Timing Module Specification

James Francis

MIT 64-30110.0701

Rev. D

May 2, 1994

Contents

Figures

1 Introduction

1.1 Scope

This document describes the components that make up the Timing Software Module (TIMING). This module consists of a set of functions that provide the Flight Software interface to the EA's Back End and Front End Timing Circuitry.

This document does not describe the overall design of the EDS, nor does it describe the operating details of each component in this module. It assumes that the reader is extremely familiar with the Flight Software Global Design, with the "C" programming language and with the architecture of the EDS.

This document should be of primary interest to Flight Software Engineering, Flight Hardware Engineering, Ground Software Engineering and Software Quality Assurance.

1.2 References

MIT 64-02009 Rev. E .... "EDS-PCA Science Requirements"
MIT 64-02012 Rev. D .... "EDS-ASM Science Requirements"
MIT 64-02006.01 Rev. F .... "XTE EDS Hardware Functional Description and Requirements"
MIT 64-30102 Rev. F .... "Flight Software Global Design".
MIT 64-30112.0601 Rev. D .... "Science Thread Module Specification" (SCIENCE).
MIT 64-30112.0201 Rev. C .... "Housekeeper Thread Module Specification" (HOUSE).
MIT 64-30110.1701 Rev. D .... "Thread Manager Module Specification" (THREAD)
MIT 64-30110.2301 Rev. D .... "Startup Module Specification" (STARTUP).
MIT 64-30110.0101 Rev. C .... "Back End Registers Module Specification" (BREG)
MIT 64-30110.0901 Rev. D .... "Front End Registers Module Specification" (FREG)
MIT 64-30110.1201 Rev. E .... "Interrupt Manager Module Specification" (INTR)
MIT 64-30110.2001 Rev. D .... "Utility Module Specification" (UTIL)

1.3 Notations

This document assumes the following notations:

UINT8 .... Refers to an 8-bit unsigned value.
UINT16 .... Refers to a 16-bit unsigned value.
UINT32 .... Refers to a 32-bit unsigned value.

BOOL .... Refers to a 16-bit value which represents a TRUE or FALSE condition.

1.4 Module Perspective

The following diagram is a copy of the Level 1 data flow diagram from the **Flight Software Global Design**, Section 4.1.2. The Timing Module has a scope which is indicated by the shaded area:

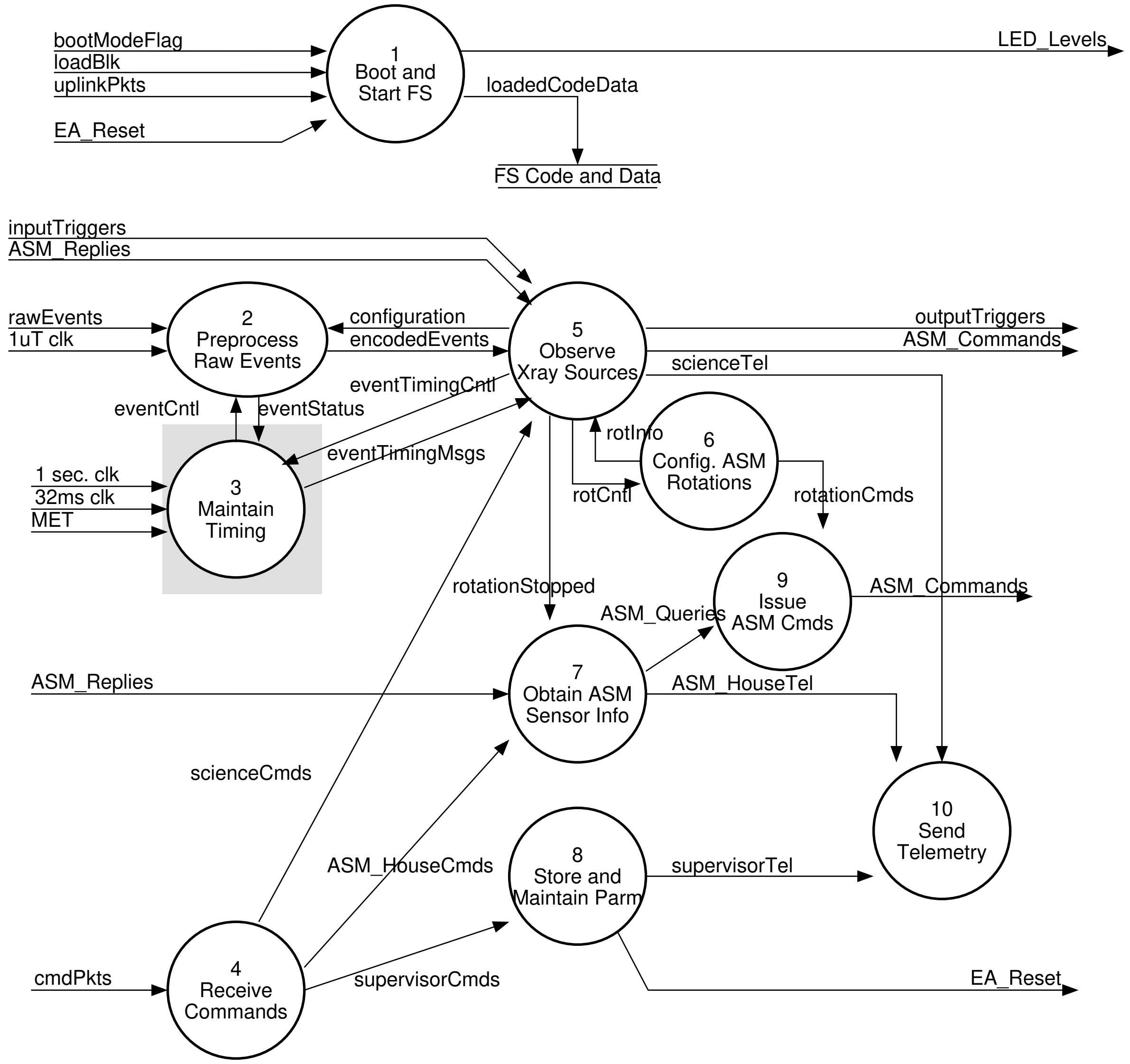

Figure 1 Module Perspective

1.5 Document Overview

Sections 1 and 2 provide an overview of the Timing Module, summarizing the purpose, services, context, and interfaces of the module. Section 3 describes all software structures and constants visible to modules outside of the Timing Module. Sections 4 and 5 describe the interrupts and events used by the Timing Module. Section 6 describes the functions provided to other modules by the Timing Module. Section 7 describes some of the implementation details of the module. Section 8 describes which requirements are met by the module. Section 9 describes what types of tests need to be performed to ensure that the implementation of the module is operating properly. Appendix A lists items which are currently undefined, or that need to be reviewed.

1.6 Acronyms

Please refer to the **Flight Software Global Design Specification**, Section 1.5, for a list of acronyms and abbreviations.

2 Overview

2.1 Purpose

The purpose of this module is to manage all timing related operations within the Flight Software. These include tracking the current Mission Elapsed Time (MET), handling MinorTick (32ms) interrupts, and managing the Front End Timing Circuitry.

2.2 Services

The Timing Module provides the following services to the rest of the Flight Software:

1) Handle MinorTick Interrupts .... The Timing Module handles MinorTick interrupts, using these interrupts to time MET acquisition and HEXTE burst trigger assertion/deassertion.
2) Provide MinorTick Callback .... The Timing Module provides a mechanism to install a callback function, which is invoked whenever a MinorTick interrupt occurs.
3) Maintain Start and Current Time . The Timing Module is responsible for obtaining the SDS provided MET values during startup and during operation, and for supplying the maintained values to the rest of the Flight Software.
4) MajorTick Event Notification ...... The Timing Module is responsible for notifying all threads in the Flight Software whenever a MajorTick occurs.
5) Configure Time Bin Hardware..... The Timing Module provides the mechanism to configure the Front End's BinClock duration and TimeBin nibble mask.
6) Schedule Intervals .... The Timing Module provides the mechanism for configuring and scheduling the Front End's Interval timer and flags.
7) Interval Interrupt Handling........... The Timing Module is responsible for handling Interval Interrupts, generated by the Front End hardware.
8) Interval Interrupt Callback .... The Timing Module provides a mechanism to install a callback function, which is invoked whenever an Interval Interrupt occurs.

9) First Flag Interrupt Handling........	The Timing Module is responsible for handling First Flag Interrupts, generated by the Front End hardware.
10) First Flag Interrupt Callback	The Timing Module provides a mechanism to install a callback function, which is invoked whenever a First Flag Interrupt occurs.
11) HEXTE Output Trigger Cntl.	The Timing Module provides the mechanism to pulse the HEXTE Output Burst Trigger signal within the timing constraints specified by **XTE EDS Hardware Functional Description and Requirements**, Section 2.1.4.12.3.

2.3 Module Context

The following illustrates the Timing Module relationships with the rest of the Flight Software residing in an EA:

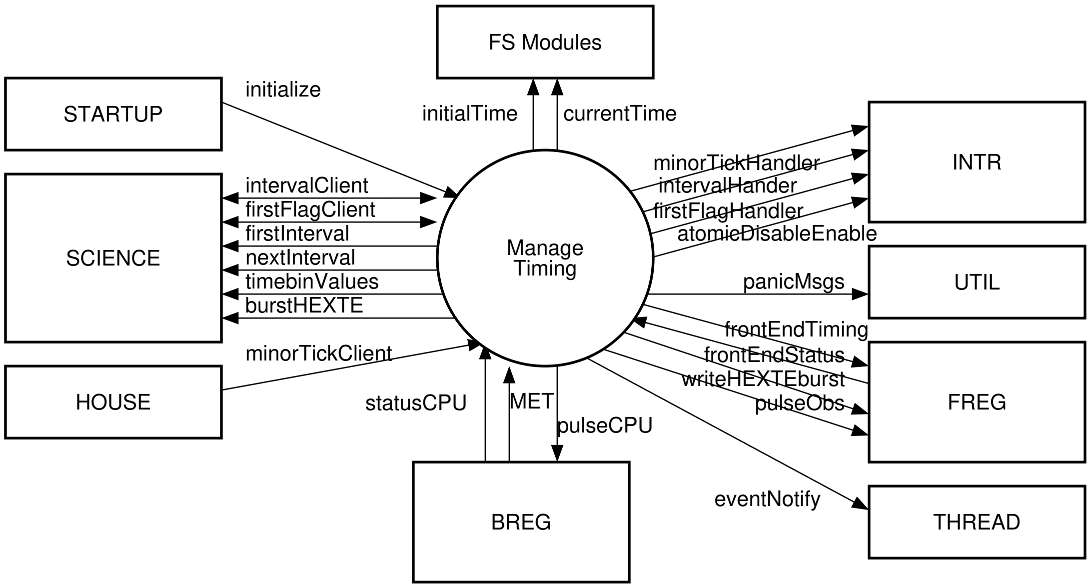

Figure 2 Timing Module Context Diagram

During system initialization, the **Startup Module** calls the Timing Module's initialization routine (initialize). The routine resets all pending Back End time related interrupt latches (pulseCPU) and Front End interrupt latches (pulseObs), waits for a MajorTick to occur (statusCPU) and then reads the current MET value (MET). After this point, other Flight Software modules may acquire the current time (currentTime) or the initial MET value read during initialization (initialTime) from the Timing Module. The initialization routine then uses the **Interrupt Manager Module** to install handlers for the MinorTick (minorTickHandler), Interval (intervalHandler) and First Flag (firstFlagHandler) interrupts. Whenever the Timing Module is performing non-interruptable operations, it uses the **Interrupt Manager Module** to atomically disable and reenable interrupts (atomicDisableEnable).

The **Science Thread Module** uses the Timing Module to control the Front End Timing Circuitry (timebinValues, firstInterval, nextInterval) and to install callback functions for the Interval and First Flag Interrupts (intervalClient, firstFlagClient). The Timing Module controls the Front End circuits (frontEndTiming) and obtains the status (frontEndStatus) using the **Front End Registers Module**.

The **Science Thread** also uses the Timing Module to pulse the HEXTE Output Burst Trigger signal (burstHEXTE). The Timing Module asserts the HEXTE trigger on a MinorTick interrupt, and deasserts the signal on the next MinorTick interrupt, using facilities provided by the **Front End Registers Module** (writeHEXTEburst).

The **ASM Housekeeper Module** uses the Timing Module to install a MinorTick callback function (minorTickClient) which is called whenever a MinorTick interrupt occurs.

The Timing Module notifies all threads whenever a MajorTick pulse occurs (eventNotify) using the **Thread Manager Module**.

If the Timing Module detects a system error, it issues a panic message and resets the EA using the **Utility Module** (panicMsg).

2.4 Theory of Operation

The Timing Module is responsible for managing all Back End and Front End time related operations. These operations can be split into two groups; MinorTick operations, and Front End timing operations.

The Timing Module uses the MinorTick Interrupt to perform certain Back End real-time operations. These operations are determined by certain hardware interface requirements. These requirements are:

Mission Elasped Time Updates .... The **XTE EDS Hardware Functional Description and Requirements**, Section 2.1.4.6.1.1, states that the MET is delivered to the EDS at least 100ms before the next MajorTick pulse and at least 100ms after the previous MajorTick pulse, and that the Flight Software should read this value within this 200ms window of a MajorTick.

HEXTE Output Burst Trigger .... The **XTE EDS Hardware Functional Description and Requirements**, Section 2.1.4.12.3, states that the Flight Software must assert the HEXTE Output Burst Trigger signal for a minimum of 20us, and a maximum of 100ms.

Watchdog Timer .... The **XTE EDS Hardware Functional Description and Requirements**, Section 2.1.4.2.1, states that, once started, the Watchdog Timer must be touched at least once every 8 seconds to prevent a hardware reset.

Enabling Interval Generation .... This requirement is Front End related, but is coupled with the Back End time management. The **XTE EDS Hardware Functional Description and Requirements**, Section 2.1.4.13.1, states that, for a deterministic start time, the Interval Enable must not be asserted within 1us before MajorTick pulse.

The Timing Module addresses each of these requirements, either directly, or indirectly, using the MinorTick interrupt. The following explains how the MinorTick interrupt handler addresses each constraint:

1) MET Update .... The MinorTick Interrupt handler tests the state of the MajorTick latch. If the latch is set, the handler reads and stores the current MET value. This value is provided to the rest of the Flight Software via a utility function.

2) HEXTE Burst Trigger ...... The Timing Module provides a utility function to request a HEXTE burst. When the MinorTick handler is invoked, it tests to see if a burst request is pending. If so, the handler asserts the HEXTE Output Burst Trigger signal. If not, the handler deasserts the signal.

3) Watchdog Timer .... Whenever a MinorTick Interrupt occurs while the MajorTick latch is set, the MinorTick Interrupt handler notifies all threads that a MajorTick pulse has occurred. The **Supervisor Thread** (see **Supervisor Thread Module Specification**, "Theory of Operation") waits for this event, and when woken up, the thread resets the Watchdog Timer.

4) Enable Intervals .... When the Timing Module is configured to start the Front End's Interval Timer (see below), it always asserts the Interval Enable signal during the MinorTick Interrupt preceeding a MajorTick pulse. The Timing Module also ensures that the first enabled interval starts when the MET (~seconds) is an even multiple of 16.

5) Other real-time ops .... The Timing Module also provides the system with the capability to install a single client callback function, which is called whenever a MinorTick Interrupt occurs. This callback function can then handle any other configuration specific real-time operations required by the Flight Software.

The Timing Module provides a set of functions and interrupt handlers which are used to control the Front End's timing circuitry during an observation. The Timing Module provides mechanisms to configure the BinClock duration, set the Timestamp nibble mask, configure and schedule an initial Interval, configure and schedule subsequent Intervals, and shutdown Interval generation.

The Timing Module installs handlers for Interval Interrupts and First Flag Interrupts. It also provides mechanisms used by the Flight Software to install Interval Interrupt and First Flag Interrupt callback functions, which are invoked by the respective interrupt handlers. Interval Interrupts are generated by the hardware whenever an Interval starts. First Flag Interrupts are generated by the hardware whenever the DSP reads an Interval Marker from its Input FIFO and the Interval is configured to generate the interrupt. To run an observation, users of the Timing Module should perform the following operations:

1) Install an Interval Interrupt Client Function

2) Install a First Flag Interrupt Client Function

3) Set the BinClock duration and nibble mask

4) Configure and schedule the first interval

When the first Interval starts, the Interval Interrupt client function should configure and schedule the next Interval. When the First Flag Interrupt occurs, the First Flag Interrupt client function should perform any operations that it requires.

2.5 Data Structures and Constants

This module uses a variety of constants supplied by the modules mentioned in Section 2.3.

2.6 Interrupt Summary

MinorTick Interrupt	This interrupt is generated by the Back End hardware for every MinorTick pulse (32ms). This interrupt is used to perform a variety of Back End real-time operations.
Interval Interrupt	This interrupt is caused by the Front End hardware's Interval generator whenever an Interval tick occurs. This interrupt is used to schedule subsequent intervals.
First Flag Interrupt	This interrupt is caused by the Front End hardware when the DSP reads an interval Marker from its Input FIFO, and the Interval was configured to generate the interrupt.

2.7 Events Summary

The Timing module may notify threads with the following event:

EV_MAJOR .... A MajorTick pulse has just occurred.

2.8 Function List Summary

This module defines the following functions:

TIMINGinit()	Initialize the Timing Module.
TIMINGinterval_client()	Install Interval Interrupt callback function.
TIMINGfirst_flag_client()	Install First Flag Interrupt callback function.
TIMINGminor_client()	Install MinorTick Interrupt callback function.
TIMINGfirst_interval()	Configure and schedule the first interval of a sequence of intervals.
TIMINGnext_interval()	Configure and schedule next interval of a sequence of intervals.
TIMINGstop()	Terminate the interval sequence immediately.
TIMINGtime_bin()	Configure the BinClock duration and TimeBin nibble mask.
TIMINGinitial_time()	Obtain the first MET value read during startup.
TIMINGcurrent_time()	Obtain the current MET value.
TIMINGtrigger_hexte()	Request to have the HEXTE Output Burst Trigger signal pulsed.

3 Data Structures and Constants

3.1 Hardware

3.1.1 CPU Status Register (BREG)

This module reads the following bits in the CPU Control Register:

CPU_STAT_MIN_LAT .... This bit is set to one by the Back End hardware whenever a MinorTick interrupt occurs. This bit, and the interrupt, are cleared using the CPU Pulse Register (see Section 3.1.2).

CPU_STAT_MAJ_LAT .... This bit is set to one by the Back End hardware whenever a MajorTick pulse occurs. This bit is cleared using the CPU Pulse Register (see Section 3.1.2).

3.1.2 CPU Pulse Register (BREG)

This module manipulates the following bits in the CPU Pulse Register:

CPU_PULSE_RST_MAJMIN .... Writing a one to this bit resets the latched versions of the MinorTick and MajorTick pulses and resets the MinorTick interrupt (see Section 3.1.1).

3.1.3 Observation Control Register (FREG)

This module manipulates the following bits in the Observation Control Register:

OBS_CNTL_HXTRIGOUT .... This bit controls the state of the HEXTE Output Burst Trigger signal. If this bit is set to one, the signal is asserted. If this bit is set to zero, the signal is deasserted.
NOTE: HEXTE expects this trigger signal to be pulsed. This module is responsible for controlling the assertion and deassertion of this signal to generate the pulse (see TIMINGtrigger_hexte).

OBS_CNTL_NXT_ACT .... This bit controls whether or not event words are written into the DSP's Input FIFO. This flag takes effect at the start of the next Interval. If set to one, event words will be written into the FIFO at the start of the next Interval. If set to zero, the hardware will stop writing event words into the FIFO at the start of the next Interval.

OBS_CNTL_NXT_FFINTEN .... This bit controls whether or not the hardware should generate a First Flag Interrupt when the DSP reads an Inteval Marker from its Input FIFO. This flag takes effect at the start of the next Interval. If set to one, when the DSP reads an Interval Marker during the next Interval, a First Flag Interrupt will be generated. If set to zero, no First Flag Interrupt will be generated.

OBS_CNTL_DSP_RST .... When set to one, this bit holds the DSP in a reset state, resets the DSP's Input FIFO, and disables event acquisition. While the DSP is held in a reset state, the Back End CPU has access to the Front End's recoder and typer lookup tables. When set to zero, this bit disables CPU access to the lookup tables, and releases the DSP's reset line.

OBS_CNTL_ENINTTK.................. This bit configures the hardware to start an Interval when a MajorTick pulse occurs. If set to one, the next Interval will start upon the occurrence of the next MajorTick pulse. If set to zero, the next Interval will start when the current Interval ends. This bit is only used to schedule the first Interval in a sequence.

3.1.4 Observation Status Register (FREG)

This module reads the following bits in the Observation Status Register:

OBS_STAT_LAT_INTVINT .... This bit is set to one by the Front End hardware whenever an Interval Interrupt occurs. This bit, and the interrupt, are cleared using the Observation Pulse Register (see Section 3.1.5).

OBS_STAT_LAT_FFLAGINT .... This bit is set to one by the Front End hardware whenever a First Flag Interrupt occurs. This bit, and the interrupt, are cleared using the Observation Pulse Register (see Section 3.1.5).

3.1.5 Observation Pulse Register (FREG)

This module manipulates the following bits in the Observation Pulse Register:

OBS_PULSE_CLRLAT_INTV .... Writing a one to this bit resets the latched Interval Interrupt and clears the status bit in the Observation Status Register (see Section 3.1.4).

OBS_PULSE_CLRLAT_FFLAG .... Writing a one to this bit resets the latched First Flag Interrupt and clears the status bit in the Observation Status Register (see Section 3.1.4).

3.2 Constants

3.2.1 TIMING_MIN_INTERVAL (constant value 1048 microticks or ~1ms)

This is the minimum interval duration supported by the Timing Module. This purpose of this constant is to allow the Timing Module to sanity check the parameters. For most modes, intervals scheduled with this duration will be too short and will most likely cause a system panic.

3.2.2 MSG_TIMING_IV_CLIENT_INSTALLED

This message code is passed to UTILpanic() if TIMINGinterval_client() is told to install an Interval Interrupt client function, and a client has already been installed. The message should read something like:

"PANIC: -- TIMINGinterval_client() --Client already installed"

3.2.3 MSG_TIMING_FF_CLIENT_INSTALLED

This message code is passed to UTILpanic() if TIMINGfirst_flag_client() is told to install a First Flag Interrupt client function, and a client has already been installed. The message should read something like:

"PANIC: -- TIMINGfirst_flag_client() --Client already installed"

3.2.4 MSG_TIMING_MINOR_CLIENT_INSTALLED

This message code is passed to UTILpanic() if TIMINGminor_client() is told to install a MinorTick Interrupt client function, and a client has already been installed. The message should read something like:

"PANIC: -- TIMINGminor_client() --Client already installed"

3.2.5 MSG_TIMING_FIRST_IV_ALREADY

This message code is passed to UTILpanic() if an interval is running when TIMINGfirst_interval() is called. The message should read something like:

"PANIC: -- TIMINGfirst_interval() --Already running"

3.2.6 MSG_TIMING_FIRST_IV_DURATION

This message code is passed to UTILpanic() if the interval duration passed to TIMINGfirst_interval() is too short. The message should read something like:

"PANIC: -- TIMINGfirst_interval(%d) --Interval Duration too Short" where %d represents the passed duration value.

3.2.7 MSG_TIMING_NEXT_IV_NOT_RUNNING

This message code is passed to UTILpanic() if an interval is not running when TIMINGnext_interval() is called. The message should read something like:

"PANIC: -- TIMINGnext_interval() --Interval Not Running"

3.2.8 MSG_TIMING_NEXT_IV_DURATION

This message code is passed to UTILpanic() if the interval duration passed to TIMINGnext_interval() is too short. The message should read something like:

"PANIC: -- TIMINGnext_interval(%d) --Interval Duration too Short"
where %d represents the passed duration value.

3.2.9 MSG_TIMING_TIMEBIN_RUNNING

This message code is passed to UTILpanic() if an interval is running when TIMINGtime_bin() is called. The message should read something like:

"PANIC: -- TIMINGtime_bin() --Interval is Active"

3.2.10 MSG_TIMING_TIMEBIN_DURATION

This message code is passed to UTILpanic() if a duration of zero is passed to TIMINGtime_bin(). The message should read something like:

"PANIC: -- TIMINGtime_bin() --Zero duration"

3.2.11 MSG_TIMING_IV_INT_SPURIOUS

This message code is passed to UTILpanic() when the Timing Module's Interval Interrupt handler is invoked, but no Interval Interrupt is pending. The message should read something like:

"PANIC: -- timing_interval_handler() --Spurious Interrupt"

3.2.12 MSG_TIMING_IV_INT_NESTED

This message code is passed to UTILpanic() by the Timing Module's Interval Interrupt handler if an Interval Interrupt occurs while processing the previous Interval Interrupt. The message should read something like:

"PANIC: -- timing_interval_handler() --Nested Interrupt"

3.2.13 MSG_TIMING_FF_INT_SPURIOUS

This message code is passed to UTILpanic() when the Timing Module's First Flag Interrupt handler is invoked, but no First Flag Interrupt is pending. The message should read something like:

"PANIC: -- timing_first_flag_handler() --Spurious Interrupt"

3.2.14 MSG_TIMING_FF_INT_NESTED

This message code is passed to UTILpanic() by the Timing Module's First Flag Interrupt handler if an First Flag Interrupt occurs while processing the previous First Flag Interrupt. The message should read something like:

"PANIC: -- timing_first_flag_handler() --Nested Interrupt"

3.2.15 MSG_TIMING_MINOR_INT_SPURIOUS

This message code is passed to UTILpanic() when the Timing Module's MinorTick Interrupt handlers is invoked, but no MinorTick Interrupt is pending. The message should read something like:

"PANIC: -- timing_minor_handler() --Spurious Interrupt"

3.2.16 MSG_TIMING_MINOR_INT_NESTED

This message code is passed to UTILpanic() by the Timing Module's MinorTick Interrupt handler if a MinorTick Interrupt occurs while processing the previous MinorTick Interrupt. The message should read something like:

"PANIC: -- timing_minor_handler() --Nested Interrupt"

3.2.17 LOG_TIMING_INTERVAL_INTS

This value is passed to UTILlog_housekeeping() by the Timing Module's Interval Interrupt handler to count the occurrence of an Interval Interrupt in Software Housekeeping.

3.2.18 LOG_TIMING_FF_INTS

This value is passed to UTILlog_housekeeping() by the Timing Module's First Flag Interrupt handler to count the occurrence of a First Flag Interrupt in Software Housekeeping.

3.2.19 LOG_TIMING_MINOR_INTS

This value is passed to UTILlog_housekeeping() by the Timing Module's MinorTick Interrupt handler to count the occurrence of a MinorTick Interrupt in Software Housekeeping.

3.2.20 EV_MAJOR

The Timing Module notifies all threads (via the **Thread Manager Module**) of this event whenever the MinorTick Interrupt handler detects a MajorTick pulse.

3.2.21 THREAD_ID_ALL_THREADS

This constant used for the Tnotify() function in place of the thread parameter ID, indicates that the event posted by that call should be posted to all threads in the system (see **Thread Manager Module Specification**).

4 Interrupts

4.1 Interval Interrupt (Level - 1)

The Timing Module installs an interrupt handler for the Interval Interrupt. This interrupt is caused by the hardware whenever an interval starts.

4.2 MinorTick Interrupt (Level - 4)

The Timing Module installs an interrupt handler for the MinorTick Interrupt. This interrupt is caused periodically (~32ms) by the MinorTick pulse.

4.3 First Flag Interrupt (Level - 6)

The Timing Module installs an interrupt handler for the First Flag Interrupt. This interrupt is generated when the DSP reads an Interval Marker from its Input FIFO, and First Flag interrupt generation is enabled.

5 Events

5.1 EV_MAJOR - A MajorTick Pulse has Occurred

The Timing Module notifies all threads of this event whenever a MinorTick Interrupt coincides with a MajorTick pulse.

6 Functions

6.1 TIMINGinit() - Initialize Module

Usage

void TIMINGinit (void);

Description

TIMINGinit() initializes the Timing Module. TIMINGinit() resets any pending MinorTick, Interval, and First Flag interrupts, and asserts the DSP's reset line and disables interval tick generation. It then obtains and stores a valid MET value, inititializes its internal state variables, and installs handlers for the MinorTick, Interval and First Flag interrupts.

To ensure that it reads a valid MET, TIMINGinit() polls the MajorTick bit in the CPU Status Register. When the bit is set for the second time, TIMINGinit() reads the MET value. This operation may take up to two seconds to perform. For the current time to remain valid, interrupts must be enabled within one second of calling this function.

TIMINGinit() MUST be called once before any other operation in this module.

Return Values

NONE - FUNCTION IS VOID.

Caveats

1) TIMINGinit() must be called with the maskable interrupts disabled.

2) This function will hang forever if MajorTick pulses are not showing up in the CPU Status Register.

See Also

Startup Module Specification, TIMINGstop().

6.2 TIMINGinterval_client() - Set/Remove Interval Client

Usage

void TIMINGinterval_client (void (**func*)());

Description

TIMINGinterval_client() sets up the indicated function to be called whenever an interval marker interrupt is generated. The client function will typically call TIMINGnext_interval() and set various flags for the benefit of the science processing mode.

If "*func*" is NULL, then the current interval client function, if set by a previous call to this function, is removed. If no interval client function is present, then this invocation has no effect.

If "*func*" is not NULL, and an existing client function is already installed, TIMINGinterval_client() will issue a MSG_TIMING_IV_CLIENT_INSTALLED panic message and reset the EA, using the **Utility Module** function UTILpanic().

Return Values

NONE - FUNCTION IS VOID

Caveats

1) TIMINGinit() MUST be called before using this function.

2) The client function will be called at the priority level of the Interval Interrupt (the highest used priority). The client function must therefore return as fast as possible. The client function's performance can affect DMA Interrupt latencies, MinorTick Interrupt latencies, First Flag Interrupt latencies, and thread execution performance.

See Also

TIMINGnext_interval(), **Utility Module Specification**.

6.3 TIMINGfirst_flag_client() - Set/Remove First Flag Client

Usage

void TIMINGfirst_flag_client (void (**func*)());

Description

TIMINGfirst_flag_client() sets up the indicated function to be called whenever a First Flag Interrupt is generated. The client function will typically acquire completed data from the DSP for the benefit of the science processing mode.

If "*func*" is NULL, then the current First Flag client function, if set by a previous call to this function, is removed. If no First Flag client function is present, then this invocation has no effect.

If "*func*" is not NULL, and an existing client function is already installed, TIMINGfirst_flag_client() will issue a MSG_TIMING_FF_CLIENT_INSTALLED panic message and reset the EA, using the **Utility Module** function UTILpanic().

Return Values

NONE - FUNCTION IS VOID

Caveats

1) TIMINGinit() MUST be called before using this function.

2) The client function will be called at the priority level of the First Flag Interrupt (the lowest used priority). The client function's performance can affect thread execution performance.

See Also

Utility Module Specification.

6.4 TIMINGminor_client() - Set/Remove MinorTick Client

Usage

void TIMINGminor_client (void (**func*)());

Description

TIMINGminor_client() sets up the indicated function to be called whenever a MinorTick interrupt is generated.

If "*func*" is NULL, then the current minor client function, if set by a previous call to this function, is removed. If no minor client function is present, then this invocation has no effect.

If "*func*" is not NULL, and an existing client function is already installed, TIMINGminor_client() will issue a MSG_TIMING_MINOR_CLIENT_INSTALLED panic message and reset the EA, using the **Utility Module** function UTILpanic().

Return Values

NONE - FUNCTION IS VOID

Caveats

1) TIMINGinit() MUST be called before using this function.

2) The client function will be called at the priority level of the MinorTick Interrupt (level 4). The client function's performance can affect First Flag Interrupt handling latencies and thread execution performance.

See Also

TIMINGnext_interval(), **Utility Module Specification**.

6.5 TIMINGfirst_interval() - Schedule the Initial Interval of a Sequence

Usage

UINT32 TIMINGfirst_interval (UINT32 *duration*,
BOOL *seq_act*, BOOL *first_flag*, UINT8 *ic*);

Description

TIMINGfirst_interval() causes an initial interval to be started at the next MET boundary divisible by 16. This operation returns immediately, after scheduling the specified interval. This function cannot be called while another interval is in progress. If an interval is already running, TIMINGfirst_interval() issues a MSG_TIMING_FIRST_IV_ALREADY panic message and resets the EA.

"*duration*" specifies the duration of the interval, in MicroTick's (there are 2^{20} MicroTicks/MajorTick). If "*duration*" is less than TIMING_MIN_INTERVAL, TIMINGfirst_interval() will issue a MSG_TIMING_FIRST_IV_DURATION panic message, passing "*duration*" as the message argument, and reset the EA.

"*seq_act*" determines whether or not to write event words to the DSP's Input FIFO during the interval. If TRUE, events will be written. If FALSE, events won't be written to the FIFO for the duration of the interval.

"*first_flag*" indicates whether or not to generate a First Flag Interrupt when the DSP reads an Interval Marker from its Input FIFO during the scheduled interval. If TRUE, a First Flag Interrupt will be generated. If FALSE, the hardware won't issue a First Flag Interrupt for the duration of the interval.

TIMINGfirst_interval() uses the lower two bits of "*ic*" as the Interval Constant to use during the scheduled interval. On PCA-EA's, these bits occupy bits 13 and 14 of the processed event words written to the DSP's Input FIFO. On ASM-EA's, these bits occupy bits 12 and 13 of the first of three ASM event words written to the DSP's Input FIFO.

TIMINGfirst_interval() uses the **Utility Module** function UTILpanic() to issue panic messages and reset the EA.

Return Values

The MET at which this interval will start.

Caveats

1) TIMINGinit() MUST be called before using this operation. One or more MajorTick pulses will occur before the interval actually starts.

6.6 TIMINGnext_interval() - Schedule Next Interval

Usage

void TIMINGnext_interval (UINT32 *duration*,
BOOL *seq_act*, BOOL *first_flag*, UINT8 *ic*);

Description

TIMINGnext_interval() schedules the next interval in a sequence. The scheduled interval begins when the currently running interval ends. This function must be called while another interval is in progress. If an interval is not running, this function issues a MSG_TIMING_NEXT_IV_NOT_RUNNING panic message and resets the EA.

"*duration*" specifies the duration of the interval, in MicroTick's. If "*duration*" is less than TIMING_MIN_INTERVAL, TIMINGnext_interval() will issue a MSG_TIMING_NEXT_IV_DURATION panic message, passing "*duration*" as the message argument, and reset the EA.

"*seq_act*" determines whether or not to write event words to the DSP's Input FIFO during the interval. If TRUE, events will be written. If FALSE, events won't be written to the FIFO for the duration of the interval.

"*first_flag*" indicates whether or not to generate a First Flag Interrupt when the DSP reads an Interval Marker from its Input FIFO during the scheduled interval. If TRUE, a First Flag Interrupt will be generated. If FALSE, the hardware won't issue a First Flag Interrupt for the duration of the interval.

TIMINGnext_interval() uses the lower two bits of "*ic*" as the Interval Constant to use during the scheduled interval. On PCA-EA's, these bits occupy bits 13 and 14 of the processed event words written to the DSP's Input FIFO. On ASM-EA's, these bits occupy bits 12 and 13 of the first of three ASM event words written to the DSP's Input FIFO.

TIMINGnext_interval() uses the **Utility Module** function UTILpanic() to issue panic messages and reset the EA.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) TIMINGinit() must have been called before calling this function.

See Also

TIMINGfirst_interval(), TIMINGstop().

6.7 TIMINGstop() - Abort the Interval Sequence

Usage

void TIMINGstop (void);

Description

TIMINGstop() causes interval generation to be immediately terminated, and puts the DSP into a reset state. This disables event accumulation and resets the DSP's Input FIFO.

If this operation is used, there is no way of knowing how far the interval(s) progressed.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) TIMINGinit() MUST be called before using this operation.

2) To start a new interval and reenable data collection, the client must issue a TIMINGfirst_interval() call.

3) If a scheduled first interval has not started by the time this function is called, it will be disabled and never occur.

See Also

TIMINGfirst_interval(), TIMINGnext_interval()

6.8 TIMINGtime_bin() - Specify a Time Bin Duration and Mask

Usage

void TIMINGtime_bin (UINT32 *duration*, UINT8 *nibble_mask*);

Description

TIMINGtime_bin() sets the duration of the Front End's BinClock and, on PCA-EA's, sets the TimeBin nibble mask. This function cannot be called while intervals are running. If an interval is running when this function is called, TIMINGtime_bin() issues a MSG_TIMING_TIMEBIN_RUNNING panic message and resets the EA.

"*duration*" specifies the duration of each TimeBin, in MicroTicks. This value must not be zero. If "*duration*" is zero, TIMINGtime_bin() issues a MSG_TIMING_TIMEBIN_DURATION panic message, passing "*duration*" as the message argument, and resets the EA.

On PCA-EA's, the lower four bits of "*nibble_mask*" specifies the TimeBin nibble mask to use. Each bit in the mask corresponds to each 4-bit nibble in the TimeBin value used in the processed event words. If a bit is set to one, the corresponding nibble's bits are forced to zero. If a bit is set to zero, the corresponding nibble is combined with the processed event word. Bit 0 corresponds to TimeBin bits 0-3, bit 1 corresoponds to bits 4-7, bit 2 corresponds to bits 8-11, and bit 3 corresponds to bits 12-14.

On ASM-EA's, "*nibble_mask*" is ignored by the hardware.

TIMINGtime_bin() uses the **Utility Module** function UTILpanic() to issue panic messages and reset the EA.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) TIMINGinit() MUST be called before using this operation.

See Also

Utility Module Specification

6.9 TIMINGinitial_time() - Return the First MET Value Read

Usage

UINT32 TIMINGinitial_time (void);

Description

TIMINGinitial_time() returns the first MET value (~seconds), which was obtained when TIMINGinit() was called.

Return Values

Initial MET Value

Caveats

1) TIMINGinit() MUST be called before using this operation.

See Also

TIMINGinit()

6.10 TIMINGcurrent_time() - Return the Current Valid MET

Usage

UINT32 TIMINGcurrent_time (void);

Description

TIMINGcurrent_time() returns the current valid MET value (~seconds), which was obtained after the last MajorTick pulse.

Return Values

Current valid MET Value

Caveats

1) TIMINGinit() MUST be called before using this operation.

See Also

TIMINGinit()

6.11 TIMINGtrigger_hexte() - Request HEXTE Output Burst

Usage

void TIMINGtrigger_hexte (void);

Description

TIMINGtrigger_hexte() causes the Timing Module to assert the HEXTE Output Burst Trigger while processing the next MinorTick interrupt, and deassert the signal while processing the subsequent MinorTick interrupt. Since the signal must always be deasserted after being asserted, this function has no effect if the HEXTE Output Burst Trigger is already asserted when this function is called.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) TIMINGinit() MUST be called before using this operation.

2) If two or more EA's are configured to burst HEXTE, it is possible that the pulse width requirements may not be met. It is up to the ground to ensure that no more than 1 EA is configured to burst HEXTE at a time.

See Also

TIMINGinit()

7 Detailed Design Considerations

7.1 Data Flow

This section describes the Level 1 data flow within the Timing Module.

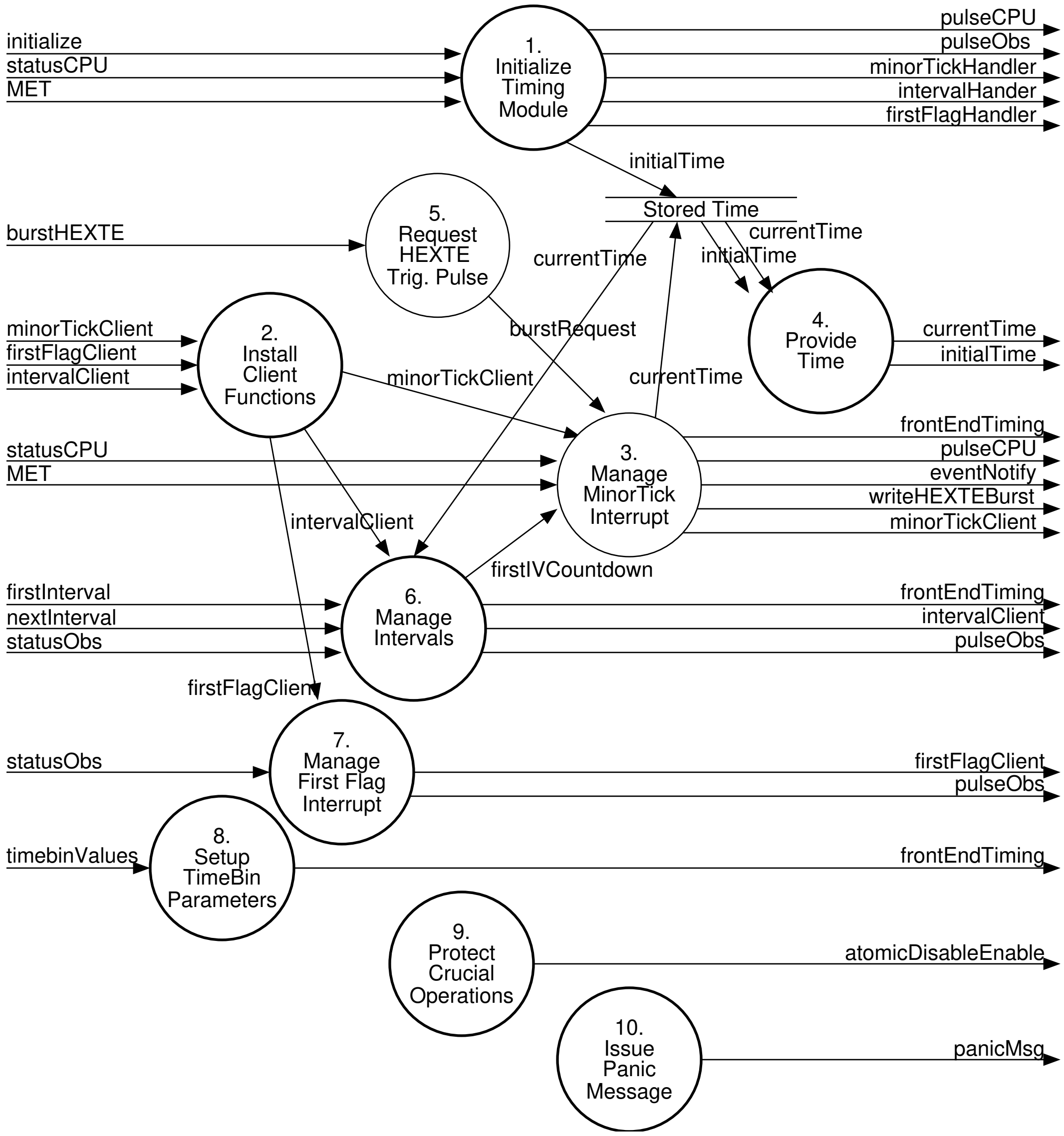

Figure 3 Timing Module Level 1 Data Flow

Figure 3 illustrates the major operations performed by the Timing Module. During system initialization (bubble 1), the Timing Module's initialization routine is invoked (initialize). This function installs interrupt handlers for the MinorTick, Interval, and First Flag Interrupts (minorTickHandler, intervalHandler, firstFlagHandler). The initialization routine also obtains and stores the initial time (initialTime), by polling the MajorTick pulse latch (statusCPU) and reading the MET register when a MajorTick occurs (MET).

Once the Timing Module has been initialized, other Flight Software Modules can install MinorTick, Interval and First Flag interrupt client functions (bubble 2). These client functions (minorTickClient, intervalClient, firstFlagClient) are invoked by their respective interrupt handlers.

The Timing Module maintains the current time as follows. Upon initialization (bubble 1), the Timing Module obtains and stores the current time (currentTime), as described above. Whenever a MinorTick interrupt occurs, the MinorTick Interrupt handler (bubble 3) reads the CPU status register (statusCPU) and checks to see if a MajorTick pulse has occurred. If a MajorTick pulse occurred, the MinorTick Interrupt handler reads the MET register (MET) and stores the value (currentTime). When a request for the current time is made (bubble 4), the Timing Module returns the stored value (currentTime). When a request for the initial time is made (bubble 4), the Timing Module returns the stored initial value (initialTime).

The Timing Module provides a service to set the BinClock duration and TimeBin nibble mask (bubble 8). The Timing Module uses the passed parameters (timebinValues) to configure the Front End hardware (frontEndTiming).

The Timing Module provides services to start and schedule sequences of intervals. When a request is made to schedule the first interval of a sequence (firstInterval), the Timing Module (bubble 6) configures the Front End's interval generator (frontEndTiming), obtains the current time (currentTime) and computes the number of MajorTick pulses that need to occur before enabling the interval generator (firstIVCountdown). The MinorTick interrupt handler (bubble 3) then decrements this count with each MajorTick pulse. On the MajorTick immediately preceeding the desired start time, the MinorTick interrupt handler enables the interval generator (frontEndTiming). The interval generator will then start the interval on the next MajorTick pulse. Whenever an interval starts, the hardware generates an Interval Interrupt. The Interval Interrupt handler (bubble 6) performs some sanity checks (statusObs), then clears the interrupt (pulseObs), and calls the installed Interval Interrupt client function. The called client function should then call the Timing Module to schedule the next interval (nextInterval), or terminate the interval sequence (not shown).

The Timing Module is responsible for handling and dispatching First Flag Interrupts (bubble 7). Whenever a First Flag Interrupt occurs, the First Flag Interrupt handler performs a sanity check (statusObs), resets the interrupt latch (pulseObs) and calls the First Flag Interrupt client function (firstFlagClient).

The MinorTick interrupt handler (bubble 3) is responsible for the assertion and deassertion of the HEXTE Output Burst Trigger signal (writeHEXTEBurst). When another module requests to assert the HEXTE Output Burst Trigger signal (burstHEXTE), the Timing Module records the request (bubble 5) for use by the MinorTick Interrupt handler (bubble 3). When the next MinorTick Interrupt occurs, the handler (bubble 3) asserts the the HEXTE Output Burst Trigger signal (writeHEXTEBurst). Upon the next MinorTick interrupt, following the assertion, the MinorTick handler then deasserts the signal.

The Timing Module performs all crucial operations in an atomic fashion, with interrupts disabled (bubble 9, atomicDisableEnable).

If the Timing Module detects an error, it issues a panic message and resets the EA (bubble 10, panicMsg).

7.2 Control Considerations

7.2.1 MinorTick Interrupt Handling

The Timing Module couples many of its services to the MinorTick Interrupt handler using internal state variables. This coupling is necessary, since requests made by the Flight Software threads are not made in real-time, whereas the services provided have real-time constraints.

For example, the **Science Thread** uses the Timing Module to assert the HEXTE Output Burst trigger because the pulse width requirements on the signal would require the **Science Thread** to either busy-wait for at least 20us (about 40 4 cycle instructions @8MHz) or relinquish control for under 100ms. Busy waiting is not desirable in this situation, because it prevents other threads from running, and prevents the **Science Thread** from continuing its operations. Relinquishing control is unexceptable because it is not completely deterministic as to when the thread will regain control to deassert the signal. As a result, the Timing Module provides the service, using the MinorTick interrupt to time the trigger signal's pulse width.

7.2.2 Interval Scheduling Formula

This section describes the formula used to schedule the first interval in a sequence. This algorithm ensures that, if the MET is being updated by the SDS properly, the first interval will always start when the MET is an even multiple of 16. It also ensures that, if the MET isn't being updated properly, the first interval will start regardless.

In a single atomic step (i.e. with interrupts disabled), the Timing Module reads the current time and computes the number of MajorTicks from the current value to the MET immediately before the next 16 MET boundary. It then computes the actual start time, in terms of the expected MET value.

The number of MajorTicks to pass before scheduling is calculated as follows:

count = 16 - (((current time) + 1) mod 16)

The actual start time is calculated as follows:

start time = (current time) + (count) + 1

NOTE: the "mod 16" operation is merely the least significant four bits of ((current time) + 1).

7.3 Module Structure

7.3.1 Configuration Function Structures

This section describes the structure of the Timing Module functions contained which are responsible for initializing or configuring the Timing Module. Bolded boxes and text refer to function provided by modules other than the Timing Module:

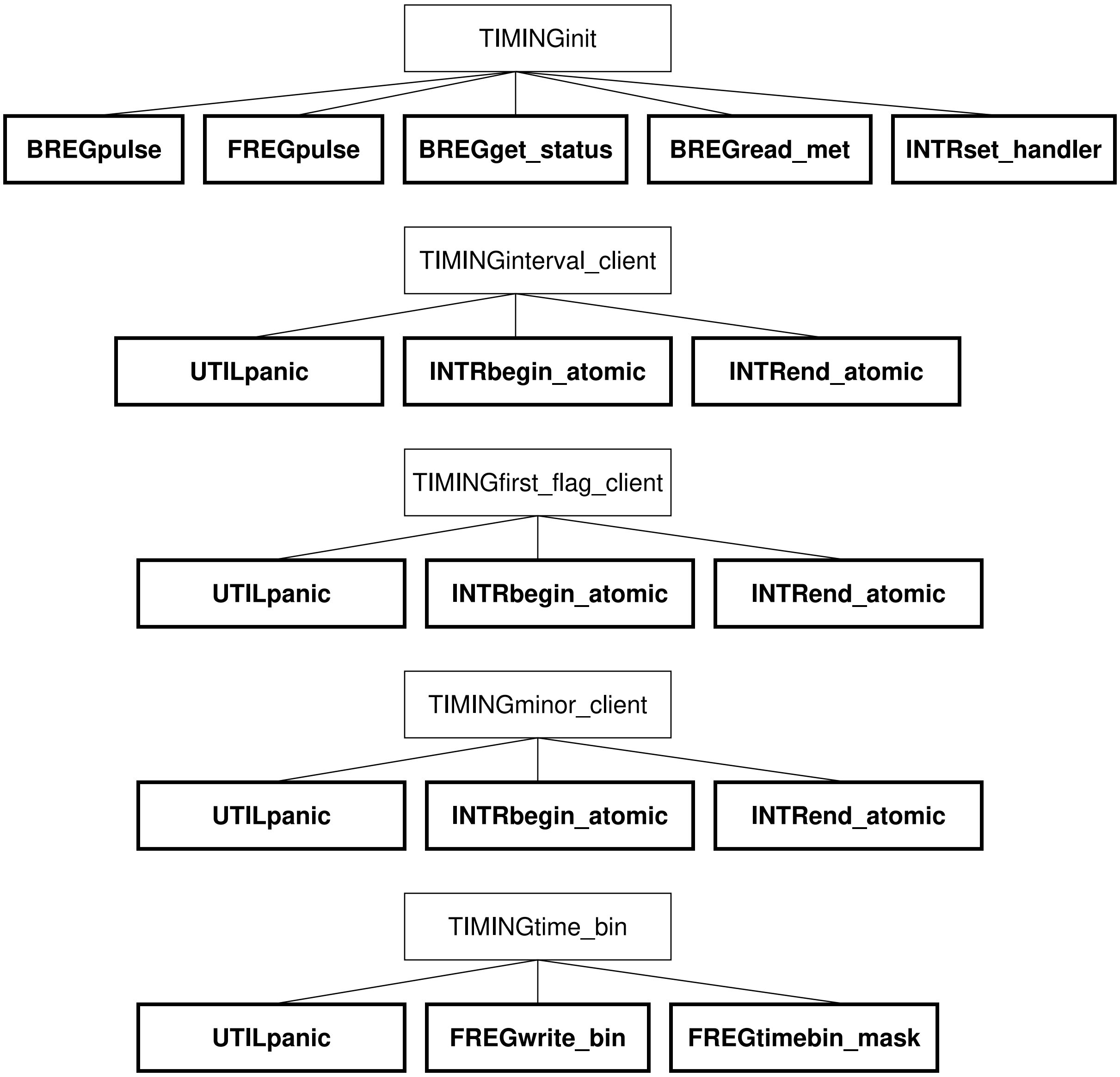

Figure 4 Timing Configuration Function Structures

The following describes the functions shown in Figure 4. The details of each function's operation and arguments are described in the Timing Module's detailed design document.

TIMINGinit .... This function is called by the **Startup Module** during system initialization. This function is responsible for initializing the Timing Module's internal state variables, for obtaining an initial MET value, and for installing the MinorTick, Interval, and First Flag Interrupt handlers. This function uses the **Back End Registers Module** (BREGpulse) to reset latched versions of the MinorTick interrupt and MajorTick pulse. It uses the **Front End Registers Module** (FREGpulse) to reset the latched versions of the Interval and First Flag interrupts. Once the latches have been reset, TIMINGinit uses the **Back End Registers Module** (BREGget_status) to poll the MajorTick pulse latch. After two MajorTick pulses occur, TIMINGinit uses the **Back End Registers Module** (BREGread_met) to read the current MET value. This value is used as the initial time value, and the current time value until the next handled MinorTick interrupt coincident with a MajorTick pulse. Once the MET value has been loaded, TIMINGinit uses the **Interrupt Manager Module** (INTRset_handler) to install its MinorTick, Interval, and First Flag interrupt handlers.

TIMINGinterval_client .... This function is called by the Flight Software Modules to install an Interval Interrupt Client function, which is called whenever an Interval Interrupt occurs. If a client function is already installed, TIMINGinterval_client uses the **Utility Module** (UTILpanic) to issue a panic message and reset the EA. If one isn't installed, or the caller is removing a client, TIMINGinterval_client prevents interrupts while manipulating its internal data structures, using the **Interrupt Manager Module** (INTRbegin_atomic, INTRend_atomic).

TIMINGfirst_flag_client .... This function is called by the Flight Software Modules to install a First Flag Interrupt Client function, which is called whenever a First Flag Interrupt occurs. If a client function is already installed, TIMINGfirst_flag_client uses the **Utility Module** (UTILpanic) to issue a panic message and reset the EA. If one isn't installed, or the caller is removing a client, TIMINGfirst_flag_client prevents interrupts while manipulating its internal data structures, using the **Interrupt Manager Module** (INTRbegin_atomic, INTRend_atomic).

TIMINGminor_client .... This function is called by the Flight Software Modules to install a MinorTick Interrupt Client function, which is called whenever a MinorTick Interrupt occurs. If a client function is already installed, TIMINGminor_client uses the **Utility Module** (UTILpanic) to issue a panic message and reset the EA. If one isn't installed, or the caller is removing a client, TIMINGminor_client prevents interrupts while manipulating its internal data structures, using the **Interrupt Manager Module** (INTRbegin_atomic, INTRend_atomic).

TIMINGtime_bin.............................. This function is called by the Flight Software Modules to set the BinClock duration and TimeBin nibble mask. This function cannot be called while intervals are in progress. If an interval is running when this function is called, or if the function is passed a duration of zero, TIMINGtime_bin issues a panic message and resets the EA, using the **Utility Module** (UTILpanic). If an interval is not running, and the duration is valid, TIMINGtime_bin uses the **Front End Registers Module** (FREGwrite_bin, FREGtimebin_mask) to set the BinClock duration and TimeBin nibble mask.

7.3.2 Interval Control Function Structures

This section describes the structure of the functions, provided by the Timing Module, which manage Interval Timing. Bolded boxes and text refer to function provided by modules other than the Timing Module:

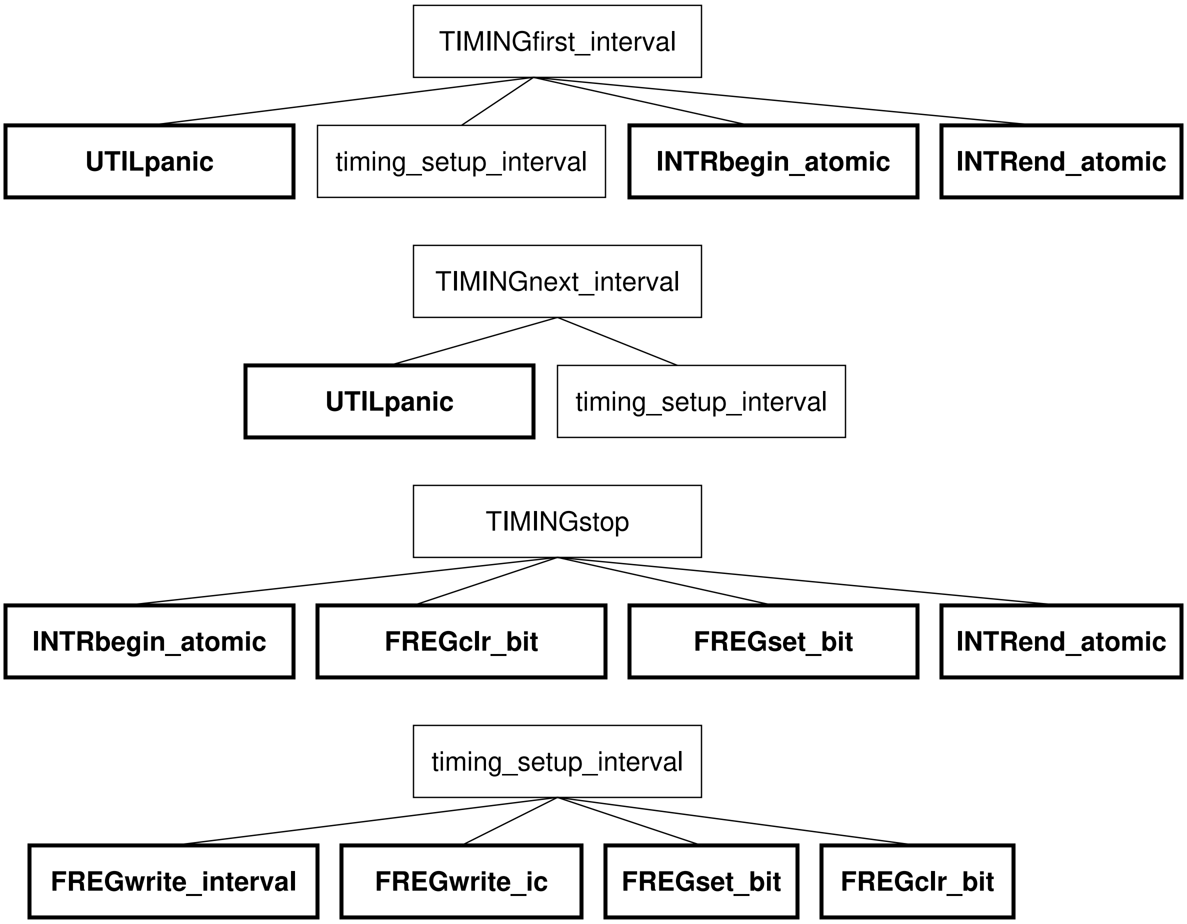

Figure 5 Timing Interval Function Structures

The following describes the functions shown in Figure 5. The details of each function's operation and arguments are described in the Timing Module's detailed design document.

TIMINGfirst_interval .... This function is called by the Flight Software Modules to schedule the first interval in a sequence. TIMINGfirst_interval can only be called when there are no intervals in progress. If an interval is running, or the function is passed illegal parameters, TIMINGfirst_interval issues a panic message and resets the EA, using the **Utility Module** (UTILpanic). If no interval is running and the parameters are valid, TIMINGfirst_interval calls timing_setup_interval to configure the interval generator and flags. It then computes the number of MajorTicks needed before enabling the interval generator, and what the time will be once the interval starts. It performs this operation with interrupts disabled, using the **Interrupt Manager Module** (INTRbegin_atomic, INTRend_atomic). It then returns the computed start time to the caller. The MinorTick Interrupt handler, timing_minor_handler, is then responsible for enabling the interval generator.

TIMINGnext_interval .... This function is called, directly or indirectly, by the installed Interval Interrupt client function. TIMINGnext_interval schedules the next interval in the sequence. If no interval is running, or the passed parameters are invalid, TIMINGnext_interval issues a panic message and resets the EA, using the **Utility Module** (UTILpanic). If an interval is in progress and the parameters are valid, TIMINGnext_interval configures the interval generator for the next interval by calling timing_setup_interval. The new parameters will take effect when the currently running interval ends.

TIMINGstop .... This function is called by timing_interval_handler and by other Flight Software Modules to disable the interval generator and stop event acquisition. TIMINGstop performs its operations as a single atomic step, using the **Interrupt Manager Module** (INTRbegin_atomic and INTRend_atomic). TIMINGstop uses the **Front End Registers Module** (FREGclr_bit) to disable the interval generator. It then uses the **Front End Registers Module** (FREGset_bit) to assert the DSP's reset line. This halts the DSP, disables writes to the DSP's Input FIFO, and resets the FIFO. It then resets some of the module's state variables to ensure that a pending first interval is not started.

timing_setup_interval .... This function is called by TIMINGfirst_interval, and TIMINGnext_interval to set the Interval duration and flags for the next interval. timing_setup_interval uses the **Front End Registers Module** (FREGwrite_interval) to set the interval duration. It then uses the **Front End Registers Module** (FREGwrite_ic) to set the interval constant for the interval. timing_setup_interval then uses the **Front End Registers Module** (FREGset_bit, FREGclr_bit) to set and clear the various single-bit interval flags.

7.3.3 Interrupt Handler and Miscellaneous Function Structures

This section describes the structure of the functions, provided by the Timing Module, which provide the current time, request HEXTE bursts, and handle interrupts. Bolded boxes and text refer to function provided by modules other than the Timing Module and the dashed boxes refer to the installed interrupt client functions:

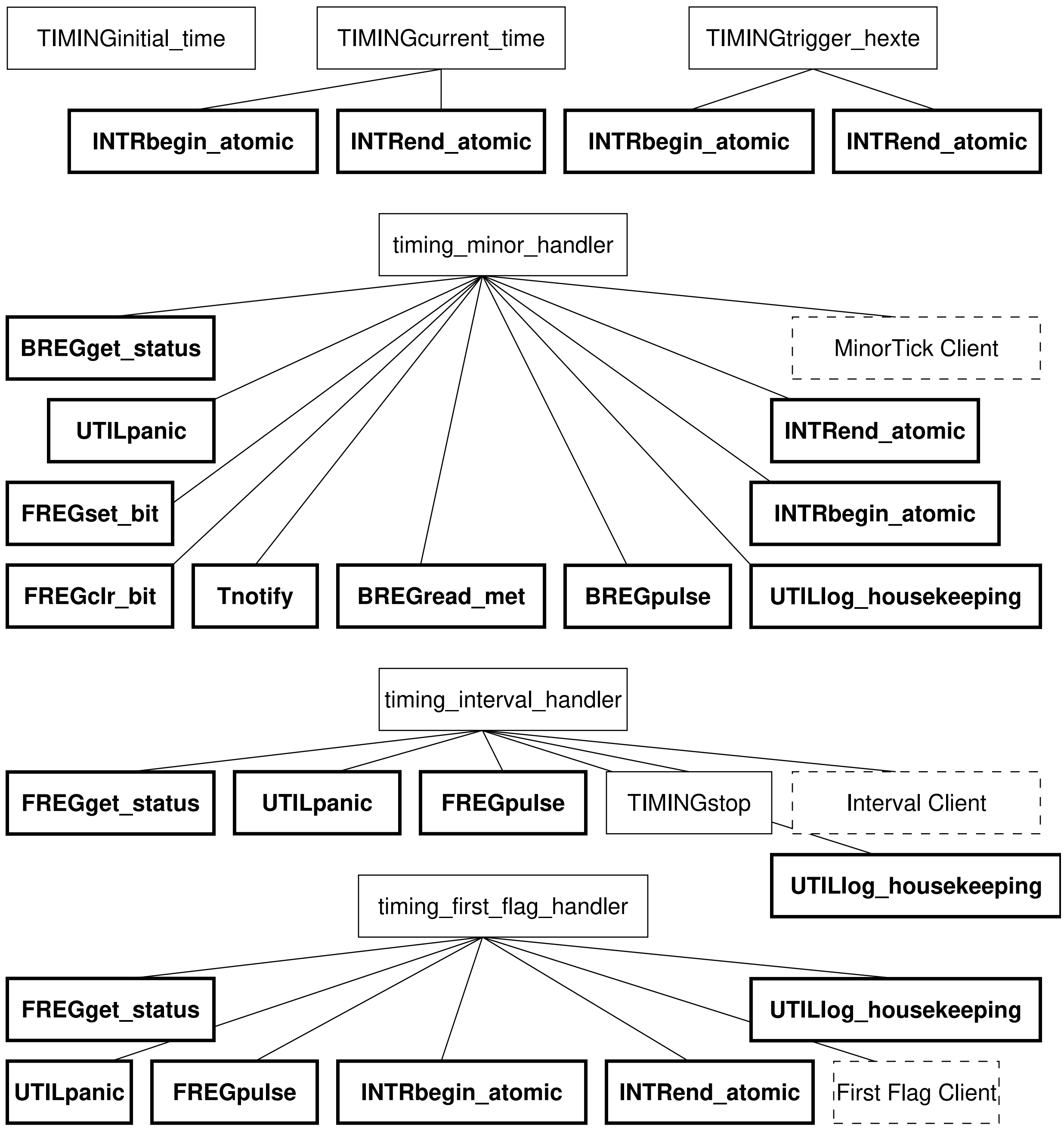

Figure 6 Interrupt Handler and Miscellaneous Structures

The following describes the functions shown in Figure 6. The details of each function's operation and arguments are described in the Timing Module's detailed design document.

TIMINGinitial_time .... This function is called by the Flight Software Modules to obtain the initial time read during system initialization. This function returns the stored initial value.

TIMINGcurrent_time .... This function is called by the Flight Software Modules to obtain the current time. This function obtains the internal time value as an atomic operation, using the **Interrupt Manager Module** (INTRbegin_atomic, INTRend_atomic), and returns the obtained value to the caller.

TIMINGtrigger_hexte .... This function is called by the Flight Software Modules to request a pulse on the HEXTE Output Burst Trigger signal. TIMINGtrigger_hexte sets an internal variable, which causes the MinorTick interrupt handler, timing_minor_tick, to generate the pulse. This function performs its operations as a single atomic step, usign the **Interrupt Manager Module**, (INTRbegin_atomic, INTRend_atomic).

timing_minor_handler .... This function is installed as the MinorTick interrupt handler, by TIMINGinit. Whenever a MinorTick interrupt occurs, this function is invoked. timing_minor_handler starts by reading the CPU Status register, using the **Back End Registers Module** (BREGget_status), and checking the latched version of the MinorTick interrupt. If a MinorTick interrupt didn't occur, timing_minor_handler issues a panic message and resets the EA, using the **Utility Module Specification** (UTILpanic). If the interrupt did occur, timing_minor_handler tests the internal HEXTE burst trigger request flag. If a request is present, timing_minor_handler asserts the HEXTE Output Burst Trigger signal, using the **Front End Registers Module** (FREGset_bit), and clears the request. If no request is present, it clears the signal, using the **Front End Registers Module** (FREGclr_bit). timing_minor_handler then tests the MajorTick pulse latch, using the read CPU Status register. If a MajorTick pulse occurred, it tests the interval enable counter. If the counter is zero, it enables the first interval in a sequence, using the **Front End Registers Module** (FREGset_bit). If the counter is not zero, it decrements the count. During MajorTick processing, timing_minor_handler notifies the threads that a MajorTick pulse occurred, using the **Thread Manager Module** (Tnotify). It then updates the current time by reading the

MET register, using the **Back End Registers Module** (BREGread_met). After handling its internal MinorTick and MajorTick operations, timing_minor_handler resets the MinorTick interrupt latch, and the MajorTick pulse latch, using the **Back End Registers Module** (BREGpulse). It then atomically fetches the MinorTick client function, using the **Interrupt Manager Module** (INTRbegin_atomic, INTRend_atomic) and calls the installed client, MinorTick Client. Once the client returns, timing_minor_handler logs the interrupt using the **Utility Module** (UTILlog_housekeeping). It then re-reads the CPU Status register and tests for another pending MinorTick interrupt. If another interrupt is pending, it issues a panic message and resets the EA. If not, it returns.

timing_interval_handler .... This function is installed as the Interval interrupt handler, by TIMINGinit. Whenever an Interval interrupt occurs, this function is invoked. timing_interval_handler obtains the Front End's Status register, using the **Front End Registers Module** (FREGget_status). It then tests to ensure that an Interval Interrupt really occurred. If an Interval Interrupt didn't occur, it issues a panic message and resets the EA, using the **Utility Module** (UTILpanic). If the status indicates that an Interval Interrupt occurrred, timing_interval_handler resets the latched interrupt, using the **Front End Registers Module** (FREGpulse). If TIMINGstop() has not been called, and an Interval Client function has been installed, it then calls the installed Interval Interrupt Client function, Interval Client. Once the client returns, timing_interval_handler logs the occurance of the interrupt by calling the **Utility Module** (UTILlog_housekeeping). timing_interval_handler then re-reads the Front End Status register. If another Interval Interrupt is pending, timing_interval_handler issues a panic message and resets the EA. If not, timing_interval_handler returns.

timing_first_flag_handler .... This function is installed as the First Flag interrupt handler by TIMINGinit. Whenever a First Flag interrupt occurs, this function is invoked. timing_first_flag_handler obtains the Front End's Status register, using the **Front End Registers Module** (FREGget_status). It then tests to ensure that a First Flag Interrupt really occurred. If an First Flag Interrupt didn't occur, it issues a panic message and resets the EA, using the **Utility Module** (UTILpanic). If the status indicates that a First Flag Interrupt occurrred, timing_first_flaghandler resets the latched interrupt, using the **Front End Registers Module** (FREGpulse). It then calls the installed First Flag Interrupt Client function, First Flag Client. Once the client returns, timing_first_flag_handler logs the interrupt using the **Utility Module** (UTILlog_housekeeping). It then re-reads the Front End Status register. If another First Flag Interrupt is pending, timing_first_flag_handler issues a panic message and resets the EA. If not, timing_first_flag_handler returns.

8 Requirements Traceability

8.1 Science Requirements

The Timing Module has a direct impact on the performance of the Flight Software. The time taken to handle Interval Interrupts and First Flag Interrupts directly affects the throughput requirements, as specified in the **PCA Science Requirements Specification**, and the **ASM Science Requirements Specification**. The Interval Interrupt handling has a special impact on Pulsar Fold Mode, in that the speed at which intervals are scheduled affects the maximum fold frequency.

The MinorTick interrupt handling directly affects the observation scheduling requirements in that it ensures that the first interval of an observation always starts on a 16 MET (~seconds) boundary. It also address the HEXTE Output Burst Trigger signal pulse width requirements, specified in the **XTE EDS Hardware Functional Description and Requirements**.

Please refer to the **Requirements Traceability Matrix** to determine which science requirements are directly affected by this module.

9 Testing Considerations

9.1 Unit Testing

To perform unit tests on this module, the following equipment is required:

a. An Event Analyzer equipped with the Intermetrics ROM Monitor Debug Board

b. A workstation running the Intermetrics debugger

c. An RS-232 serial cable between the ROM Monitor's serial port and the workstation

d. A logic analyzer

To unit test this module, the tester must link and locate this module's functions to some EA RAM memory location. Using a resident monitor/debugger, the tester should first verify the operation of all functions internal to this module, and then verify the interactions between all functions internal to the module. These tests should verify all branches, calls, and legal and illegal parameters. The tester should then verify all calls made to functions outside of this module, and ensure that the passed arguments are always within the limits specified by the corresponding module specifications. The tester should also verify that the caller always handles values returned by these functions in an appropriate manner.

In addition to these tests, the tester should perform the following:

a. Verify that the data coupling between the TIMINGtrigger_hexte function and the MinorTick interrupt handler, timing_minor_handler, is correct. Ensure that after a request is made, if the trigger is not asserted, the trigger is asserted at the next MinorTick interrupt. Then verify that the trigger is always deasserted on the subsequent MinorTick interrupt, even if a second request is made.

b. Verify interval start time logic. Ensure that the first interval in a sequence always starts on a 16 MET (~seconds) boundary.

c. Verify TIMINGcurrent_time and ensure that returned time reflects the current MET value. These tests should include calls to TIMINGcurrent_time from interrupt handlers which interrupt the MinorTick interrupt handler. Verify that TIMINGinitial_time provides the correct initialization time.

d. Ensure that pending interrupts at a given level, which occur after the respective latch has been reset but before the handler retests the latch, result in a panic message and reset.

9.2 Integration Testing

For integration tests, the following equipment is required:

a. An Event Analyzer

b. A populated System Manager board

c. A logic analyzer

d. Hardware which can:

   1. Drive the MicroTick, MinorTick and MajorTick clock signals (S/C Simulator)

   2. Receive telemetry packets from an EA (S/C Simulator)

   3. Block telemetry packets from the EA (i.e. stop receiving packets).

The integration tests should verify the operation of the Timing Module in conjunction with the other FS modules. These tests should include:

a. Verify that the proper MET value is obtained upon reboot.

b. Verify that the first interval in a sequence starts on a 16 MET (~seconds) boundary.

c. Verify the Interval Duration and flags for several chosen intervals.

d. Verify the operation of the Front End event processing and time-stamp circuitry, in conjunction with the scheduled intervals.

e. Verify the HEXTE Output Trigger signal pulse width. It must always be greater than 20us and less than 100ms.

f. Measure the performance of the Interval and First Flag Interrupt processing. This information is used when determining the minimum interval duration, and Interval and First Flag interrupt overhead.

Appendix A

A.1 Items Under Review

This section lists items and issues which need to be resolved:

Document Reference	Section	Resolution Expected by
NONE		