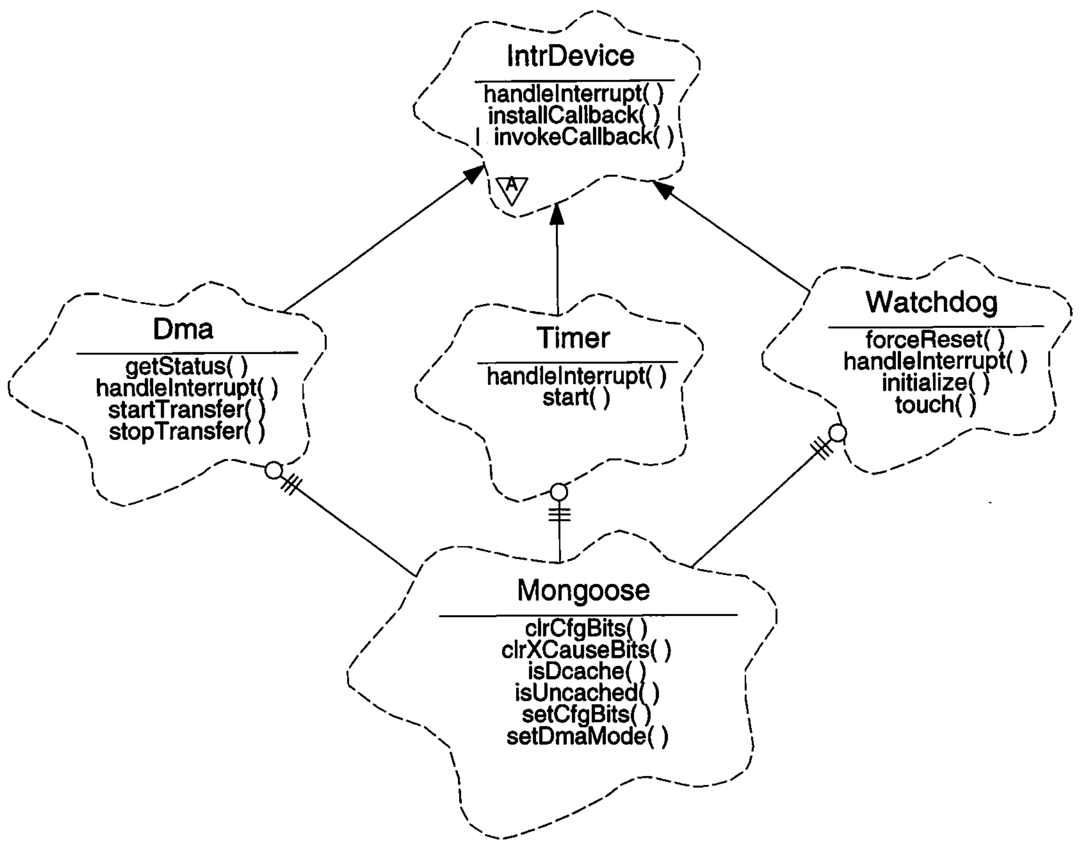
The Mongoose **`Dma`**, **`Timer`** and **`Watchdog`** classes provide the following features:

Use 1:: Initiate DMA Transfers
Use 2:: Stop DMA Transfers
Use 3:: Handle DMA interrupts and forward control to an installed callback object
Use 4:: Manage the General Purpose Timer
Use 5:: Prevent hardware resets by setting Watchdog Timer count
Use 6:: Force a hardware reset using the Watchdog Timer

7.3 Organization

The Mongoose DMA, Timer and Watchdog devices are interruptible, and therefore are all a subclass of **`IntrDevice`** (see Section 6.0). Each of these classes use the **`Mongoose`** class to obtain access to the Mongoose's Command/Status Interface (CSI) registers, and the **`IntrGuard`** class (not shown) to prevent interrupts during certain sections of code. Figure 15 illustrates the relationships used by the **`Dma`**, **`Timer`** and **`Watchdog`** classes.

FIGURE 15. Mongoose Dma, Timer and Watchdog Class Relationships



Dma - This class represents the Mongoose DMA device. It provides functions which start and stop DMA transfers (`startTransfer`, `stopTransfer`), return the status of the controller (`getStatus`), and handle interrupts (`handleInterrupt`). In addition to theses functions, it inherits the ability to install and invoke interrupt callback instances from **IntrDevice**.

Timer- This class represents the Mongoose General-Purpose Timer device. It provides functions which start the timer, specifying the interrupt period (`start`), and handle interrupts (`handleInterrupt`). In addition to theses functions, it inherits the ability to install and invoke interrupt callback instances from **IntrDevice**.

Watchdog- This class represents the Mongoose Watchdog Timer device. It provides functions to configure the watchdog time-out period (`initialize`), sets the down-counter to the time-out value (`touch`), and use the timer to force a system (`forceReset`). In order to support debugging when the reset logic is not enabled in the hardware, it also provides a function to handle interrupts (`handleInterrupt`) and inherits the ability to install and invoke interrupt callback instances from **IntrDevice**.

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 **Dma**, may use their parent's protected method, `invokeCallback()` to invoke the installed callback instance (see Section 6.0).

Mongoose - This class represents the lowest level hardware access to the features provided by the R3000 core processor and the Mongoose Microcontroller. The **Dma**, **Timer** and **Watchdog** classes use the **Mongoose** class to obtain access to the Mongoose's DMA, Timer, and Watchdog Count, Timer Count registers, the Mongoose Control Register (via `setDmaMode`, `setCfgBits`, and `clrCfgBits`), and the interrupt cause register (`clrXCauseBits`). The **Dma** class also uses the Mongoose class to test addresses for legal Mongoose DMA transfers (`isDcache`, `isUncached`).

7.4 DMA Transfer Types and Restrictions

The "Mongoose ASIC Microcontroller Programming Guide, Section: D-side Interface Block, DMA Channel" describes the interface to the DMA controller and lists the kinds of transfers supported by the Mongoose's DMA Controller. These transfer types are:

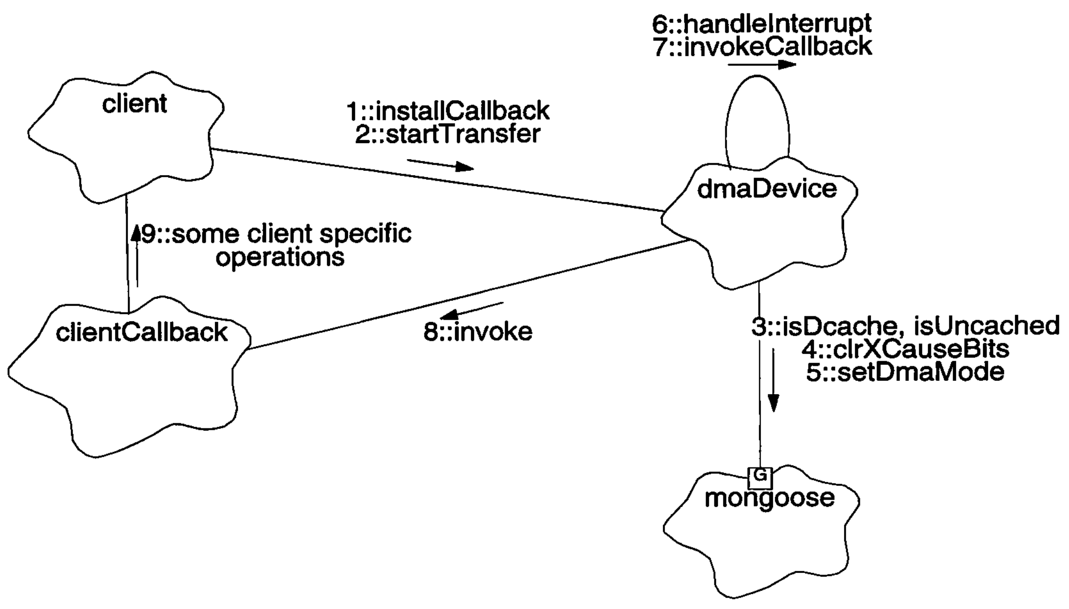
- Transfers from one uncached addressable region to another uncached region
- Transfers from an uncached address region into the data cache
- Transfers from the data cache into an uncached address region

All DMA transfers use 32-bit word reads and writes. When used with a 16-bit memory-mapped device (such as the Command FIFO), the most significant 16-bits of each copied DMA word will contain unspecified values.

7.5 Scenarios

The following diagram is used to illustrate the overall process involved in DMA transfers. Section 7.5.1 and Section 7.5.3 describe the scenarios illustrated by this diagram.

FIGURE 16. DMA Transfers and Interrupt Handling



7.5.1 Use 1: Initiate DMA Transfers

The primary purpose of the DMA controller is to transfer information from one section of memory or device to another. Only one transfer is performed at a time, and it is up to the users of the **Dma** class to arbitrate (using `getStatus`) among themselves for access to the controller.

1. During system initialization, the user of the **Dma** class constructs a subclass of **DevCallback** (not shown) and passes it to the **Dma**'s `installCallback()` function (inherited from **IntrDevice**).
2. When the client wishes to perform a transfer, it invokes the **Dma**'s `startTransfer()` function.
3. `startTransfer()` then determines the type of transfer by inspecting the source and destination addresses, using the **Mongoose**'s `isDcache()` and `isUncached()` determine if an address is located within data cache or external memory, respectively.
4. Once the type of transfer is determined, `startTransfer()` uses the **Mongoose** function `clrXCauseBits()` to clear any outstanding DMA interrupts.
5. It then uses the **Mongoose** "*regs*" structure (not shown) to program the source, destination and transfer length registers, and sets the transfer into motion using `setDmaMode()`. Once the transfer completes, a DMA interrupt will be generated and the interrupt controller invokes the **Dma**'s `handleInterrupt()` function. The interrupt handling actions are described below in Section 7.5.3

7.5.2 Use 2: Stop DMA Transfers

This use is not illustrated in Figure 16.

To abort a DMA transfer in progress, the user of the **Dma** invokes the `stopTransfer()` function. This function then uses the **Mongoose** functions `setDmaMode` and `clrXCauseBits` to stop the transfer, and clear any pending DMA interrupt.

7.5.3 Use 3: Handle DMA Interrupts

Refer to Figure 16, "DMA Transfers and Interrupt Handling," on page 137 for an illustration of the scenario described in this section.

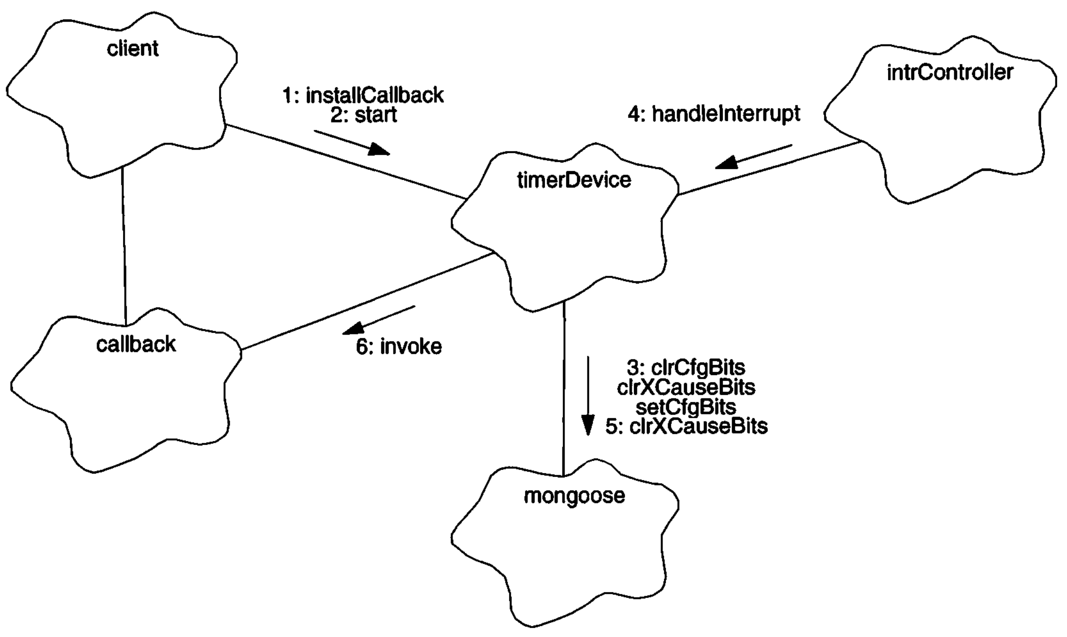
Whenever a DMA interrupt occurs, the Interrupt Controller (not shown) determines the cause of the interrupt, masks off any lower priority interrupt causes and re-enables interrupts. It then dispatches control to the **Dma** class's `handleInterrupt()`. The following describes the actions taken by the **Dma** class in response to an interrupt.

6. Once the DMA transfer completes, the interrupt controller invokes the **Dma**'s `handleInterrupt()` function. `handleInterrupt()` then clears the interrupt cause latch using the **Mongoose**'s `clrXCauseBits()` function, as in step 4 (see Section 7.5.1).
7. `handleInterrupt()` then invokes **IntrDevice**::`invokeCallback()` (inherited by **Dma** from **IntrDevice**) to test for and invoke the installed callback instance.
8. **IntrDevice**::`invokeCallback()` then calls the installed callback instance's `invoke()` function.
9. The callback's `invoke()` function then performs client specific operations needed to deal with the end of the DMA transfer.

7.5.4 Use 4: Manage the General Purpose Timer

Figure 17 illustrates the management of the Mongoose's General Purpose Timer.

FIGURE 17. Timer management

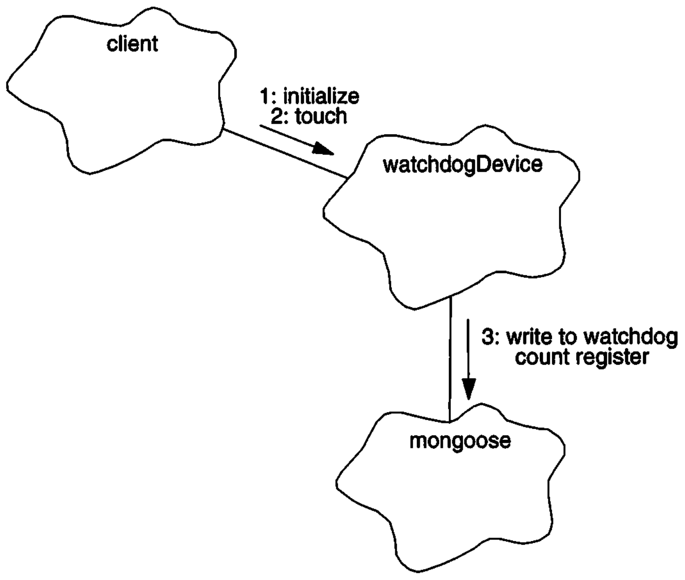


1. A client object installs a callback instance, callback, to be invoked during interrupt processing, using *timerDevice*.`installCallback()`.
2. The client starts the timer, specifying the interrupt period to be generated, using *timerDevice*.`start()`.
3. *timerDevice*.`start()` saves the passed count into an instance variable, disables the timer enable bit, using *mongoose*.`clrCfgBits()`, clears any pending timer interrupt using *mongoose*.`clrXCauseBits()`, writes the passed count into the timer count register (not shown), and enables the timer using *mongoose*.`setCfgBits()`.
4. When the timer's count reaches zero, it generates an interrupt. The interrupt controller object, *intrController*, determines the cause, and tells the device object to handle the interrupt, using *timerDevice*.`handleInterrupt()`.
5. `handleInterrupt()` re-loads the timer's count register and clears the interrupt using *mongoose*.`clrXCauseBits()`.
6. `handleInterrupt()` then invokes the callback instance, using *callback*.`invoke()`. The process repeats from Step 4 until the instrument is reset.

7.5.5 Use 5: Prevent hardware resets by setting Watchdog Timer count

Figure 18 illustrates how a client configures and uses the Watchdog Timer. Since the normal operating environment resets the processor rather than cause watchdog interrupts, the diagram and subsequent description omit interrupt handling.

FIGURE 18. Watchdog Timer Management



1. The client sets the watchdog time-out period using `watchdogDevice.initialize()`.
2. At a rate within the time-out period, the client prevents a watchdog reset from occurring by periodically calling `watchdogDevice.touch()`.
3. `watchdogDevice.touch()` writes the configured time-out value into the watchdog down-count register. If the client does not call `touch()` within the configured period, the down-count register will reach zero, and the hardware will cause a reset on the Back End Processor.

7.5.6 Use 6: Force a hardware reset using the Watchdog Timer

To force a hardware reset, the client calls `watchdogDevice.forceReset()`. `forceReset()` stores a 1 into the Mongoose's watchdog down-count register and attempts to enter an infinite loop. The down-counter decrements once and reaches zero, causing the hardware to generate a physical reset of the Back End Processor.

7.6 Class Dma

Documentation:

This class represents the Mongoose on-chip Direct-Memory-Access (DMA) controller.

Export Control:	Public
Cardinality:	1

Hierarchy:

Superclasses: **IntrDevice**

Implementation Uses:

Mongoose

Public Interface:

Operations:

```
getStatus()
handleInterrupt()
startTransfer()
stopTransfer()
```

Protected Interface:

Has-A Relationships:

DmaState *dmaState*: This variable contains the current state of the DMA device.

Concurrency: Guarded

7.6.1 getStatus()

Public member of: **`Dma`**

Return Class: **`DmaState`**

Documentation:

This function returns the current transfer status of the **`Dma`** class. The return values are as follows:

DMASTATE_IDLE- No transfer in progress
DMASTATE_BUSY - A DMA transfer is underway

Concurrency: Guarded

7.6.2 handleInterrupt()

Public member of: **`Dma`**

Return Class: **`void`**

Documentation:

This function handles DMA interrupts, by clearing the DMA extended cause bit, and by invoking the installed interrupt callback instance.

Postconditions:

If no new transfers are started by the callback instance, subsequent calls to `getStatus()` will return DMASTATE_IDLE. If the callback starts a new transfer, `getStatus()` will return DMASTATE_BUSY if the new transfer is still underway.

Concurrency: Synchronous

7.6.3 startTransfer()

Public member of: **Dma**

Return Class: **Boolean**

Arguments:

const unsigned* *srcAddress*
volatile unsigned* *dstAddress*
unsigned *xfrLength*

Documentation:

This function initiates a DMA transfer, copying *xfrLength* words from *srcAddress* to *dstAddress*. If the arguments are valid, this function starts the transfer and returns `BoolTrue`. If the arguments are invalid, it returns `BoolFalse`.

Preconditions:

A transfer must not already be in progress.

srcAddress and *dstAddress* and *xfrLength* must be set to support one of the following transfer types:

data cache to external memory/device transfer
external memory/device to data cache transfer
external memory/device to external memory/device.

Semantics:

Test transfer type, clear DMA mode and DMA interrupt, program source, destination and length and set the DMA mode to kick off the transfer

Postconditions:

A transfer will be underway (or already completed if very short). A DMA interrupt may occur after this function returns and the buffer pointed to by *dstAddress* will be changing under one's feet. Calls to `getStatus()` may return either DMASTATE_IDLE if the transfer has already completed, or DMASTATE_BUSY if the transfer is still underway.

Concurrency: Guarded

7.6.4 stopTransfer()

Public member of: **`Dma`**

Return Class: **`void`**

Documentation:

This function stops a DMA transfer in-progress and clears any pending DMA interrupts.

Semantics:

Set the DMA mode to stop a transfer and clear the DMA Extended Cause bit.

Postconditions:

Any transfers will be aborted. Calls to `getStatus`() will return DMASTATE_IDLE.

Concurrency: Guarded

7.7 Class Timer

Export Control:	Public
Cardinality:	1

Hierarchy:

Superclasses:	**`IntrDevice`**

Implementation Uses:

`Mongoose`

Public Interface:

Operations:	`handleInterrupt()`
	`start()`

Protected Interface:

Has-A Relationships:

`unsigned` *`timeout`*: This is the value to store into the timer's counter upon each interrupt.

Concurrency:	Guarded
Persistence:	Persistent

7.7.1 handleInterrupt()

Public member of: **`Timer`**

Return Class: **`void`**

Documentation:

This function overloads the **`IntrDevice`** `handleInterrupt` function. For the **`Timer`** device, this function clears the interrupt cause, re-loads the timer's count and invokes the installed callback.

Concurrency: Guarded

7.7.2 start()

Public member of: **`Timer`**

Return Class: **`void`**

Arguments:

`unsigned` *`clockTicks`*

Documentation:

This starts the Mongoose Timer. *`clockTicks`* specifies the number of processor clocks per timeout of the timer.

Concurrency: Guarded

7.8 Class Watchdog

Export Control:	Public
Cardinality:	1

Hierarchy:

Superclasses:	**IntrDevice**

Implementation Uses:

Mongoose

Public Interface:

Has-A Relationships:

unsigned *count*: This is the time-out count to use. Each call to `touch()` causes this count to be loaded into the timer's counter.

Operations:

```
forceReset()
handleInterrupt()
initialize()
touch()
```

Concurrency:	Guarded
Persistence:	Persistent

7.8.1 forceReset()

Public member of: **`Watchdog`**

Return Class: **`void`**

Documentation:

This function causes the watchdog timer to reset the instrument hardware by writing a 1 into the timer's down-count register and entering an infinite loop. When the down-counter reaches 0, the hardware will reset the Back End Processor.

Concurrency: Guarded

7.8.2 handleInterrupt()

Public member of: **`Watchdog`**

Return Class: **`void`**

Documentation:

This function overloads the **`IntrDevice`** `handleInterrupt` function, and is used when debugging the instrument software.This function clears the interrupt cause, and invokes the installed callback.

Concurrency: Guarded

7.8.3 initialize()

Public member of: **Watchdog**

Return Class: **void**

Arguments:

unsigned *timeout*

Documentation:

This function sets the time-out period of the watchdog timer. *timeout* is the maximum number of clock cycles that can occur between each call to `touch()`.

Concurrency: Guarded

7.8.4 touch()

Public member of: **Watchdog**

Return Class: **void**

Documentation:

This function copies the configured *count* value into the mongoose's watchdog down-count register. This function must be called at least once every *count* clock cycles, or the timer will expire and reset the processor (or, if debugging, cause a watchdog interrupt).

Concurrency: Synchronous