Thread Manager Module Specification

James Francis

MIT 64-30110.1701

Rev. D

March 5, 1993

Contents

Figures

1 Introduction

1.1 Scope

This document describes the components that make up the Thread Manager Software Module (THREAD). This module implements a non-preempting, multi-threaded executive.

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. It also assumes that the reader is familiar with the concepts of multi-tasking and multiple threaded systems.

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

1.2 References

ISBN 1-55512-055-5 1987	"80286 and 80287 Programmer's Reference Manual"
January 19, 1990 Version 2.32	"Intermetrics Tookit User's Manual, 8086 Family, Release 6.0, Unix"
MIT 64-30102 Rev. E.......................	"Flight Software Global Design"
MIT 64-30110.1201 Rev. D	"Interrupt Manager Module Specification" (INTR)
MIT 64-30110.2001 Rev. C	"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. Typically, a value of zero indicates FALSE, any other value, TRUE.

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 Thread Manager does not appear directly in this data flow, but has a scope which affects items indicated by the shaded areas:

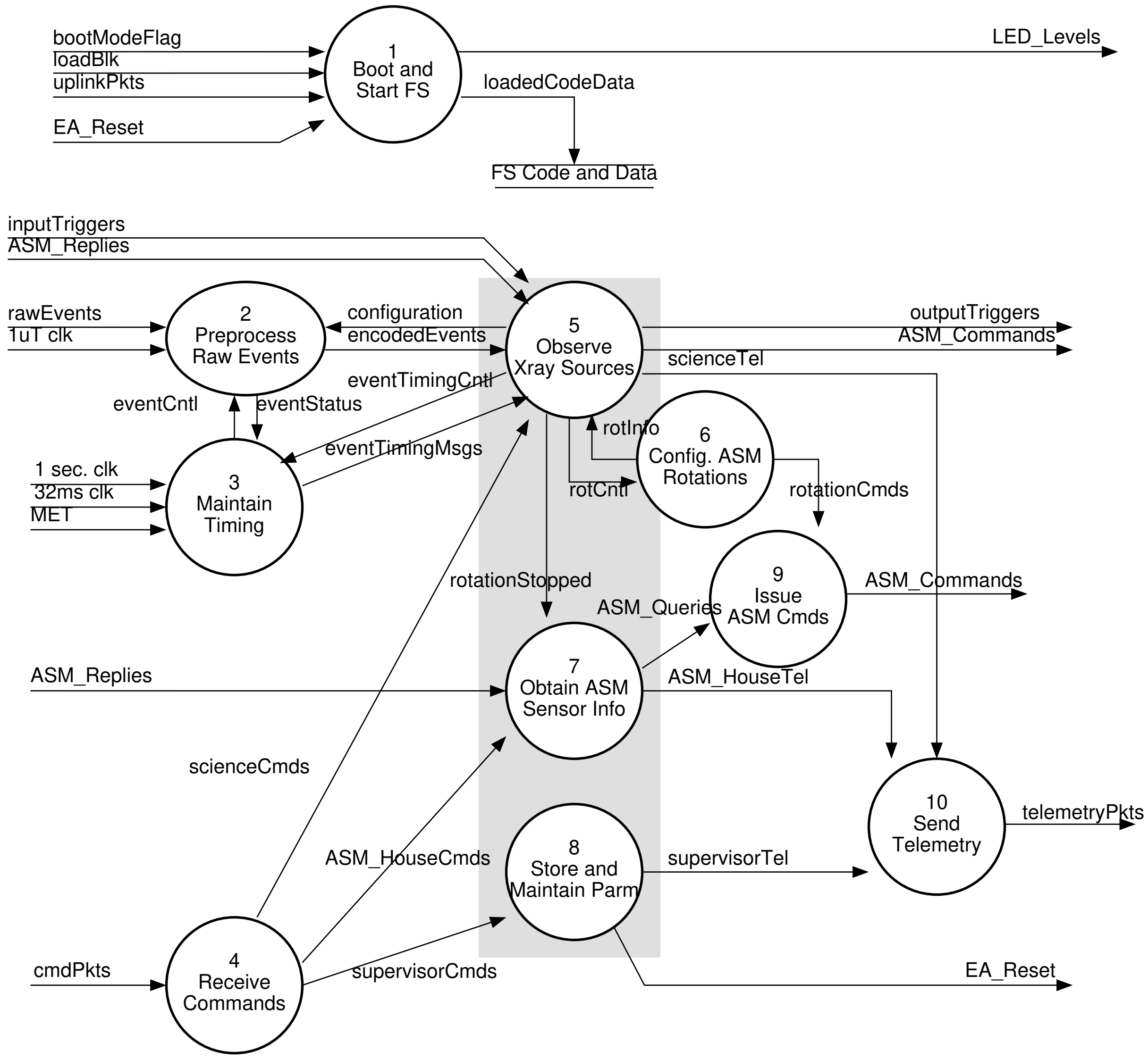

Figure 1 Module Perspective

1.5 Document Overview

Sections 1 and 2 provide an overview of the Thread Manager Module, summarizing the purpose, services, context, interfaces of the module. Section 3 describes the interface to the 80286 registers, and all software structures and constants visible to modules outside of the Thread Manager. Section 4 provides an overview of the operation of the Thread Manager. Section 5 describes the functions provided to other modules by the Thread Manager. Section 6 describes some of the implementation details of the module. Section 7 describes which requirements are met by the module. Section 8 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 the Thread Manager Module is to provide the Flight Software with a simple, non-preemptive, multi-threaded executive. It is responsible for thread creation, scheduling, context switching, and event notification. It provides thread specific stack contexts, register values and execution flow. It does not provide thread specific RAM data space, preemptive scheduling or thread-prioritized scheduling.

2.2 Services

The Thread Manager Module provides the following services:

1) Thread Creation .... Initialize a thread context control structure.
2) Scheduling .... Maintains an ordered list of threads available to run.
3) Context Switching .... Performs stack and register context switching between threads, whenever the running thread waits or yields.
4) Event Notification .... Notifies a thread or threads whenever events are posted.

2.3 Module Context

The following illustrates the Thread Manager Module with respect to the EDS system:

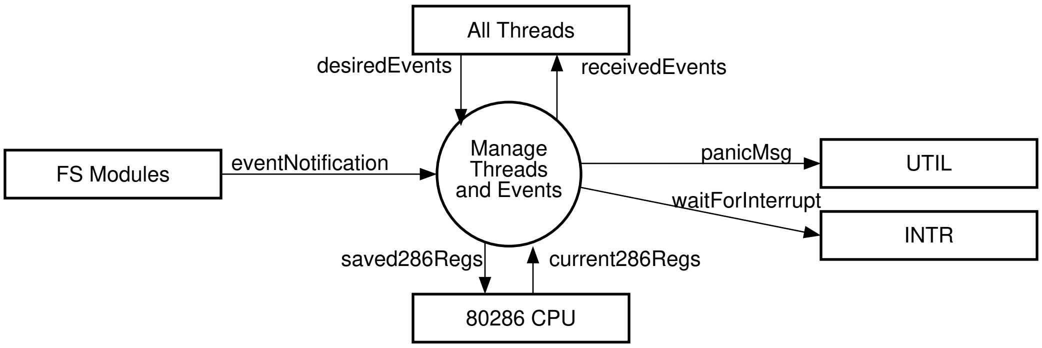

Figure 2 Thread Manager Module Context Diagram

The Thread Manager maintains up to 8 80286 register and stack contexts. Each context is associated with a thread. Threads relinquish control by either waiting for one or more event notifications (desiredEvents), or by yielding. When a thread reliquishes control, some of the 80286 registers are saved in the thread's context (current286Regs), and the next thread is activated by restoring the registers saved when it relinquished control (saved286Regs). If a thread waits for events, the thread manager supplies the thread with the received events (receivedEvents) when it returns control to the thread. Event notifications (eventNotification) can be produced by any Flight Software module. If, at any point, the Thread Manager determines that the system has been corrupted, it issues a panic message (panicMsg) to the **Utility Module**.

2.4 Theory of Operation

See Section 4.

2.5 Interface Summary

The Thread Manager Module directly reads and writes certain 80286 registers. When all threads are waiting for events, it uses the **Interrupt Manager Module** to wait for an interrupt to occur. If the Thread Manager Module detects corruption of the EA's memory, it uses the **Utility Module** to issue panic messages and reset the EA.

2.6 Data Structures and Constants Summary

The Thread Manager must know about the register set for the 80286. Refer to Section 3.1 or the **80286 and 80287 Programmer's Reference Manual** for a description of this register set.

Thread and event identifiers are global system constants. They are not needed by the Thread Manager, but must conform to the limits specified in this document (see Sections 3.2.1 and 3.2.2).

2.7 Event Summary

The Thread Manager can support up to 32 events. The Thread Manager uses event notifications to schedule waiting threads. The Thread Manager itself doesn't produce any events.

2.8 Scheduling Summary

The Thread Management module implements a non-preemptive (ie. cooperative) scheduling algorithm. Threads must voluntarily give up control. Section 4 describes the operation of this algorithm.

2.9 Function List Summary

The Thread Management module consists of the following functions:

Tinit()	- Initializes the state of the thread manager.
Tcreate()	- Creates a new thread.
Texists()	- Indicates whether or not a given thread exists.
Tmulti()	- Starts multi-threaded execution.
Twait()	- Yields a thread or suspends until a selected event occurs.
Tclear_events()	- Clears any pending event notifications for a given thread.
Tnotify()	- Notifies a thread of an event's occurrence.

3 Data Structures and Constants

3.1 Hardware

3.1.1 CPU Registers

This section supplies information necessary to define the interface of this module to the Flight Hardware and compiler generated code. It lists the CPU registers defined by the 80286. More detail of the operation of the 80286 can be found in the 80286 Reference Manual. The register set is as follows:

CS	Code Segment Register.	**Save,** used to resume ***thread*** execution.
IP	Instruction Pointer.	**Save,** used to resume ***thread*** execution.
SS	Stack Segment Register.	**Save,** it is different for each ***thread.***
SP	Stack Pointer.	**Save,** is different for each ***thread.***
DS	Data Segment Register.	In large ***memory model***, this never changes.
ES	Extra Segment Register.	***compiler*** dependent.
BP	General Register (base).	***compiler*** dependent.
SI	Source Index Register.	***compiler*** dependent.
DI	Destination Index Register.	***compiler*** dependent.
AX	General Register (multiply/divide).	***compiler*** dependent, usually return values.
DX	General Register (multiply/divide).	***compiler*** dependent.
BX	General Register (base).	***compiler*** dependent.
CX	General Register (count).	***compiler*** dependent.
F	CPU Status Flags.	***never counted on*** across function calls.
MSW	Machine Status Word.	real mode, should ***never change***.

The Flight Software is built using the Intermetrics 8086 Family "C" Compiler. This compiler specifies which registers need to be saved across a function call (please see **Intertools Toolkit User's Manual**, Appendix C, Subroutine Linkage). The Thread Manager saves and restores the following registers; SI, CX, SS, BP, CS, IP, and SP.

3.2 Constants

3.2.1 THREAD_MAX_THREADS (8)

The Thread Manager can support up to 8 threads. The threads are referred to using a *thread_id*, which can take on values from 0 through 7.

3.2.2 THREAD_MAX_EVENTS (32)

The Thread Manager can support up to 32 distinct events. The events are referred to using an event mask. Each bit in the mask corresponds to a unique event. A single mask can refer to all events in the system. For example, 0x0090000061 refers to events 0, 5, 6, 28 and 31.

3.2.3 THREAD_NUM_STACK_GUARD (128)

The stack must include this number of additional 16-bit words to allow the Thread Manager to detect stack underflow and overflow corruption.

3.2.4 THREAD_ID_ALL_THREADS (8)

The function Tnotify() uses this value to notify all threads in the system of a particular event.

3.2.5 MSG_TCREATE_NULL_FUNC

This message code is passed to UTILpanic() if the function argument to Tcreate() is NULL. The message should read something like:

"PANIC: ---Tcreate() -- function argument is NULL"

3.2.6 MSG_TCREATE_BAD_ID

This message code is passed to UTILpanic() if the thread id argument to Tcreate() is invalid. The message should read something like:

"PANIC: ---Tcreate([%d]) -- Invalid thread id"
where %d is the thread id that was passed to Tcreate().

3.2.7 MSG_TCREATE_ID_USED

This message code is passed to UTILpanic() if the thread id argument to Tcreate() is already in use. The message should read something like:

"PANIC: ---Tcreate([%d]) -- Thread id in use"
where %d is the thread id that was passed to Tcreate().

3.2.8 MSG_TCREATE_NULL_STACK

This message code is passed to UTILpanic() if the stack argument to Tcreate() is NULL. The message should read something like:

"PANIC: ---Tcreate() -- NULL stack pointer"

3.2.9 MSG_TCREATE_SMALL_STACK

This message code is passed to UTILpanic() if the stack length argument to Tcreate() is too small. The message should read something like:

"PANIC: ---Tcreate([%d]) -- Stack too small"
where %d is the stack size that was passed to Tcreate().

3.2.10 MSG_TCREATE_CORRUPTED

This message code is passed to UTILpanic() if Tcreate() detects corruption of the EA's RAM.

"PANIC: ---Tcreate() -- System Corrupted"

3.2.11 MSG_TEXISTS_BAD_ID

This message code is passed to UTILpanic() if the thread id argument to Texists() is invalid. The message should read something like:

"PANIC: ---Texists([%d]) -- Invalid thread id"
where %d is the thread id that was passed to Texists().

3.2.12 MSG_TMULTI_NO_THREADS

This message code is passed to UTILpanic() if Tmulti() is called, but no threads have been created. The message should read something like:

"PANIC: ---Tmulti() -- No threads"

3.2.13 MSG_TMULTI_CORRUPTED

This message code is passed to UTILpanic() if Tmulti() detects corruption of the EA's RAM.

"PANIC: ---Tmulti() -- System Corrupted"

3.2.14 MSG_TMULTI_THREAD_RETURN

This message code is passed to UTILpanic() if a thread ever returns. The message should read something like:

"PANIC: ---Tmulti([%d]) -- Thread attempted to return"
where %d is the id of the thread that was active when the

return was attempted.

3.2.15 MSG_TMULTI_RETURN

This message code is passed to UTILpanic() if the scheduler ever returns control to the main multitasking start routine. The message should read something like:

"PANIC: ---Tmulti() -- Scheduler returned"

3.2.16 MSG_TWAIT_CORRUPTED

This message code is passed to UTILpanic() if Twait() detects corruption of the EA's RAM.

"PANIC: ---Twait() -- System Corrupted"

3.2.17 MSG_TWAIT_NOT_MULTITASKING

This message code is passed to UTILpanic() if Twait() detects that multitasking has not yet started.

"PANIC: ---Twait() -- Not Multitasking"

3.2.18 MSG_TCLEAR_CORRUPTED

This message code is passed to UTILpanic() if Tclear_events() detects corruption of the EA's RAM.

"PANIC: ---Tclear_events() -- System Corrupted"

3.2.19 MSG_TCLEAR_NOT_MULTITASKING

This message code is passed to UTILpanic() if Tclear_events() detects that multitasking has not yet started.

"PANIC: ---Tclear_events() -- Not Multitasking"

3.2.20 MSG_TNOTIFY_BAD_ID

This message code is passed to UTILpanic() if the thread id argument to Tnotify() is invalid. The message should read something like:

"PANIC: ---Tnotify([%d]) -- Invalid thread id"

where %d is the thread id that was passed to Tnotify().

3.2.21 MSG_TSCHED_STACK_CORRUPTED

This message code is passed to UTILpanic() if the internal scheduling function, t_sched, detects that a thread's stack has been corrupted. The message should read something like:

"PANIC: ---t_sched() -- Thread %d's stack is corrupted"
where %d is the id of the thread whose stack was corrupted.

3.2.22 MSG_TSCHED_RESTORE_RETURNED

This message code is passed to UTILpanic() if the internal scheduling function, t_sched, detects that the restore context function, t_restore_state, ever returns. The message should read something like:

"PANIC: ---t_sched() -- t_restore_state returned"

3.2.23 MSG_TNEXT_THREAD_NO_THREADS

This message code is passed to UTILpanic() by the internal thread search function, t_next_thread. t_next_thread issues this panic if, when there are no threads to run, it detects that there are also no waiting or yielded threads. The message should read something like:

"PANIC: ---t_next_thread() --No running, waiting or yielded threads"

3.2.24 MSG_TSCAN_WAITING_WRONG_LIST

This message code is passed to UTILpanic() by the internal waiting list scan function, t_scan_waiting. t_scan_waiting issues this panic if it detects that a thread is inappropriately contained in the waiting list. The message should read something like:

"PANIC: ---t_scan_waiting() --Thread %d should not be in the waiting list"
where %d is the id of the thread which was misplaced.

4 Operation

4.1 Scheduling

The Flight Software maintains threads by using a reference to the currently running thread, and by using three different lists, the **runable list**, the **satisfied list** and the **waiting list**. When the running thread relinquishes control, by either yielding or by waiting for an event, the Flight Software gives control to the first thread in the **satisfied list**. If the **satisfied list** is empty, then the Flight Software gives control to the first thread in the **runable list**. A thread yields by performing a wait, without specifying any events. When a thread yields, it is appended to the end of the **runable list**. When it waits for an event, it is appended to the end of the **waiting list**. As threads in the **waiting list** are notified that one or more of the waited for events have occurred, they are appended to the end of the **satisfied list**. If two threads are notified at the same time, the threads are appended to the **satisfied list** in the order in which they were appended to the **waiting list** (ie. in the order in which they "waited"). If all threads are in the **waiting list**, the scheduler will wait for an interrupt to notify one or more threads. Figure 3 shows the flow of threads through the system.

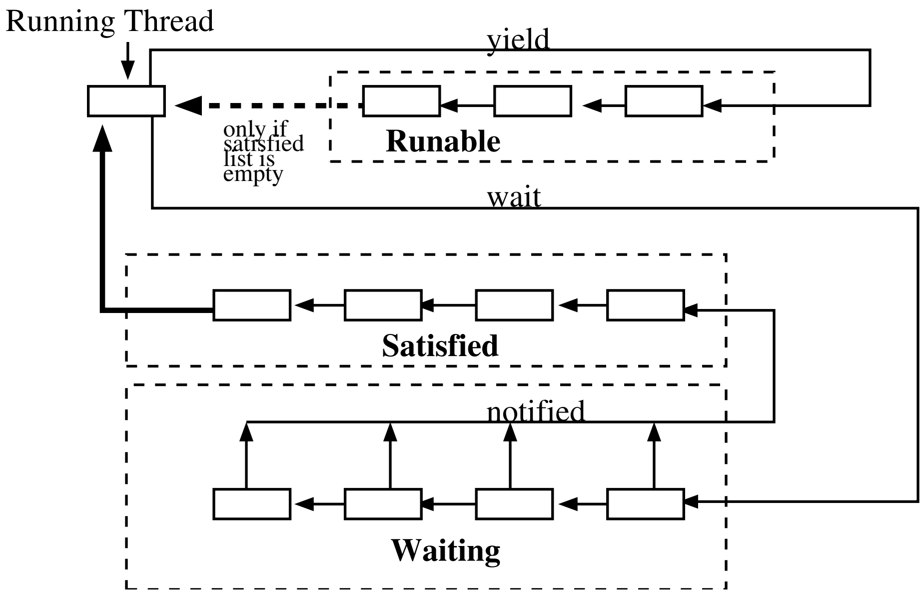

Figure 3 Scheduling Diagram

5 Functions

5.1 Tinit() - Initialize Module

Usage

void Tinit (void);

Description

Tinit() initializes the state of the Thread Manager Module. Initially, there are no thread definitions. The initialization module should call Tcreate() after calling this function to construct all the threads that it needs, and then call Tmulti() to start multi-tasking the defined threads. This function must be called before any other functions in this module.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) This function must be called before any other functions in this module.

See Also

N/A

5.2 Tcreate() - Create a Thread

Usage

void Tcreate (void (**func*) (void), UINT8 *thread_id*,
UINT16 **stack*, UINT16 *stack_len*);

Description

Tcreate() constructs a new thread. "*func"* defines the initial function to invoke when the thread is started. "*thread_id*" specifies the id of the created thread, and must have a value of 0 through 7. "*stack*" points to the stack that the new thread should use, and "*stack_len*" specifies the number of 16-bit words in the stack. The caller is responsible for declaring the memory pointed to by "*stack*". Upon creation of the thread, this function will return. The newly created thread is initially in a "runable" state. Once it is run, it retains control until it Twait()'s.

The following scenarios cause this function to call UTILpanic() with the shown message codes and arguments:

func is NULL	MSG_TCREATE_NULL_FUNC	NONE
thread_id > 7	MSG_TCREATE_BAD_ID	*thread_id*
thread_id already used	MSG_TCREATE_ID_USED	*thread_id*
stack is NULL	MSG_TCREATE_NULL_STACK	NONE
stack_len <= NUM_STACK_GUARD	MSG_TCREATE_SMALL_STACK	*stack_len*
System Corruption Detected	MSG_TCREATE_CORRUPTED	NONE

Return Values

NONE - FUNCTION IS VOID

Caveats

1) Tinit() must be called before calling this function.

2) Make sure that you use a large enough stack. Take into account calling depth, stack variables, nested interrupts and the THREAD_NUM_STACK_GUARD.

3) This function may panic the EA if corruption is detected, or if the passed arguments are invalid.

See Also

UTILpanic().

5.3 Texists() - Test to see if a thread exists

Usage

BOOL Texists(UINT8 *thread_id*);

Description

Texists() tests to see if a thread with "*thread_id*" already exists. If so, this function returns TRUE, otherwise, it returns FALSE. "*thread_id*" must be a value from 0 to 7, inclusive.

If *"thread_id*" is greater than 7, this function will call UTILpanic() with a MSG_TEXISTS_BAD_ID message code, and "*thread_id*" as the message argument.

Return Values

TRUE	- If a thread with "*thread_id*" exists.
FALSE	- The thread doesn't exist.

Caveats

1) Tinit() must be called before calling this function.

2) This function will panic the EA if "*thread_id*" is invalid.

See Also

UTILpanic().

5.4 Tmulti() - Start Executing Threads

Usage

void Tmulti(void);

Description

Tmulti() causes the Thread Manager to start executing the threads, defined by Tcreate(), in a multi-tasking mode. This function does not return.

If no threads exist, this function calls UTILpanic() with a MSG_TMULTI_NO_THREADS message code.

If, at some point, all threads are waiting for events to occur, the scheduler will call INTRwait(), to wait for an interrupt to occur.

If this function detects that the Thread Manager's state is corrupted, this function will call UTILpanic(), with a MSG_TMULTI_CORRUPTED panic message.

Once running, if a thread ever returns, this function will call UTILpanic() with a MSG_TMULTI_THREAD_RETURN message code, and the current thread id as the message argument.

Return Values

NONE - FUNCTION SHOULD NEVER RETURN!!!

Caveats

1) Tinit() must be called before calling this function.

2) Cannot be called from within an interrupt handler.

3) Cannot be called in between unbalanced INTRbegin_atomic() and INTRend_atomic() calls.

4) This function may cause the EA to reset if the system integrity is suspect.

See Also

INTRwait(), UTILpanic()

5.5 Twait() - Wait for One or More Events

Usage

UINT32 Twait(UINT32 *ev_mask*);

Description

Twait() causes the calling thread to suspend until one or more of the events in "*ev_mask*" have occurred. "*ev_mask*" is a 32-bit mask, in which each bit corresponds to one event. Once the thread regains control, this function will return a mask of the events which have occurred. If "*ev_mask*" is zero, then this function performs a yield, and, when it regains control, returns a zero.

Each call to Twait() causes pending events specified by "*ev_mask*", to be cleared. If the events are already satisfied upon calling Twait(), this function will suspend until all previously "satisfied" threads have run (this prevents a single continuously satisfied thread from "blocking" other satisfied threads). The following lists the various scenarios at the time of a call to Twait():

mask is zero, events pending	Perform yield. Pending events preserved.
mask matches some pending	Suspend until all previously "satisfied" threads have run. When run, matching events are cleared and non-matching events are preserved.
mask doesn't match any pending......	Suspend. When matching event occurs, put thread in "satisfied" list. When it runs clear matching events, preserve non-matching events.

To clear any pending events before calling Twait(), use Tclear_events(). If this function detects that the Thread Manager's state is corrupted, this function will call UTILpanic(), with a MSG_TWAIT_CORRUPTED panic message. If multitasking has not started (i.e. Tmulti() has not been called yet), this function will call UTILpanic(), with a MSG_TWAIT_NOT_MULTITASKING panic message.

Return Values

unsigned, 32 bit mask of events that were waited for and have occurred, or zero, if "*ev_mask*" is zero.

Caveats

1) This function must be called from within a running thread.

2) Cannot be called in between unbalanced INTRbegin_atomic() and INTRend_atomic() calls.

See Also

UTILpanic(), INTRbegin_atomic(), INTRend_atomic().

5.6 Tclear_events() - Clear Pending Events

Usage

void Tclear_events(UINT32 *clr_mask*);

Description

Tclear_events() clears any pending events specified in "*clr_mask*." Pending events not specified in "*clr_mask*" are not affected. This function affects pending events for the calling thread only. Events pending for other threads are not affected.
If "*clr_mask*" is zero, this function has no effect.

If this function detects that the Thread Manager's state is corrupted, this function will call UTILpanic(), with a MSG_TCLEAR_CORRUPTED panic message.

If multitasking has not started (i.e. Tmulti() has not been called yet), this function will call UTILpanic(), with a MSG_TCLEAR_NOT_MULTITASKING panic message.

Return Values

NONE - FUNCTION IS VOID.

Caveats

1) This function must be called from within a running thread.

2) If "*clr_mask*" is zero, this function just returns.

See Also

UTILpanic().

5.7 Tnotify() - Notify a Thread of One or More Events

Usage

void Tnotify(UINT8 *thread_id*, UINT32 *ev_mask*);

Description

Tnotify() informs the thread(s) specified by "*thread_id*" of the occurrence of the set of events specified in "*ev_mask*." If the specified thread is waiting for one or more of these events, it becomes "satisfied" and is ready to run after all previously "satisfied" threads are run. If the thread is not yet waiting for these events, they become "pending" events for the thread, and hang around until the thread issues a Twait(), or Tclear_events() call.

"*thread_id*" must be a value from 0 through 7, inclusive, or set to THREAD_ID_ALL_THREADS. If "*thread_id*" is set to THREAD_ID_ALL_THREADS, then all threads in the system are notified of the events. If "*thread_id*" is greater than 7, and not equal to THREAD_ID_ALL_THREADS, this function will call UTILpanic() with a MSG_TNOTIFY_BAD_ID message code, and with "*thread_id*" as the message argument.

This function CAN, and probably will, be called from within an interrupt.

Return Values

NONE - FUNCTIONS IS VOID.

Caveats

1) Tinit() must be called before calling this function.

2) If "*ev_mask*" is zero, this function just returns.

See Also

UTILpanic().

6 Detailed Design Considerations

6.1 Data Flow

The following illustrates the overall data flow within the Thread Manager Module:

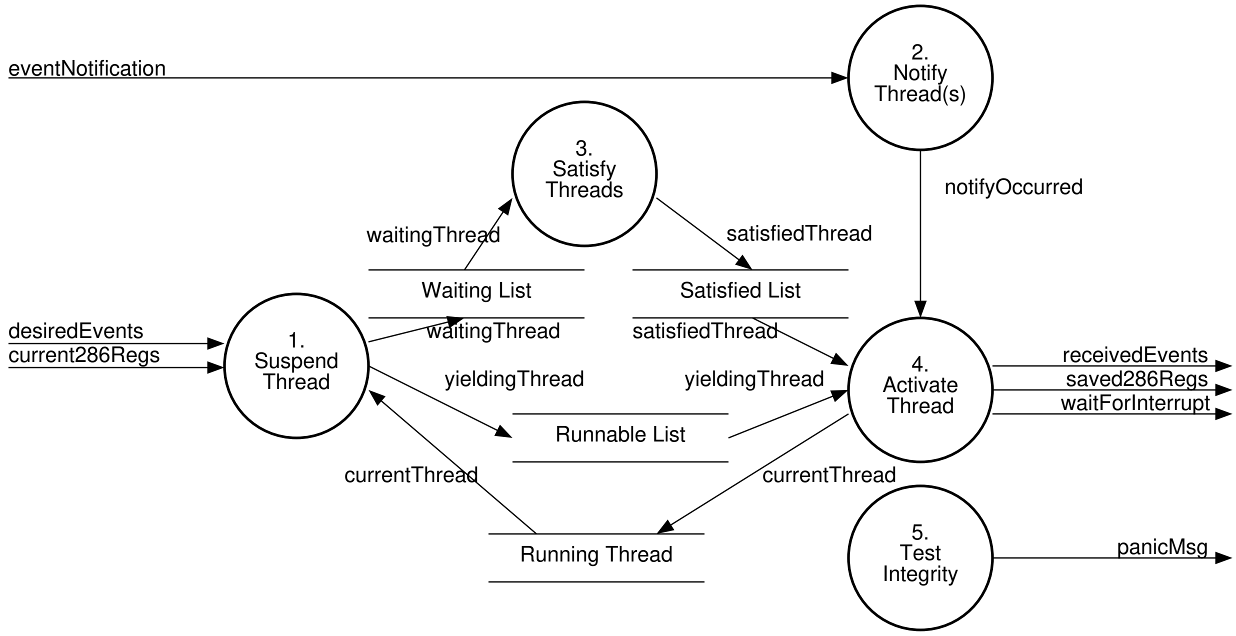

Figure 4 Thread Manager Module Data Flow

The Thread Manager maintains the state of the system's threads using thread descriptors. Each descriptor corresponds to a particular thread, and contains all state information for that thread. A thread exists in one of 4 states: waiting, runnable, satisfied, and running. Any number of threads may be waiting, runnable, or satisfied. Only one thread is running at a time (there's only one 80286 per EA).

When a thread relinquishes control (bubble 1), the thread manager records the state of the thread (current286Regs), and the events which will cause the thread to be activated (desiredEvents), and adds the thread's descriptor to either the Waiting List, or if no events are specified (desiredEvents == NULL), to the

Runnable List. It then determines the next thread to run (bubble 4). If there is a thread in the Satifsied List, it is run next. If not, but there is a thread in the Runnable List, it is run. If no threads are ready to run, and a notification has not occurred (notifyOccurred) while lists were being checked, the Thread Manager calls the **Interrupt Manager Module** to wait for an interrupt to occur. If a notification occurs while the Thread Manager is checking the lists, it repeats the tests, since a thread may have been satisfied during the time the tests were being performed. The transfer of control from one thread to another is known as a "context switch."

Whenever an FS module notifies a thread (or all threads) of an event (bubble 2), the notified events are recorded in the thread descriptor(s), and a flag is asserted, indicating that one or more notifications have occurred (notifyOccurred). Every time a context switch occurs, the threads in the Waiting List are checked to see if the desired event(s) have occurred. If so, the threads are removed from the Waiting List and are appended to the Satisfied List (bubble 3).

Whenever the Thread Manager gets control, it performs argument and integrity checks. If the Thread Manager detects an error, it resets the EA by issuing a panic message (bubble 5).

6.2 Control Considerations

6.2.1 Context Switching

This section describes the issues involved when performing thread context switching. The following lists some of the issues involved:

1) Only modify the stack pointer with interrupts disabled. Interrupt handlers use the current stack. If the stack segment (ss) and stack pointer (sp) aren't matched, random memory corruptions can occur if an interrupt happens at the wrong time.

2) When reading or modifying data which can be modified by an interrupt handler (in particular, event notification masks, and the "notify occurred" flag), guard the operations against unexpected modification by atomically disabling interrupts using INTRbegin_atomic and INTRend_atomic (see **Interrupt Manager Module Specification**).

6.2.2 Thread Latencies

The Thread Manager passes control from one thread to another only when the currently running thread relinquishes control. The time it takes for a thread to regain control, once it has relinquished control, is dependent on the time taken by all other threads in the system, and on time spent processing interrupts.

In the current architecture, one thread (Supervisor) is responsible for resetting the Watchdog Timer. It must gain control at least once every 8 seconds. Therefore, the maximum time taken by all other threads must be less than 8 seconds. This implies two things. First, each thread is responsible for relinquishing control often enough, such that the Watchdog Timer is reset before it expires. Second, each thread can assume that it will get control within 8 seconds after relinquishing control. In practice, there are tighter constraints on the time between context switches, such as; the command packet rate, the MajorTick processing latency requirements of a given thread, the desired telemetry rate, etc. Each thread's design must take these factors into account, and specify its latency requirements. Once all of the thread's designs are complete, the system utilization needs to be evaluated to ensure that each thread gets control often enough to meet its needs.

6.3 Module Structure

6.3.1 Public Module Functions

The Thread Manager provides a set of functions for use by the rest of the Flight Software. This section describes the structure of all functions which comprise the Thread Manager Module.

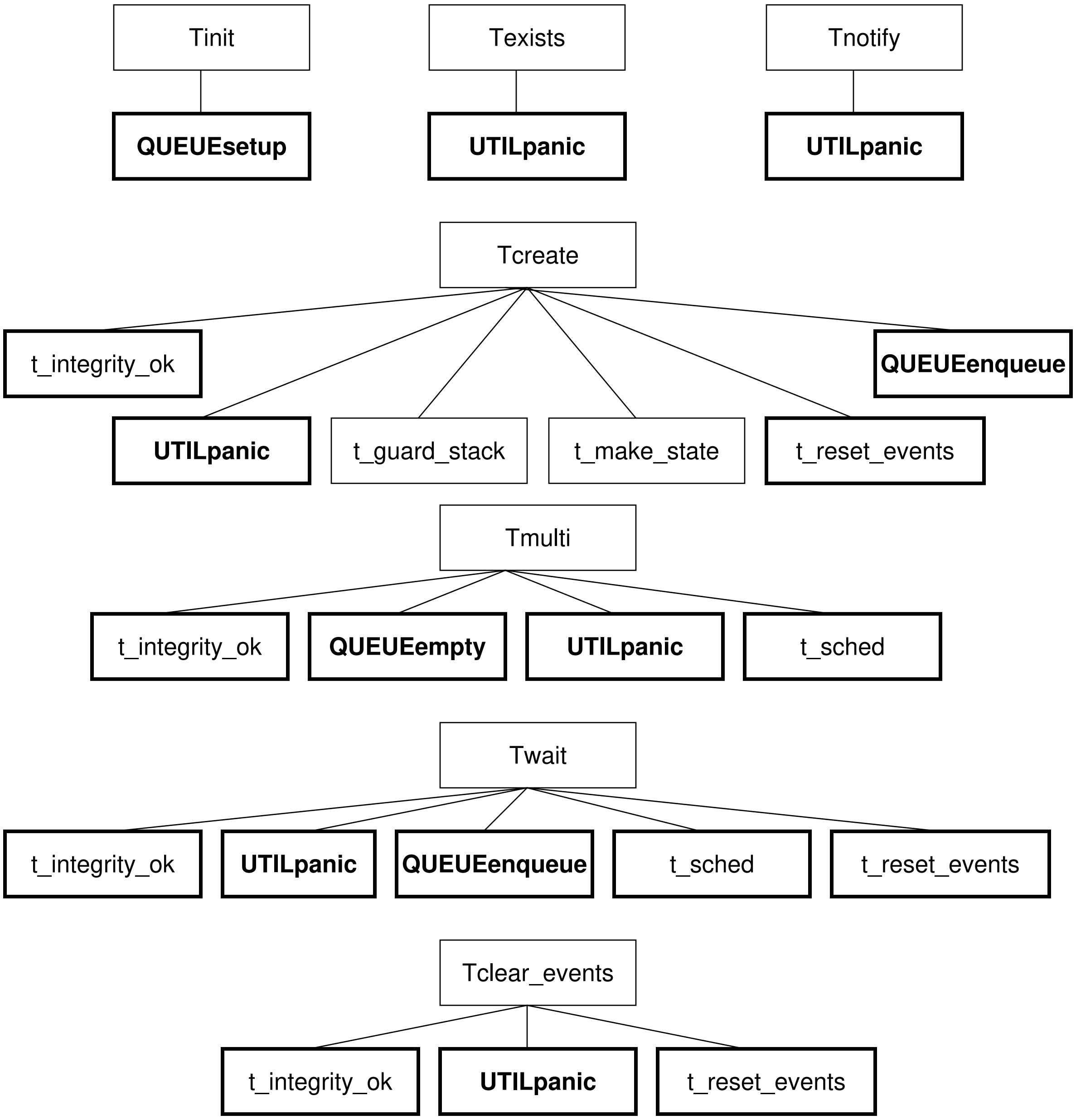

Figure 5 Thread Manager Structure

Figure 5 does not show calls to the **Interrupt Manager** functions INTRbegin_atomic() and INTRend_atomic(). The following describes each of the functions shown in Figure 5. The details of each function's operation and arguments are described in the Thread Manager Module's detailed design document.

Tinit .... This function is invoked by the **Startup Module**, before any other functions in this module are called. This function is responsible for initializing all thread descriptors and module specific state variables (see Section 5.1). Tinit uses the **Queue Utility Module** to setup the runnable, satisfied and waiting lists (QUEUEsetup).

Texists.. This function is invoked by other Flight Software modules to determine if a particular thread has been created via Tcreate(). It performs tests on passed arguments, and will issue a panic using facilities provided by the **Utility Module** (UTILpanic), if it suspects that the Flight Software has been corrupted (see Section 5.3).

Tnotify .... This function is called by other Flight Software modules to notify a particular thread, or all threads, when an event occurs. This function performs tests on passed arguments, and will issue a panic using facilities provided by the **Utility Module** (UTILpanic), if it suspects that the Flight Software has been corrupted (see Section 5.7). If the tests succeed, it "or"'s in the passed event mask with the specified thread's notification mask. If all threads are specified, Tnotify() instead "or"'s in the event mask with a special "broadcast" mask, which is distributed to all of the threads later by t_broadcast(). It then sets a "notify occurred" flag, which is used by t_get_thread() during scheduling.

Tcreate .... This function is called by the individual thread initialization functions, which, in turn, are called by the **Startup Module**. This function initializes the thread's descriptor and stack, and then adds the descriptor to the end of the Runnable List. This function verifies the state of the Thread Manager using t_integrity, and if any inconsistencies are detected, this function will issue a panic using facilities provided by the **Utility Module** (UTILpanic). Tcreate then sets up the thread's stack, inserting guard values using t_guard_stack, and fills in the descriptor's initial values, using t_make_state, and clears any pending events using t_reset_events. Finally, Tcreate appends the thread's descriptor to the end of the Runnable List, using the **Queue Utility Module** (QUEUEenqueue) (see Section 5.2).

Tmulti .... This function is called by the **Startup Module** to start running the threads created by Tcreate. Tmulti first tests the integrity of the Thread Manager, using t_integrity. It then verifies that there are threads in the Runnable List, using the **Queue Utility Module** (QUEUEempty). If a problem is found, it issues a panic using facilities provided by the **Utility Module** (UTILpanic). Tmulti starts the threads, using t_sched. At this point, the first thread in the Runnable List becomes the running thread (see Section 5.4).

Twait .... This function is called by the thread code to wait for one or more event notifications. Twait first calls t_integrity to validate the state of the Thread Manager. If a problem is found, Twait issues a panic message using the **Utility Module** (UTILpanic). Twait then places the running thread's descriptor into the Waiting List, using the **Queue Utility Module** (QUEUEenqueue). Twait then calls t_sched to save the context of the thread, and to restore the context of the next thread to run. When control is returned to the thread, Twait clears any satisfied events from the thread's descriptor, using t_reset_events and returns (see Section 5.5).

Tclear_events .... This function is called by the thread code to clear any pending events from the thread's descriptor. Tclear_events tests the integrity of the module, using t_integrity, and if a problem is found, issues a panic message using the **Utility Module** (UTILpanic). Tclear_events then clears any pending events, using t_reset_events (see Section 5.6).

t_integrity_ok.................................... This function is called by the other Thread Manager functions to perform integrity checks on the state of the module. This function is described in more detail in Section 6.3.2.

t_guard_stack .... This function is called by Tcreate to initialize a given thread's stack. It inserts guard values at the top and bottom of the thread's stack, so that stack integrity checks can detect possible stack overruns and underruns.

t_make_state .... This function is called by Tcreate to initialize a given thread's descriptor and stack area, such that the thread is runnable once multitasking has started. t_make_state inserts the starting address of t_exit_hook into the thread's stack space, such that, if the thread ever returns, t_exit_hook will be invoked.

t_reset_events .... This function is called by Twait and Tclear_events to safely clear events notified to a particular thread. This function is described in more detail in Section 6.3.2.

t_sched .... This function is called by Tmulti and Twait to transfer control from one thread to another. This function is described in more detail in Section 6.3.2.

UTILpanic .... This function is provided by the **Utility Module**. It is called by the Thread Manager's functions to issue panic messages and to reset the EA.

QUEUEsetup .... This function is provided by the **Queue Utility Module**. It initializes the elements of a queue control structure. Tinit calls this function to setup the control structures for the Runnable List, the Satisfied List and the Waiting List.

QUEUEenqueue .... This function is provided by the **Queue Utility Module**. It appends an element to the end of a queue, specified by a passed queue control structure. This function is used by various Thread Manager functions to append thread descriptors to the end of the Runnable, Satisfied and Waiting Lists.

QUEUEdequeue .... This function is provided by the **Queue Utility Module**. It removes an element from the start of a queue, specified by a passed queue control structure. This function is used by various Thread Manager functions to retrieve thread descriptors to the beginning of the Runnable, Satisfied and Waiting Lists.

QUEUEempty .... This function is provided by the **Queue Utility Module**. Its return value indicates whether or not a queue, specified by a passed queue control structure, is empty or not. This function is used by Tmulti's sanity check to determine if there are any threads in the Runnable List.

6.3.2 Private Module Functions

This section describes some of the Thread Manager Module's private functions. The structure of these functions are illustrated below:

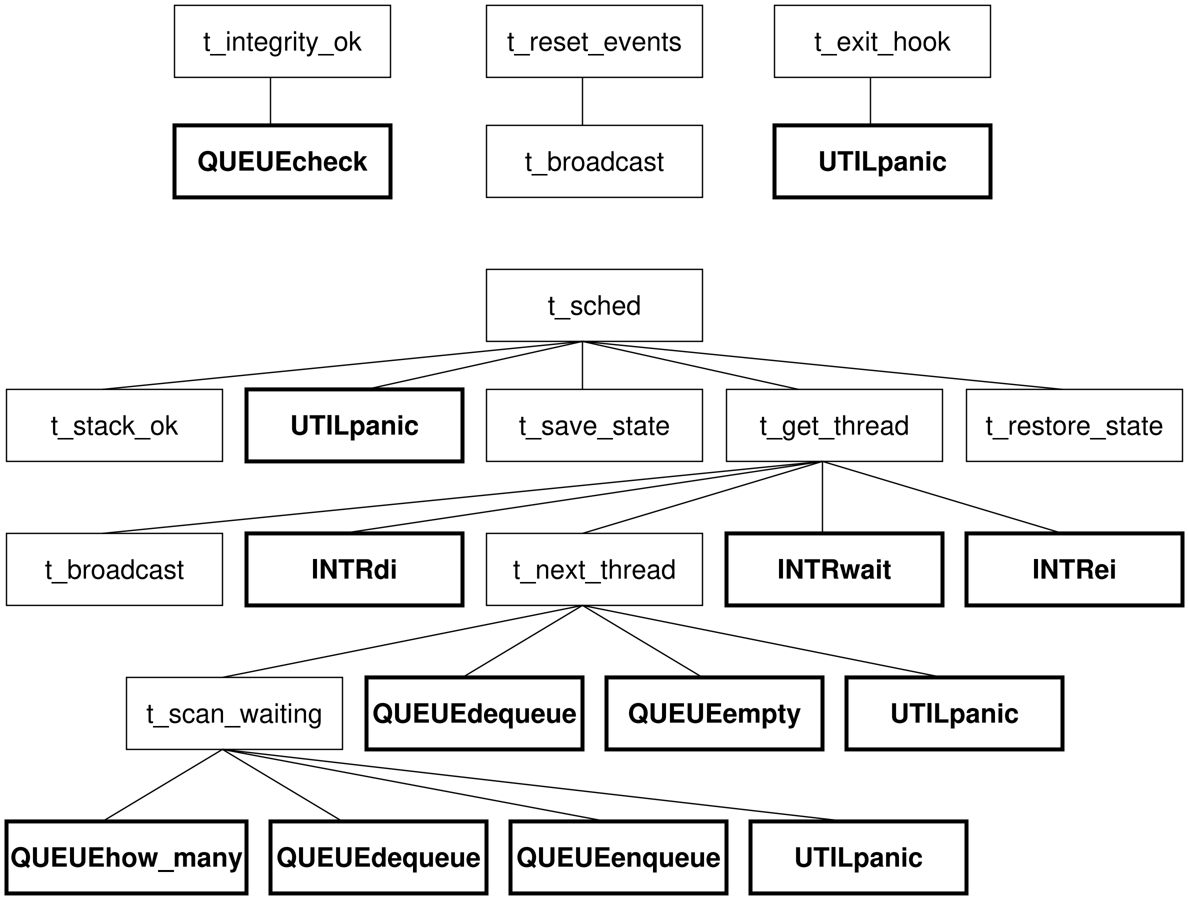

Figure 6 Thread Manager Private Functions Structure

Figure 6 does not show calls to the **Interrupt Manager** functions INTRbegin_atomic() and INTRend_atomic(). The following describes each of the functions shown in Figure 6. The details of each function's operation and arguments are described in the Thread Manager Module's detailed design document.

t_integrity_ok .... This function is called by the other Thread Manager functions to perform integrity checks on the state of the module. It performs sanity checks on the current thread id. It also tests the integrity of the Waiting List, Satisfied List, and Runnable List, using the **Queue Utility Module** (QUEUEcheck).

t_reset_events .... This function is called by Twait and Tclear_events to safely clear events notified to a particular thread. This function first updates all of the thread descriptor's received event masks, using t_broadcast. It then clears the specified events from the specified thread's event mask.

t_exit_hook .... The starting address of this function is inserted into a newly created thread's stack by t_make_state, such that, if the thread ever returns, t_exit_hook is invoked. t_exit_hook uses the **Utility Module** (UTILpanic) to issue a panic message and reset the EA.

t_broadcast .... This function is called by t_reset_events and t_get_thread to distribute events which were notified to all threads since the last call to t_broadcast. This function "or"'s in the "broadcast" notification mask, set by Tnotify(), into each of the thread's private notification masks, and then clears the "broadcast" mask.

t_sched .... This function is called by Tmulti and Twait to transfer control from one thread to another. t_sched first tests the integrity of the stack of the thread which is relinquishing control, using t_stack_ok. If a problem is discovered, t_sched issues a panic message, using the **Utility Module** (UTILpanic). If the relinquishing thread's stack is ok, t_sched saves the stack and register context of the thread in the thread's descriptor, using t_save_state. t_sched then gets the next thread to run, by calling t_get_thread. t_sched tests the integrity of the stack of the thread that is about to get control, using t_stack_ok. If a problem is discovered, t_sched issues a panic message, using the **Utility Module** (UTILpanic). If no problems are discovered, t_sched then restores the stack and register context of the obtained thread, using t_restore_state.

t_stack_ok .... This function is called by t_sched to test the integrity of a given thread's stack. This function tests for stack overruns and underruns.

t_save_state .... This function is called by t_sched to save the state of the currently running thread. This function appears to have two types of returns. The first is the return when the state is saved. The second is the "apparent" return due to the thread obtaining control (having its state restored) via t_restore_state (see Section 6.2.1).

t_get_thread .... This function is called by t_sched to obtain the next thread to run. t_get_thread atomically deasserts the "notify occurred" flag. It then enables interrupts and updates all of the thread descriptors received event masks, using t_broadcast. t_get_thread then attempts to get the next thread to run, using t_next_thread, and if a thread is ready to run, it returns with the new thread. If no threads are ready to run, t_get_thread disables interrupts on the CPU (i.e. interrupts remain enabled on the interrupt controller), and tests the "notify occurred" flag. If the flag remains deasserted, no calls to Tnotify were made during the search, hence no threads were satisfied. In this case, t_get_thread calls INTRwait, in the **Interrupt Manager Module**, to wait for an interrupt request from the controller to occur and to reenable interrupts. If the flag was asserted, then Tnotify was called during the search and a thread may have been satisfied. If the flag was asserted, or when INTRwait returns, t_get_thread enables interrupts and repeats the process, starting with the deassertion of the "notify occurred" flag.

t_restore_state .... This function is called by t_sched to restore the state of a suspended thread. This function never returns (see t_save_state and Section 6.2.1). This function will temporarily disable interrupts, to ensure that no interrupts occur while it is manipulating the stack pointer (see Section 6.2.1, Item 1).

t_next_thread .... This function is called by t_get_thread to obtain the next thread to run. This function starts by calling t_scan_waiting, which scans the Waiting List, moving any satisfied threads onto the Satisfied List. t_get_thread then checks the Satisfied List to see if there are any threads ready to run. If so, it removes the first thread from the Satisfied List, using the **Queue Utility Module** (QUEUEdequeue) and returns with the removed thread. If there are no threads on the Satisfied List, t_next_thread removes the first thread on the Runnable List, using the **Queue Utility Module** (QUEUEdequeue), and returns with the removed thread. If there are no threads on the Runnable List, t_next_thread tests to see if there are any threads on the Waiting List, using the **Queue Utility Module** (QUEUEempty). If there are no threads on the Waiting List, t_next_thread assumes that the Thread Manager has been corrupted, and issues a panic message (UTILpanic). If there are threads on the Waiting List, t_next_thread just returns with a code indicating that there are no threads ready to run.

t_scan_waiting .... This function is called by t_next_thread to move any satisfied threads from the Waiting List to the Satisfied List. This function uses the **Queue Utility Module** to determine the number of threads in the Waiting List (QUEUEhow_many), to remove threads from the Waiting List (QUEUEdequeue) and to add threads to the Satisfied List, or back on to the Waiting List (QUEUEenqueue). t_scan_waiting starts by determining the number of threads in the Waiting List. Using this count, it dequeues each thread and checks to see if the thread has been satisfied. If the thread has been satisfied, it is enqueued onto the Satisfied List. If the thread has not been satisfied, it is enqueued back onto the Waiting List. The initial list count is used to ensure that each thread is examined only once. During this scanning process, t_scan_waiting checks the thread descriptor's internal state variables, to ensure that the thread is supposed to be on the Waiting List. If the descriptor's state variables indicate a discrepency, t_scan_waiting assumes that the system has been corrupted, and calls the **Utility Module** (UTILpanic) to send a panic message and reset the Event Analyzer.

7 Requirements Traceability

7.1 Science Requirements

This module does not address any specific Science Requirements, beyond the ability to execute multiple threads, and coordinate the operation of these threads.

8 Testing Considerations

8.1 Unit Testing

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

a. An Event Analyzer equipped with the Intermetrics ROM Monitor

b. A workstation running the Intermetrics debugger

c. An RS-422 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 must ensure that interrupts are never disabled for more than 1ms at a time within any function in the Thread Manager, except around calls to INTRwait(). This can be accomplished using the logic analyzer. Since calls to INTRwait() indicate that the system is waiting for an interrupt, interrupts may be disabled for up to 32ms (the frequency of the Minortick interrupt).

8.2 Integration Testing

For integration tests, the following equipment is required:

a. An Event Analyzer

b. A populated ROM Repository board

c. A logic analyzer

d. Hardware which can:

   1. Drive the MinorTick and MajorTick clock signals

   2. Receive telemetry packets from the EA.

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

a. Verify that context switches occur properly during heavy interrupt loads.

b. Verify that event notifications from interrupt handlers are handled properly.

Before system integration test, the worst case latencies of the individual threads need to be tested to ensure that the system requirements specified in Section 6.2.2 are handled properly. These tests may be part of the tests of the specific thread modules.

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		