SuperTask!®
User's Guide

Revision 6.03
December 1999

U S SOFTWARE®
EMBEDDED EXCELLENCE

www.ussw.com

Copyright and Trademark Information

United States Software Corporation
7175 NW Evergreen Parkway, Suite 100
Hillsboro, OR 97124
(503) 844-6614
Fax (503) 844-6480
E-mail: support@ussw.com

Quick Contents

1 2 3 4 5 6 A B C

Documentation Conventions

Computer output and code examples: Courier, usually in a separate paragraph.

Function names and command names: ***Bold italic***, usually followed by parentheses, as in ***main()*** function.

Variables: Courier 11 italic (*mt_busy)*.

File names: Times bold (the file **usrclk.asm**), in lower case.

Key names: Initial capital, in angle brackets, as in press <Enter>.

Menu names and selections, dialog box names, screen titles, window titles: Times bold, as in **File** menu**.**

NOTE: Indicates important information.

CAUTION: Indicates potential damage to hardware or data.

WARNING: Indicates potential injury to users.

Revision History

Revision Number	Date
6.01	July 1997
6.02	October 1997
6.03	December 1999

Quick Contents (continued)

D

E

I

Notes

Contents

1. Read This First

1

Chapter Contents

How to Use this Document

MultiTask!™ is the RTOS kernel you will be running on the target platform with your application code. MultiTask! is delivered in source form with test programs that will run on your target processor, and appropriate makefiles for building these programs with one or more cross-compilers. Your application code will be compiled and linked with the MultiTask! RTOS code and the whole works placed on your target (commonly in ROM). In this manual we will refer to the RTOS as SuperTask!®, MultiTask!, or even MT!.

The *SuperTask! User's Guide* contains:

1. Read This First (installation instructions, customer support, and general information)
2. MultiTask!
3. MultiTask! Library Reference
4. MultiTask! Internals
5. Stream I/O
6. Stream I/O Library Reference

A. Platform-Specific Information
B. PC-Compatible Console/Keyboard
C. Global Variables
D. Error Codes
E. Glossary

Index

We know from experience that software documentation arrives at a rate faster than you can possibly read. We have therefore tried to write these new manuals in a fashion that will tell you clearly what you need to know as briefly as possible without a lot of needless verbiage.

All users should read through the *MultiTask!* chapter of this manual. This explains our terminology and presents a descriptive explanation of all features of the operating system. The *MultiTask! Library Reference* chapter gives detailed information for each function call in the MultiTask! library.

The *How to Design your Application* section of the *MultiTask!* chapter gives some examples of how to accomplish various things with MultiTask!, based on years of experience. This might be especially helpful if this is your first excursion into multitasking with our operating system.

The appendix on *Platform-Specific Information* contains information for each target processor. You should read all sections that apply to your environment. Any information that applies only to specific processors or configurations is located there.

Installing SuperTask!

Each of the SuperTask! diskettes contains an **install.bat** installation batch file that should be used for installing the product. The installation batch file will allow you to select the destination directory where the software will be installed and the tool chain (compiler, etc.) that you are using.

Put the diskette into the appropriate disk drive and make that the current directory. Type:

```
a:\> install
```

The batch file will display the appropriate syntax. Repeat the installation with the appropriate syntax.

The **install** syntax requests the destination directory (which will be created if it does not exist) and a code for the compiler you will be using. For the SuperTask! x86 real mode release, the compiler options are B (for Borland) and M (for Microsoft).

Example: Installing the x86 real mode SuperTask! release for the Borland compiler (diskette inserted in drive A), type:

```
c:\> cd a:
a:\> install c:\st B
```

The preceding will create the directory **C:\ST** and install all files appropriate for use with the Borland C compiler there.

Text Files on Delivery Diskettes

If any of your delivery diskettes contain text files (extension **.txt**), these may contain important information not in the printed documentation. We try to keep everything up to date, but it is just not possible to update all products simultaneously. Our highest priority is to make sure everything on the delivery diskettes is functional and complete. The diskettes may contain some important last-minute instructions not printed in the manual, which can be found in the following files: **relnotes.txt**, **cpunotes.txt**, and/or **compnote.txt**. Not all of these files exist for every distribution.

Makefiles

In your installation directory there will be a **makefile**. This is a text file named **makefile** that is used with a **make** program to build the libraries and test programs from the source supplied. All makefiles supplied with MultiTask! versions 6.20 or later are designed to use the **make** from "Opus Software". This is supplied as part of the product package. Use the supplied makefile with Opus **make** only, to build the product test programs and libraries. You can create your own makefile using whatever **make** (**nmake**, etc.) you want and link to the library created by the makefile we supply.

The **makefiles** contain important *comments* (lines beginning with #) about compiling and linking the libraries and test programs. If you have not been using a **make** program, we most strongly recommend you do so as it will automate the whole build process and make it much less tedious and error-prone.

Usually the only change necessary to the makefile to run on your system will be to edit the *PTH* variable in the file to be the pathname to where the compiler you are using resides. If you are running under the MS-DOS prompt from Windows and experience problems, try running directly from DOS.

To support a particular development platform, you might need to adjust the symbols *TRG_ID* and *DBG_ID*, which specify the evaluation board and debugger, respectively. There will be notes in the **makefile** that explain the use of these symbols. Not all makefiles use the *DBG_ID* symbol.

The **makefile** supplied will build the library for your configuration and the test programs supplied. Once you have a library, you might want to do your actual development in another directory with your own **makefile**, and reserve the original installation directory for rebuilding the libraries and test programs when you change any configuration parameters.

Source Files

Even though you should not need to change most of the source files in the delivery, you will probably want to look at them at some time to get a better understanding of the code. We have made an attempt to replace all tabs with four spaces in our source code, but it is possible that we have missed something. Editors do not all display tabs in the same way, so you might encounter strangely formatted text if we missed some tabs. To fix this formatting, we have included a utility program called **detab.exe** that can be used to make a copy of the files, replacing tab characters with the appropriate number of spaces.

The syntax is:

```
detab file [files] dir
```

The *file* specification can contain standard DOS wild card characters “*” and “?”. The *dir* is the pathname to a directory to copy the *file* to.

Example:

```
mkdir src2
detab *.c *.h src2
```

This will copy all "**.c**" and "**.h**" files in the current directory to files of the same name in the directory **src2,** with tabs translated.

A UNIX-style **grep** utility can also be of much use in finding where things are in the source. The Borland C compiler includes a **grep** utility. It is also available in a number of commercially available toolkits, such as the Thompson Automation Software toolkit.

Grep searches text files and prints lines containing a given text string. For example, **`grep setevt *.c`** would print the filename and line in every *.**c** file in the current directory that contained the text string `setevt`.

What Is Required of You

SuperTask! is not an application program. It is code that will be combined with code you write, and become part of your final application. SuperTask! is designed primarily to be a part of an embedded application, meaning one that will reside on a piece of hardware (usually in ROM) that is dedicated to some function.

You will need to know or learn how to perform the device-level programming for your target hardware, i.e., serial ports, timers, interrupts, and any other applicable devices. You will also need to know general programming practices in C and assembly for your selected target processor, and the use of the **make** utility program (Borland or UNIX **make**, Microsoft **nmake**, Opus **make**, etc.). In addition, you will need to provide the necessary startup code for your application. SuperTask! is simply a library of functions. It is not an environment.

We thoroughly test each release in-house on as many test platforms as we have available. Unless your target board has exactly the same configuration as our test platform, you will have to make the necessary changes to the target-specific code we supply (usually only timer and I/O routines) before you can run on your target. The majority of the code (more than 95%) is not specific to any platform and will not require change. The more your target differs from ours, the more you will have to do. In most cases this will still be a small task.

If you think massive changes will be necessary to support your environment, we suggest you call us first to confirm what you are thinking before you start. We might be able to save you from starting down the wrong road and doing a lot of needless work.

Lastly, if our programming style differs from what you are used to, we hope you will forgive us our differences. We are constrained somewhat by the fact that all the C code must and does run on at least a dozen different processor families, and more than two dozen compilers. Sometimes this requires things to be not quite as pretty as

we would like, or as we could do if we were supporting only one environment.

If you have constructive suggestions for improvement, let us know. We usually incorporate the best ideas we can find, although this does take a bit of time. Since it seems to be rare that any two of our customers want exactly the same things, it will also probably be rare that we do everything you want. Our code will, however, probably satisfy most of your needs as long as you are willing to fill in the last little bit yourself, or live without it.

Calling for Support

When to Call

The support line does not function as a substitute for reading the manual. Please do not call for simple questions that are answered in the manual. If the manual is not clear to you, or if you have some other question that is not answered, then by all means call. We will not design your application for you, but we can offer suggestions on how you might best use the operating system features, given a description of what you are trying to accomplish.

If you are lost and just want to know where to start, or you want to check to make sure you are not lost, go ahead and give us a quick call. To save time for both of us, please have the information listed under *How To Call* ready before you place the call.

We may be able to refer you to a consultant if you need additional help developing your application.

How to Call

Locate the product name and version number you want support for. This is obtained from the **vsnlog.txt** file that is placed in the directory where you installed the product through the **install.bat** procedure. If you are running on an MS-DOS system, there is also an executable program called **vsn.exe** in that directory that will display the pertinent information from the **vsnlog.txt** file. In this case, typing the following will display what you need:

```
CD \product_directory
VSN
```

If you cannot run **vsn.exe**, then you can read the top few lines of **vsnlog.txt** to get this information.

Call our office at (503) 844-6614 during normal business hours, which are 8 a.m.–5 p.m. (Pacific Time) Monday through Friday.

When the phone is answered, request "technical support for *name_of_product.*"

When you are transferred to an engineer, state the "*product-code*, *version-number*" you are calling about, and the "*processor* and *compiler*" you are using.

This will allow us to support you better. You might have only one version of our product, but we might be dealing with perhaps 100 variants. The aforementioned information will tell us exactly what you have and give us the proper frame of reference to answer your questions.

sample *name_of_product* = {SuperTask!}

sample *product-code* (from **vsnlog.txt**) = {MT80x86, MT960, etc.}

sample *version-number* (from **vsnlog.txt**) = {6.29}

sample *processor* = {68332, i960CA, etc.}

sample *compiler* = {Microsoft, Borland, Watcom, etc.}

You may also fax your questions to us at any time or contact us by e-mail.

Voice phone: **(503) 844-6614**

Fax: **(503) 844-6480**

E-Mail: **support@ussw.com**

Reporting Bugs

If you believe you have found a bug in any of our products, please do not hesitate to notify us. We normally do not ship products with any known bugs, and if a new one is found we try to fix the problem as quickly as possible. From a practical standpoint, it will be much more helpful if you can send a short example that will reproduce the problem, along with a description of what happens. If you have an example, it is best to fax it to us at (503) 844-6480 so we have something printed to look at, rather than explaining it verbally. Be sure to include the information requested under *How To Call*, as well as your return fax number and voice phone number.

2. MultiTask!

2

Chapter Contents

Overview

Introduction

Microprocessors may be fast, but they can perform only one task at a time. To make multiple activities appear to be occurring simultaneously, the microprocessor must use a process called *multitasking* by switching rapidly between them.

Writing a multitasking program to be run on a specific microprocessor can be a highly complicated process requiring a considerable investment in design and development. However, a multitasking executive, such as MultiTask! (abbreviated MT!), simplifies design and development by allowing the application to be logically separated into simple tasks that are then integrated into the application as a whole.

Multitasking applications typically operate in a real-time environment, meaning they must perform specific tasks within a set period of time in response to ongoing events. A real-time system must respond quickly to interrupts and input from independent data or processes that occur asynchronously, and it must produce output based on the input data. The real-time system must also manage these input and output processes as nearly simultaneously as possible.

Real-time systems are developed to control or monitor industrial processes that require immediate response to input. For example, process measurement input from an instrument might require an immediate response for proper operation. In such a case, it is extremely important that the application be available when needed to respond quickly and accurately to input.

Multitasking executives such as MT! are used to create applications on target processors. They can simplify the writing of an embedded application, especially when the application requires managing a number of different activities with events that occur asynchronously.

MT! performs this function by allocating microprocessor resources for the tasks. These resources are limited — a fixed amount of memory space and computer time exists to perform any given task. MT! allocates these resources on a user-specified basis. In other words, MT! enables a user (programmer) to apply rules to computer time and memory distribution inside a program. Essentially, MT! supervises the orderly execution of separate application tasks according to the rules the programmer specifies.

MT! provides the operating system services needed to write real-time multitasking programs. It provides specialized software containing functions that handle processor resource allocation, software events, timing functions, queue management, inter-program communications, and reentrant memory management.

MultiTask! Features

MT! is a user-configurable family of multitasking executives designed specifically for embedded applications. The MT! code is supplied in source form with the necessary makefile to generate an object library of functions configured for your environment. The MT! code is both ROMable and reentrant. MT! provides fully preemptive priority-based scheduling of tasks, as well as round-robin time-slicing of tasks if desired. The MT! system uses a multithreaded, as opposed to multiprocessing, approach, and each individual task is a single thread.

To enhance the implementation of control applications, MT! provides a comprehensive set of ANSI C system functions. These functions allow the user to customize MT! to the application by selecting parameters that control the amount of time, available memory, the number of tasks, and other parameters.

Optimized for the microprocessor selected by the user, MT! is designed to be easily configurable for specific applications. You configure the MT! system by setting system parameters in a configuration file and choosing only the MT! functions needed. You can minimize RAM usage by the operating system by configuring limits before you compile MT!. Only the code for functions you

actually use will be linked to your application from the MT! library, thus minimizing code space usage.

In the application program ***main()*** function, you initialize MT! and transfer control to MT! to initiate multitasking. The C functions you designate in the application code will be made to run as independent tasks.

Multitasking

Multitasking is used to solve problems where multiple tasks must run concurrently (and possibly at different priority levels). With MT!, you may write these tasks (or processes) independently of one another to reduce the time and complexity of design, implementation, and test. Without MT!, integrated solutions can become unwieldy and difficult to debug, thus extending the design and implementation cycle.

For example, multitasking might be used in the firmware for an instrumentation application. A typical application could require software support for measurement, user interface, printer control, and various I/O and internal functions, as shown in Figure 2-1.

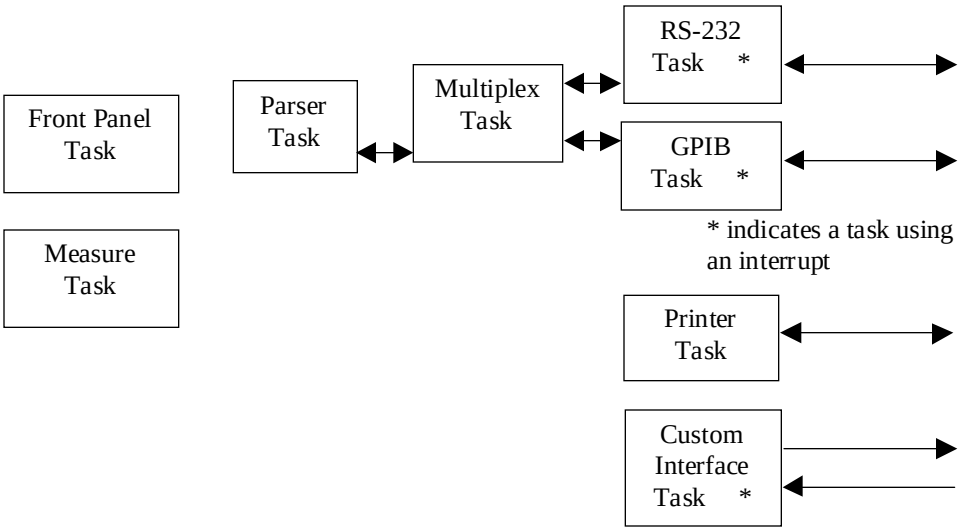

Figure 2-1: Multitasking system

These activities could be programmed as a set of tasks such as T1, T2, T3, T4, etc. Task T1 could handle measurement. Task T2 could provide the user interface. Task T3 might control the printer, and task T4 could direct the RS-232 activity, and so on.

Figure 2-2 compares an integrated software design approach, an in-line design approach, and a multitasking approach to the problem.

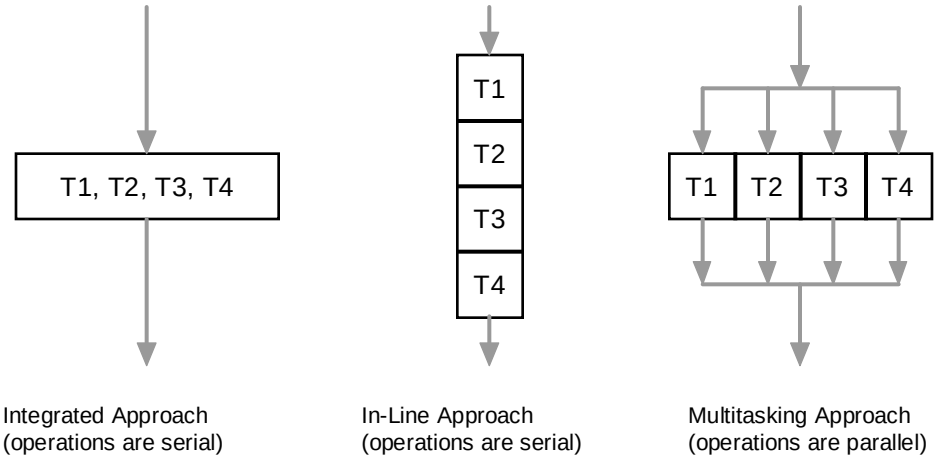

Figure 2-2: Comparison of integrated and multitasking approaches

MultiTask! Concepts

The information in the following section reviews essential concepts and terminology used in working with SuperTask!.

Tasks

A *task* is any C function in your application program that you designate as a task by using the ***runtsk()*** MT! function.

Tasks have these properties:

- Each task has its own stack space and CPU context (register set) and runs independently of other tasks.
- A waiting task, like a non-active Interrupt Service Routine, is not using any CPU resources (time, registers, etc.).
- All tasks have access to global-scope variables in the application, and must not use these in a way that would conflict with other tasks.
- Since each task has its own stack, local variables in the task and any function called by that task reside on its private stack and are, therefore, private to the task.
- The same code may be started as a task more than once, in which case multiple separate tasks will exist, each using the same copy of code, but with their own private stacks and CPU context (register set).

Associated with each task is an internal control structure that we call the *Task Control Block* or TCB. The TCB contains important information about the context of the task, its slot (ID), and the queue in which it is currently linked, which will define its state.

Task Control Block
Next Task Link
Starting Address
Stack Base Address
Stack Pointer
Allocated Memory Pointer
Current Queue Index
Group Event Set Mask
Group Event Clear Mask
Group Event Or Mask
Task Priority
Status Flags
TASK_ID
Task *errno*

Figure 2-3: The general form of a task control block (TCB)

Task States

Non-existent

When SuperTask! is initialized, an array of TCBs is created. Any TCB in this array that does not have a task associated with it is marked as non-existent. A task is made known to the system with the ***runtsk()*** function, which associates the task with an entry in the TCB array. After a task is terminated by the ***klltsk()*** function, it returns to the non-existent state, i.e., it is no longer a task.

Running/Ready

When each task is made known to the system by the ***runtsk()*** function, it is initially in the *ready* state (meaning it is linked into the *run queue*). The highest priority task in the ready state will be the *running* task (only one task can actually be running at any time). If no tasks are ready, the system enters an idling or low-power state.

Tasks that exist but are not ready must meet at least one of the following criteria: 1) the task is suspended, 2) the task is waiting for something, or 3) the task has a timeout specified (i.e. is waiting for a particular time to elapse).

Suspended

The ***suspend()*** function allows specific tasks to be temporarily blocked from running without affecting any other tasks. The *suspended* state is a non-running state that consumes no CPU time. A waiting task may be flagged to be suspended, so that when the normal action is taken that would cause it to return to the ready state, it will instead enter the suspended state and an additional action will be required to return it to the ready state.

Waiting

The *waiting* state is a non-running state.

Tasks in any queue other than the run queue are in the waiting state. Tasks usually enter the waiting state by performing some MT! function call to wait for some condition to occur, like receiving a message from a mailbox. Tasks in the waiting state will consume no CPU time. Tasks are moved from the waiting state back to the ready/ running state either by the action of some interrupt routine or by the running task's performing some MT! function call.

There are several queues in which a task may reside while waiting. Tasks in the Time Delay Queue are only waiting for a certain time to elapse. Tasks in the Limbo Queue are in a special wait state. Each item that a task can wait on (e.g. a mailbox or a resource) also has a wait queue. Tasks in these queues have a 'suspend' bit that allows a task to be both waiting and suspended. They may also have a timeout specified so that they do not wait indefinitely.

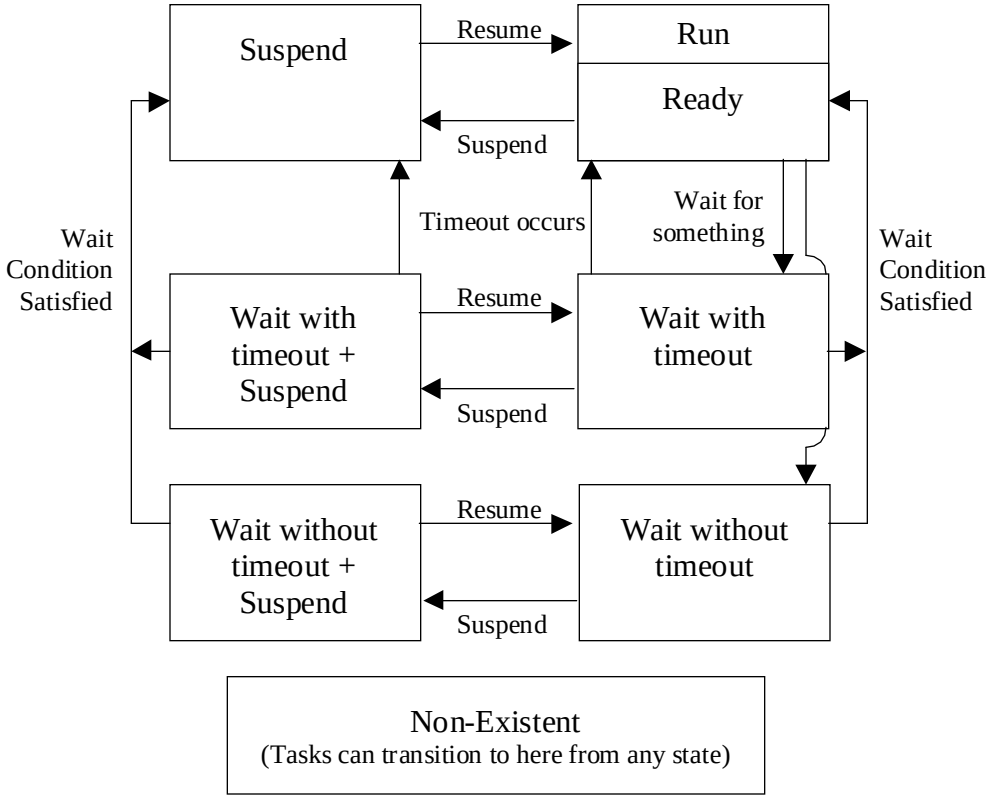

Figure 2-4: Possible task state changes

Arrowheads indicate direction of flow. A line touching a box with no arrowhead indicates a task coming out of that state (box).

Task Queues

A number of queues exist in MT!. These queues are:

Run queue	Suspend queue
Delay queue	Group event queues
Event set queues	Event clear queues
Resource queues	Mailbox queues
Memory queues	I/O queues
(Limbo queue)	(Non-Existent queue)

NOTE: The I/O queues are implemented for stream I/O functions, and their use is being phased out.

MT! maintains the queues in priority order with the highest priority task at the head of the queue. The TCB contains the link to the TCB of the next task in the queue, and identifies the queue in which the task is linked. The queue that a task is linked into identifies the state of the task; e.g., all tasks in the *run queue* are in the ready or running state, and tasks in a *mailbox queue* are waiting to receive a message from a particular mailbox.

You will notice that many of the queues listed above are referred to in the plural. There are actually multiple queues. Each condition it is possible to wait for has a separate queue. There is a queue for each mailbox, group event, resource, and stream path. Each event actually has two queues, one in which a task waits for the event to be set, and another in which tasks will wait for the event to be clear. This abundance of queues enables the fastest possible processing of tasks waiting in a queue. The *next task link* field in the TCB is used to link tasks into a *task queue* with other tasks.

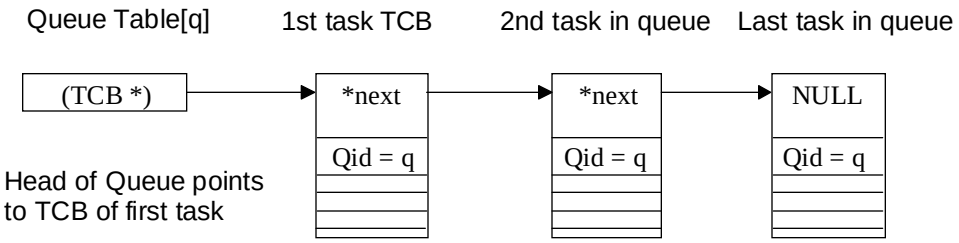

Figure 2-5: Task queues

Task Priority

Tasks have a priority assigned when they are made known to the system with the ***runtsk()*** function call. The priority is in the range 0..255. Zero is the lowest priority and 255 is the highest. The actual magnitude of the priority is of no significance; only its magnitude relative to other tasks in the system is significant. All that matters when considering two tasks is that one has a higher priority than the other; the amount of the difference does not matter.

Tasks are ordered by priority within task queues. The highest priority task is placed at the head of the queue. Tasks of equal priority are placed in the queue in FIFO order, i.e., the first task placed in the queue of a given priority will remain ahead of other tasks with the same priority that are added to the queue later.

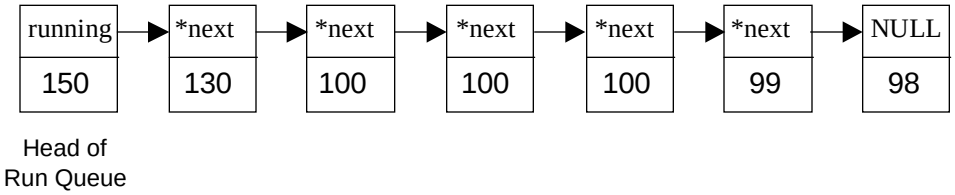

Figure 2-6: Run queue

All tasks in the run queue are in the ready state, and the highest priority task in the run queue at any time will be the running task.

If there are two or more tasks at the highest priority in the run queue, then the default action is for MT! to time-slice between them. At each clock scheduling tick, a task switch is made to the next task of that priority with the task just preempted rotated to the end of that priority level in the run queue.

Figure 2-7 shows this rotation of tasks at the scheduling tick.

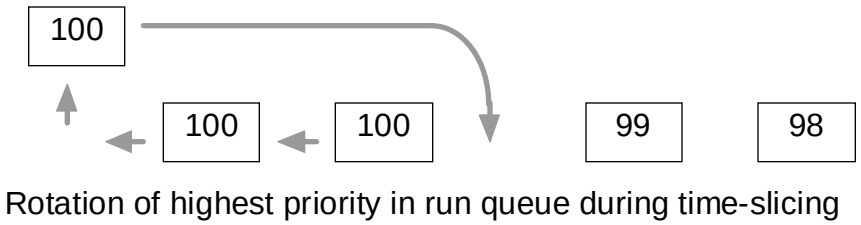

Figure 2-7: Rotation of tasks

This rotation of task order with the highest priority in a queue takes place only in the run queue, not within any other queue the task might be in. Also, this time-slicing behavior and queue rotation can be totally inhibited by a compile-time option when you compile any of the SuperTask! libraries.

Changing the priority of a task changes the MT! view of its readiness for execution. You can maintain control of task execution by assigning appropriate priorities to the tasks.

For example, MT! round-robin time-slices the tasks T1 and T2 (shown in Figure 2-8 below) until both are blocked. MT! then round-robins tasks T3, T4, and T5 until they are blocked as well. Finally, MT! executes task T6.

It is often necessary to delay task execution or periodically run a task. To do this, MT! provides time-oriented suspend and resume services. Using these services, task execution may be suspended for a specified time interval, or if suspended, task execution may be resumed.

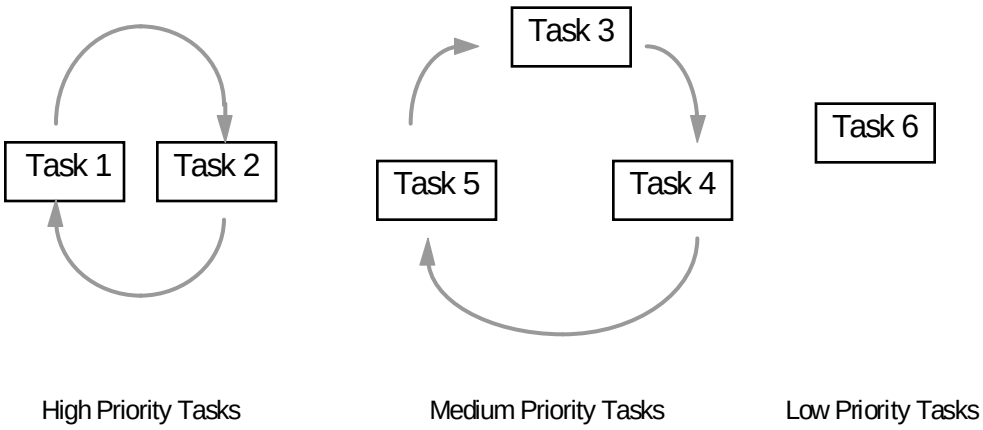

Figure 2-8: How task priority affects task execution

Preemption and Task Switching

In MT!, a *task switch* occurs when either the running task's time-slice expires if it is time-slicing, or when the running task voluntarily relinquishes the CPU during its time-slice, or when it is preempted. *Preemption* is when a higher-priority task interrupts the execution of a lower-priority task (much like an interrupt routine). When the running task voluntarily gives up the CPU to another task, we refer to this as a *cooperative task switch* because the running task is cooperating with other tasks by sharing the CPU. In MT!, time-slicing occurs only between tasks of equal priority. Under these

conditions, when the clock interrupt occurs and the next ready task has the same priority as the running task, a task switch will occur. This is referred to as round-robin scheduling. Time-slicing is a compile-time option that can be disabled in MT! if the user desires.

When preemption occurs, the context of the running task is saved and the context of the preempting task (the task to be run) is restored. The saving of one task's context (its running state) and the restoring of another task's context is known as a *context switch*.

The Time Queue

MultiTask! uses a time queue to implement all time-related features including task timeouts, delays, and periodic events. This time queue is maintained as a linked list of time-control structures in chronological order. If a task is waiting in a task queue (for an event, resource, etc.), and a timeout was specified, it will also have an entry in the time queue. The entry in the time queue is linked by a time control structure, not the task TCB. When each clock tick is processed, the system tick count is incremented and a check is made to see if the entry at the head of the time queue is waiting for the updated system tick count to equal the new setting. If it is, appropriate processing is performed; if not, no other check need be done regardless of how many time-related occurrences are scheduled.

The Command Queue

MultiTask! uses a *command queue* for processing system services from an interrupt service routine (ISR). This is part of a strategy for minimizing the amount of time that interrupts are masked by the operating system and thus minimizing interrupt latency. Normal system calls never mask interrupts. System call reentrancy is arbitrated by a flag (`mt_busy`) instead. This mandates that ISRs never call the system service functions directly. To allow system services to be instigated by an ISR, there is a special function call, ***MTqcmd_c()***, that will place the desired function call in the command queue, where it will be processed after the ISR completes.

The command queue is implemented as a circular buffer. The pointer `cmdadd` points to the location in the buffer where the next command will be placed by ***MTqcmd_c()***. The pointer `cmdprc` points to the next command in the buffer to be processed by the operating system. When these two pointers are equal, the command queue is considered empty. The command queue only supports commands that will not wait and will not return data to the calling routine. Figure 2-9 illustrates the command queue with two as-yet unprocessed commands in it. The first command is ***tiktok(),*** which is the system clock interrupt service process, and the second is ***setevt()*** to set event number 2.

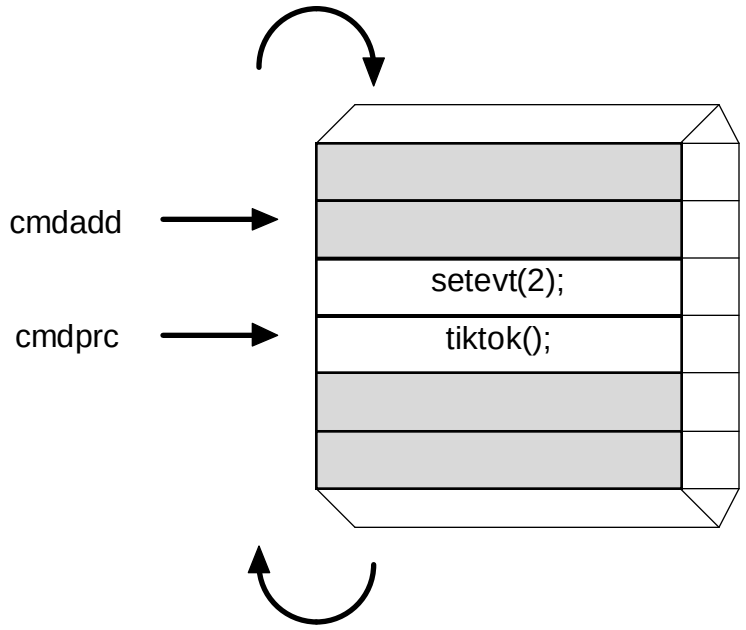

Figure 2-9: Command queue with two unprocessed commands

The ***tiktok()*** command is placed in the queue by the clock ISR function ***usrclk()***. The ***setevt()*** function in this example would have been placed in the command queue by some other ISR in the application. With the exception of the ***ireqbuf_c()*** function, no other system call can be made directly from an ISR. However, all useful services (excluding those that will wait or return information) can be initiated from an ISR indirectly, by using the ***MTqcmd_c()*** function call. All commands in the command queue are processed at the end of every system call, or immediately after any ISR that exits via the ***MTsched*** or ***MTsched_c()*** functions rather than the normal return from interrupt. See the later section on *Interrupt Service Routines* for more detail before attempting to use service from an ISR.

The number of items that can reside in the command queue at any one time is specified by the parameter `MAX_CMD_CNT` in the **depends.h** file. The default value is 32 and it uses about 4 words per entry.

Systems with long interrupts and frequent interrupts, especially those with both, should increase this value. Entries are filled (meaning that commands are queued) from the start of a long interrupt until ***MTqproc()*** can clean them out. Nesting interrupts and those that use several commands also use up entries.

NOTE: Long system calls also require more entries, since the queue is not flushed until the end of the system call. A call to ***reqmem()*** when lots of blocks are being used can take a while. Systems with lots of tasks will see a similar problem. In both cases, linked lists have to be searched, and the time to do so goes up linearly with the length.

MultiTask! Services

Starting and Controlling Tasks

When a task first comes to the attention of MT! (via the ***runtsk()*** function) it is placed in the run queue (a queue of tasks waiting for the CPU). The ***runtsk()*** function allocates stack space for the task, creates an initial context for the task on the task stack, and places the task in the run queue.

To perform these actions, ***runtsk()*** assigns a TCB structure for the task and fills in pertinent information in the TCB.

The task's slot number (which is the TCB index) is assigned by ***runtsk()***. The slot number is a unique task identifier used by many MT! functions to identify the task to be operated on. The slot number returned by ***runtsk()*** should be saved in some variable for future reference if needed.

In the following examples, tasks are shown in various queues (the priority of each task is in parentheses). Note that the tasks are in priority order in the queues.

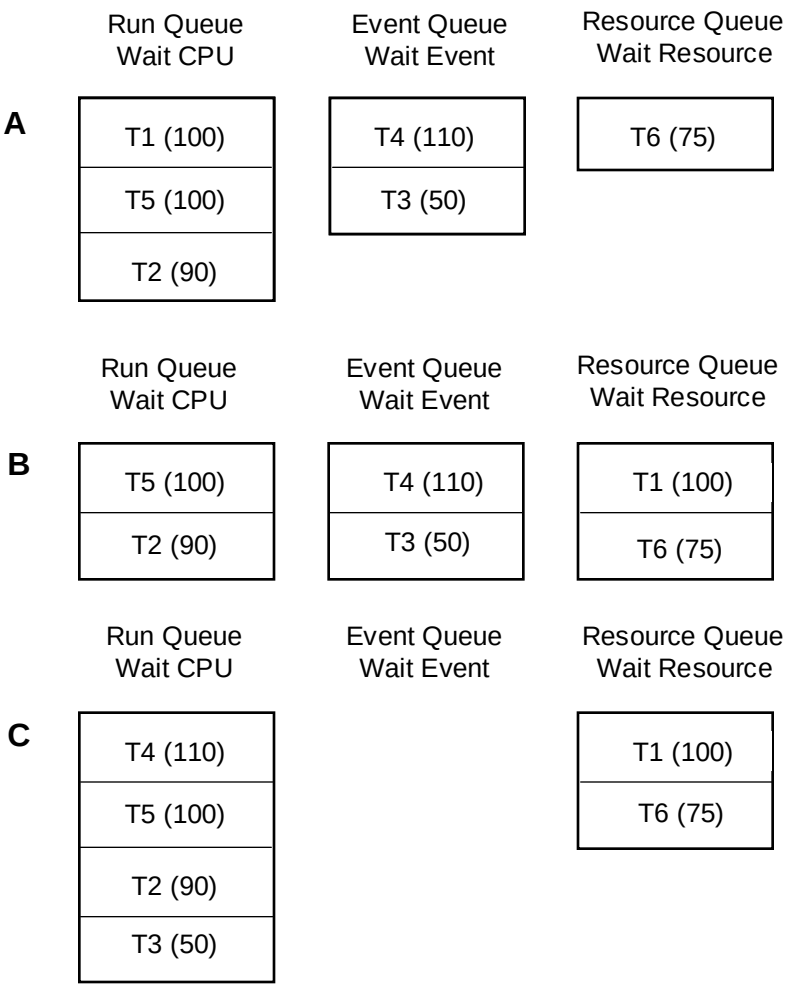

Figure 2-10: Tasks shown in various wait queues according to priority order

In **A**, tasks T1 and T5 will take turns getting all of the CPU time-slices (in a round-robin fashion) since these tasks are at the highest priority level. (This is assuming the MT! library was compiled with time-slicing enabled.)

In **B**, T1 has attempted to get a resource that is unavailable, so it is moved to a queue to wait for the resource. Task T5 now gets all of the CPU time-slices since it is the highest priority task waiting for the CPU.

In **C**, the event that T4 and T3 were waiting for has been set, so they both move out of the event queue where they were waiting back to the run queue. Task T4 will now get all of the CPU time-slices since it is the highest priority task waiting for the CPU.

Any task in the MT! system may start, kill, or prioritize a task. The MT! task management functions are:

```
runtsk(priority, task address, stack size, argument)
runtskss(priority, task address, stack size, stack base,
         argument)
klltsk(task ID)
pritsk(task ID, priority)
slttsk(task address)
scdtsk(void)
suspend(task ID)
reanimate(task ID)
```

runtsk	Dynamically defines a task to MT!.
runtskss	Defines a task with static stack.
klltsk	Dynamically kills a task.
pritsk	Dynamically prioritizes a task.
slttsk	Finds a task's slot number.
scdtsk	Cooperative task switch.
suspend	Prevents a task from running.
reanimate	Allows a suspended task to run again.

A task becomes a candidate for execution only when it is defined to MT! and in the run queue. The ***runtsk()*** function is used to define a task to MT!. MT! assigns a TCB and stack space to the task and then places the information required to control the task in the TCB. MT! moves the task to the run queue at the assigned priority level. The task is now ready for execution. The task is identified by its slot number, which is the index into the array of TCBs kept by the system. The slot number is returned by the ***runtsk()*** function and is passed to other functions as an argument to identify the task.

The ***klltsk()*** function is used to halt the task and remove it from its system slot (freeing the task's TCB). The ***klltsk()*** function also removes the task from its current queue, deallocates its stack and any LOCAL memory the task had allocated, frees any resources owned by the task, and flushes and closes any I/O streams opened by the task.

The ***pritsk()*** function allows the priority of a task to be changed dynamically. The ***pritsk()*** function changes the task's priority and causes its current queue position to be reevaluated.

The ***slttsk()*** function returns the system slot number of a task. The task is specified by address. If no address is given, MT! returns the slot number of the currently running task. If an address is specified, MT! searches the system task table for the task's slot number.

A task runs only when it is the highest priority task at the head of the run queue. A task that is running continues to run until its time-slice expires, it performs a call that causes it to wait (on a time delay, event, resource, message, etc.), it is preempted by a higher priority task, or it relinquishes the CPU with the ***scdtsk()*** function. A task may forfeit the remainder of its time-slice with the ***scdtsk()*** function.

A task in any queue can be prevented from running with the ***suspend()*** function. A suspended task will still receive any messages, events, or resources it is waiting on, but will be prevented from running until the ***reanimate()*** function is called for that task.

Switching and Running Tasks

MT! schedules tasks for execution by selecting the next task that can be run at the highest priority, i.e., the highest priority task in the run queue. The task runs until removed from the run queue (to wait for a resource, event, message, time period, etc.), it is preempted (by a higher-priority task entering the run queue), or until its time-slice expires (in which case it is rotated to a new position in the run queue). The time-slice (specified by the user) is usually in the range 10–100 milliseconds. The shorter the time-slice, the greater the percentage of CPU time consumed performing task switching. A very short time-slice causes the system to spend all of its time switching tasks (thrashing). On the other hand, a very long time-slice causes excessive delays between task executions.

When a task switch occurs because the task's time-slice expires or it is preempted, MT! selects the task at the head of the run queue as the new task to be run. If the new task also happens to be the running task (i.e., no task switch), then the process is simplified. In other cases, however, the context of the running task is saved and the context of the task to be run is restored. Since the run queue is organized in priority order, the highest priority task in the queue is the next task to be run. If more than one task is at this priority level, the tasks take turns receiving time-slices (in round-robin fashion, as explained earlier).

Blocking Preemption and Interrupts

The preferred method of controlling preemption is by carefully selecting task priorities when you design your application. If you have a high-priority task waiting for an event to be set, it will be in an event queue and not actually be running or requiring any CPU time until the event is set. When the event is set, it will be moved to the run queue and, if it has a higher priority than any other task there, will preempt that task and commence running. Typically this task would take some small finite amount of time to perform its function and then clear the event it waited on, loop back, and wait for the event to be set again, at which point the preempted task would pick up where it left off.

It is also possible to change the priority of a task with the ***pritsk()*** function, and block preemption by giving the current task the highest priority and then later dropping its priority level.

If you feel you must block preemption without regard to priority levels, there are two macros available to accomplish this.

The ***block_preemption()*** macro will stop all task switching until the ***unblock_preemption()*** macro is executed. Interrupts remain unmasked while preemption is blocked by this method.

NOTE: Most applications do not require these macros, and their heavy use is probably an indication that you do not yet understand the proper use of the other operating system features.

If you need to only block task switching for a very short time (e.g., a line or two of code), you could do this by globally masking interrupts. There are two macros provided in the file **depends.h** for doing this. The ***MASK_INTS()*** macro globally masks all interrupts, and the ***UNMASK_INTS()*** macro does the reverse. (Interrupts are normally enabled at all times in an MT! application except during actual interrupt processing.)

Assigning System ID Numbers

Tasks are assigned ID numbers dynamically, while events, group events, mailboxes, and resource ID numbers are assigned statically. To aid in assigning ID numbers statically to items such as resources and mail boxes, we have provided the file **usrasign.h**. Using this one file to maintain these IDs will help to avoid duplication. If MT! has been integrated with another product (such as USFiles) some IDs may be defined here, and these must not be duplicated or reused.

Events

Events are user-defined synchronization primitives used for basic communication between tasks. Implemented as a count byte, an event is treated as having only two states: *set* or *clear*. When the event status byte is non-zero, the event is considered set, and when the byte is zero, the event is clear. An event differs from other synchronization primitives because it causes all tasks waiting for the occurrence of the event to be rescheduled when it occurs. In other words, when an event is set by a task, any and all tasks waiting for that event to be set are returned to the run queue. When an event is cleared by a task, all tasks waiting for that event to be cleared are returned to the run queue.

Events might be used to indicate some state such as “motor-up-to-speed,” “buffer-ready-for-processing,” “safety-off,” etc.

Events are useful in waiting for user-specified activities and determining when they have occurred, but they don’t allow the passing of information between tasks. For this purpose, MT! uses messages, mailboxes, or pipes.

An interrupt routine or a task may be used to set, clear, increment, or decrement the event associated with an activity. Setting, clearing, incrementing, or decrementing the event causes waiting tasks to move to the run queue and possibly preempt the running task. Only tasks can check on or wait for events.

Events may be defined to be incremented automatically after a specified period of time (see ***period()*** and ***oneshot()*** in the *MultiTask! Library Reference* chapter). This is useful for stimulating activities such as periodically sampling a sensor input for integration.

The event numbers are assigned by the user, and range from zero to one less than the limit set by the configuration when MT! was compiled. Be careful in assigning event numbers, because the first `NUMPER` event ID numbers can be used for periodic and one-shot events. The order in which you assign event ID numbers can be significant.

Example

```
#define MOTORON_EVT  0   /* define event for motor on */
status = wteset(MOTORON_EVT, 0);
```

An event is given meaning by the programmer. MT! sets and clears events on request and arbitrates tasks waiting for those events to become set or clear.

Managing Events

As stated earlier, a task may set, clear, or check an event, wait for an event to be set, or wait for an event to be cleared. A *periodic event*, one that is set automatically by the ***period()*** function, is treated the same as any other event. A task may wait for a periodic event to be set the same as for any other event. Group events are also provided. These allow a task to wait for a combination of things to happen. Periodic events use the same ID numbers as events, but the group events use independent ID numbers. The number of events allowed in the application is specified in the file **mtcfg.h**. The symbol *NUMEVT* indicates the number of byte events for the application, and *NUMPER* limits the number of periodic events.

Events are user-defined and may represent any number of interrupt- or task-initiated occurrences. MT! manages these events on a real-time basis. When an event occurs, tasks waiting for the event are moved from the event wait queue to the run queue. Preemption will occur if a higher-priority task moves to the run queue.

There is no limit on the number of tasks that may wait for an event. The ***MTinitialize()*** function clears all events.

Event Functions

The event management functions are:

setevt(*event ID*)*	**clrevt**(*event ID*)*
incevt(*event ID*)*	**decevt**(*event ID*)*
chkevt(*event ID*)	**wtesetdec**(*event ID, timeout*)
wteclr(*event ID, timeout*)	**wteset**(*event ID, timeout*)
period(*event ID, period*)	**oneshot**(*event ID, time*)

NOTES: Functions marked with * can be used from an interrupt routine. For example, calling ***setevt()*** from an interrupt routine is achieved through the call ***MTqcmd_c***(SETEVT, *ID*). See the chapter on *MT! Internals* for more information on using these functions.
Event ID numbers are usually defined in **usrasign.h,** and the values must be coordinated with periodic event ID numbers.

setevt Sets the specified event value to 1, and moves to the run queue all tasks waiting for this event to be set.

clrevt Clears the specified event and moves to the run queue all tasks waiting for this event to be clear.

chkevt Returns the status of the specific event (numeric value 0..255). A zero status indicates the event is clear and non-zero indicates it is set.

incevt Adds one to the specified event value. Any time the event has a non-zero value, it is considered set.

decevt Subtracts one from the specified event value. The event is considered cleared any time its value is zero.

wteset If the event is clear, moves the task from the run queue to a wait queue, where it waits for the specified event to be set. An event is considered set when it has any non-zero value. The functions ***setevt()*** and ***incevt(),*** therefore, both set the event.

wtesetdec Moves a task from the run queue to a wait queue, where it waits for the event to be set. When the event is set, the task is returned to the run queue, and the event is decremented. (This saves the task from doing a separate ***decevt()*** call and thus saves time.)

wteclr If the event is set, moves the task from the run queue to a wait queue, where it waits for the specified event to clear.

Any of the wait event functions can optionally timeout if the condition is not met within a specified time. If the condition is already true when the wait call is performed, then the task remains in the run queue and does not sleep. See the *MultiTask! Library Reference* chapter for function-calling details.

period Specifies that an event is to be set periodically (automatically at a specified interval). At each interval, the periodic event will be automatically incremented. The ***period()*** function can also terminate such action.

oneshot Is used to increment an event only once at a specified system time. If a subsequent call to ***oneshot()*** is made for the same event before the previously specified time is reached, then the first ***oneshot()*** time will be replaced by the most recent call.

Two examples of synchronizing with events follow.

Example 1

```
char buffer[100];
void fill_task(void){
  for(;;){
        <Collect some data>
        buffer = <data>;
        setevt(my_event);
        wteclr(my_event);
  }
}

void consumer(void){
  for(;;){
        wteset(my_event);
        <empty buffer>
        clrevt(my_event);
  }
}
```

Example 2

```
void __interrupt my_isr(void){
   <clear interrupt>
   MTqcmd_c(INCEVT, my_event);
   ++mt_busy;
   MTsched_c();
}

void my_task(void){
   for(;;){
         wtesetdec(my_event);
         iprintf(''DING!\n'');
   }
}
```

Periodic Events

Periodic events are a mechanism that can be used to cause a task to run synchronized to a regular time interval measured in system clock ticks. This is done by using the ***period()*** function to set up an event to be incremented automatically at the desired interval. The task to be synchronized to this period then has a loop in which it performs a ***wteset()*** and a ***decevt()*** call or just a ***wtesetdec()*** function call.

The periodic event gets incremented on a regular interval. Whether the task is able to run immediately when the event is incremented depends upon whether any higher-priority tasks are in the run queue at the time the event is set.

If the task is blocked from running by higher-priority tasks, then it is possible that the event may be incremented several times before the periodic task is able to run. If the blocking task gives up the CPU, then the periodic task will run through its loop several times in immediate succession until the event is decremented back to zero.

If you wish to have the period task run only once under these circumstances, you can start its loop with ***wteset()*** followed by ***clrevt()*** rather than using ***wtesetdec()***.

The ***period()*** function is also used to deactivate a periodic event. See the *MultiTask! Library Reference* chapter for details.

The event number used with the ***period()*** function must be less than the configuration limit set by *NUMPER* in **mtcfg.h**. See the *Configuring MultiTask!* section of this chapter for more details.

One-Shot Events

One-shot events are used to signal an event at a particular value of the system time. These events differ from periodic events, because they execute only once. They do not repeatedly execute, unless the application repeatedly calls ***oneshot()***.

Group Events

Group events are similar to simple events but are composed of a word with significance to each bit. A task may wait for combinations of bits to be set, or a combination of specific bits to be set *and* others to be clear, and an *or* condition where any of selected bits are set.

The group event functions are:

setgrp(*group ID, mask*)*
clrgrp(*group ID, mask*)*
waitgrp(*group ID, set mask, clear mask, or mask, timeout*)
chkgrp(*group ID, result*)
GrpWakeValue

NOTE: Functions marked with * can be called from an ISR. To set a group event from an ISR, the call ***MTqcmd_c***(*SETGRP, ID, mask*) can be used. A similar call exists for ***clrgrp***.

setgrp Sets the bits you specify in a group event.

clrgrp Clears the bits you specify in a group event.

waitgrp Waits until the bits you specify to be set are set, and the bits you specify to be cleared are clear and any of the *or* bits you specify are set.

chkgrp Returns the current setting of a group event.

GrpWakeValue
Tests the group event condition that caused the task to wake up.

Group events can be useful for synchronizing several tasks. If several tasks each must wait until all of the others have arrived at a certain stage of processing, they may each set a different bit in a group event when they arrive, and then wait for all of the bits representing the other tasks to be set.

Another use can be in waiting for one of several different conditions to happen. Each condition could be represented by a bit in a group event, and the task could then wait for an *or* of any of those bits to be set. When the task wakes up, it could test what the wake-up condition was with the macro ***GrpWakeValue***. (This type of action could be achieved with messages also; the waiting task would wake for any message, and any number of other tasks could send messages to the mailbox at which that task was waiting.)

The number of group events available to the application is specified by the symbol *NUMGEVT* in **mtcfg.h**. See the *MultiTask! Library Reference* chapter for function-calling details.

Example

```
void wait_for_group(void){
  for(;;){
        clrgrp(my_group, 0xFFFF);
        waitgrp(my_group, 0xF0F0, 0, 0, 0);
        < do processing >
  }
}

void signal_group(void){
  for(;;){
        < do processing >
        setgrp(my_group, 0xF0F0);
  }
}
```

Mailboxes

Mailboxes are places where messages and packets are queued. If a task makes a request to receive a message or a packet from a mailbox and none is available, the task waits in a mailbox queue for a message or packet to arrive at the mailbox.

When you send a message or packet, a message header is automatically allocated to link the message pointer into the mailbox. The total number of message headers and thus the total number of messages that can reside in all mailboxes is set by the configuration parameter *NUMMSG* in **mtcfg.h**. When a message is received, the message header is deallocated and is available for reuse. When *NUMMSG* messages are outstanding (i.e., have been sent but not received), no more messages can be sent until one is received.

A per-mailbox message limit is implemented and is set by *MBXLIMIT*. When *MBXLIMIT* messages reside in a mailbox, that mailbox will accept no more messages until one is removed. This prevents a task from sending messages to a mailbox at which they are not being received (receiving task blocked), and from consuming all message headers, thus preventing other tasks from being able to send any messages.

The specific mailbox number to use to acquire messages for a task is arbitrarily defined by the user. It is often convenient to assign a task's slot number (*TASK_ID* of task) as the mailbox number the task will use, especially when many tasks will be receiving messages. The task's slot number can then be used to reference a specific mailbox for sending messages to that task.

Mailboxes are independent of tasks and may store messages for more than one task. Mailboxes are referenced by mailbox number. Any task may send or receive through any mailbox. It is often useful to have several tasks sending messages to the same mailbox, but it is rarely useful to have more than one task receive messages from a particular mailbox. When multiple tasks are waiting for a message at the same mailbox, the highest priority task will be the one to get the next message.

Mailbox Functions

chkmbx(*mailbox ID*)
flushmbx(*mailbox ID*)

chkmbx Returns the number of messages or packets currently in a mailbox.

flushmbx Discards all messages and packets in a mailbox and wakes any waiting tasks.

Messages

Communication between tasks in the MT! system may be accomplished by message passing. The messages are left and accessed through mailboxes. A task may leave or access a message at a mailbox.

Message Functions

chkmsg(*mailbox ID*) (***chkmbx*** is now preferred)
putmsg(*mailbox ID, message pointer, priority, timeout*)
sndmsg(*mailbox ID, message pointer, priority*)*
rcvmsg(*mailbox ID, timeout*)

chkmsg Checks if any messages are in mailbox (***chkmbx*** is now preferred).

putmsg Sends a message, and waits if mailbox is full.

sndmsg* Sends a message and returns an error if mailbox is full.

rcvmsg Waits for and returns the next message to arrive at a mailbox.

NOTE: ***rcvmsg()*** is used to receive both messages and packets. See ***rcvmsg()*** in the *MultiTask! Library Reference* chapter for details.
The * indicates that ***sndmsg()*** can be invoked from an ISR by using ***MTqcmd_c(****SNDMSG, ID, pointer, priority****)***.

A message is passed as a pointer to any user-defined data. The data that are pointed to are not copied, only the pointer is passed.

See also: In this chapter, see *Packets* in the *Mailboxes* section for information on passing copies of data through mailboxes; also see *Pipes* in the *Stream I/O* chapter.

One of the features of MT! is that these messages are assigned a priority. Messages have a priority (0..255) and are queued at the mailbox in priority order. Messages with the same priority are queued in a first-in-first-out (FIFO) order. The highest priority message in the queue is the one that will be returned by the next ***rcvmsg()*** function call. There is also a *super priority* (`SUPERPRI`) that can be assigned to a message that will always force it to the front of the mailbox even if the mailbox already contains a 255-priority message.

When a task requests a message from a mailbox, the highest priority message is returned to the task. When a task sends a message to a mailbox, the highest priority task waiting at the mailbox for a message receives the message and is made runnable. At any specific message priority, messages are returned in FIFO order.

A special feature of the ***sndmsg()*** function can be used to suspend the task sending the message immediately after the message is sent (see ***sndmsg()*** in *MultiTask! Library Reference*). This is useful when sending a message to a server task. The server task will process messages, performing some service as instructed by the message. If the sender of the message (requester of the service) should wait until the service is complete, the sender can have ***sndmsg()*** automatically suspend. The server task would be written so as to ***reanimate()*** the requester task at the appropriate time. The task ID (slot number) of the sending task needs to be known to the receiver so it can issue a ***reanimate()*** call to the sender at the appropriate time. The easiest way to identify the sender is to include its ID in the message.

Example

```
#define SERVERMBX   1     /* mailbox to use */
typedef struct {
   TASK_ID     slot;
   int   function;
   char returned_data[32];
}SERVERMSG;

void sender_task(void)
{
SERVERMSG p;
   p.slot = cur_task;            /* our ID */
   p.function = 2;         /* some meaningful value */
   sndmsg(SERVERMBX, &p, 100|MSGSUSPEND);
   /* now process p.returned_data from server */
}

void server_task(void)
{
SERVERMSG *p;
   for(;;){
       p = rcvmsg(SERVERMBX, 0);
       /* process data here, return value can be
          written into packet */
       reanimate(p->slot);       /* wake caller */
   }
}
```

The ***putmsg()*** and ***rcvmsg()*** functions can specify a timeout value in system clock ticks when waiting. If the timeout expires before they can put/receive a message, then they cease waiting and return an error indication.

Packets

Packets are a form of message that can also be passed through a mailbox. Sending a packet is similar to sending a message, except that memory is allocated to hold a copy of the message, the message data is copied to that memory, and then a pointer to the copy of the data is sent to the mailbox. The receiving task receives the pointer to the copy of that data and, when finished processing it, must release the packet memory with the ***relpkt()*** function. Since packets are exchanged through mailboxes, they have the same priority, mailbox limit, and queuing characteristics as messages.

Packet Functions:

```
putpkt(mailbox ID, message pointer, priority, size,
        memory pool, timeout)
sndpkt(mailbox ID, message pointer, priority, size,
        memory pool)
relpkt(packet address)
rcvmsg(mailbox ID, timeout)
```

putpkt	Sends a packet and waits if the mailbox is full.
sndpkt	Sends a packet and returns an error if the mailbox is full.
relpkt	Releases the packet memory.
rcvmsg	Waits for and receives a packet.

NOTE: ***rcvmsg()*** is used to receive both messages and packets. See ***rcvmsg()*** in the *MultiTask! Library Reference* chapter for details.

Pipes can transfer data faster than packets, but only from a single task to some other task, i.e., one sender to one receiver.

Many different tasks can send packets to the same mailbox where some other task may receive them, i.e., many senders to one receiver.

The ***putpkt()*** and ***sndpkt()*** functions take an argument that specifies where the memory is to be allocated from. This argument can be from *COLOR0* to *COLOR2,* or a buffer pool ID number from 0 to *NUMPOOLS*-1. If *COLOR0, COLOR1*, or *COLOR2* is used, then the memory is allocated from the variable-size allocation pools with a ***reqmem()*** function call, and ***relpkt()*** will release the memory with a ***relmem()*** function call. If a number 0..*n* is given, then the memory is allocated from the buffer pool specified by that number with a ***reqbuf()*** call, and ***relpkt()*** will release the memory with a ***relbuf()*** function call. In either case, the memory pool must be initialized with either the ***Mtmeminit2()*** or ***init_mem_pool()*** function, as appropriate, before the ***putpkt()*** and ***sndpkt()*** can be used successfully. If the memory pool in use cannot allocate enough memory to send a packet, then ***sndpkt()*** and ***putpkt()*** will fail.

Example

```
MTmsg_t const *message = (MTmsg_t *) "Test Message";
struct ourpkt{
   PKT_HDR h;
   char info[100];
};

void sender(void)
{
   if( sndpkt(MBX_A, (void *)message, 100,
         sizeof(message), COLOR0) )
         /* ERROR */
}

void receiver(void)
{
ourpkt *pkt;
   pkt = (struct ourpkt *)rcvmsg(MBX_A | PKTRCV, 0);
   /* Handle pkt->info */
   if( relpkt( (PKT_HDR *)pkt) )
         /* Error releasing packet */
}
```

Resources

Resources are nesting semaphores that provide a mutual exclusion mechanism (referred to as a *mutex* in some literature). For example, the computer display screen or keyboard can be resources, but they must be defined (a resource number assigned) by the user. A task is assigned exclusive use of a resource. When the task is finished with the resource, it is released to the system for use by another task. When a resource is released, the highest priority task waiting for it is assigned the resource.

Any task may request, get, release, or check a resource. MT! manages resources on a priority basis. The highest priority task waiting for a resource will be assigned the resource when it is available.

Resource Management Functions:

```
reqres(resource ID)
getres(resource ID, timeout)
relres(resource ID)
chkres(resource ID)
```

reqres	Requests a resource.
getres	Waits for and gets a resource.
relres*	Releases a resource.
chkres	Checks who owns a resource.

NOTE: In MT!, resources are referenced by user-assigned resource numbers. We recommend defining these ID numbers in the file **usrasign.h**.
The * indicates that ***relres()*** can be called from an ISR via ***MTqcmd_c(***RELRES*, ID)*.

Only one task may own a resource at a time. Any other task trying to get a resource while it is owned by another task is placed in a

resource wait queue until the resource becomes available. A single task may acquire a resource several times. It must then release that resource an equal number of times before it is available to the system. Resources enable tasks to share memory, code, hardware, or other user-associated items.

The ***getres()*** function moves a task to the resource wait queue if the required resource is not available. When the resource becomes available to the task, the task is moved back to the run queue, preempting any lower-priority task that is running. When more than one task is waiting for the same resource, the task with the highest priority in the resource wait queue will be the first to obtain that resource. The ***getres()*** function may be specified with a timeout delay. If the delay expires before the resource is available, then the task that made the ***getres()*** call is reactivated and returned an error status indicating that the timeout has expired. (The task is not assigned the resource when the timeout expires.)

The ***reqres()*** and ***getres()*** functions are similar. With ***reqres()***, however, the task is not delayed if the resource is unavailable. Instead, MT! indicates that the resource is not available.

The ***relres()*** function releases the resource and assigns it to the highest priority task waiting for it (if any).

The ***chkres()*** function returns the ID of the task that owns the resource, or zero if no task owns it.

Care should be taken when requesting and releasing resources. For example, assume that Task 1 owns resource A and Task 2 owns resource B. If Task 1 now attempts to acquire resource B and Task 2 attempts to acquire resource A, then a “deadly embrace” results. Both tasks are placed in a queue where they wait for a resource to become available. Since both tasks are waiting, neither task runs and the resources are never released.

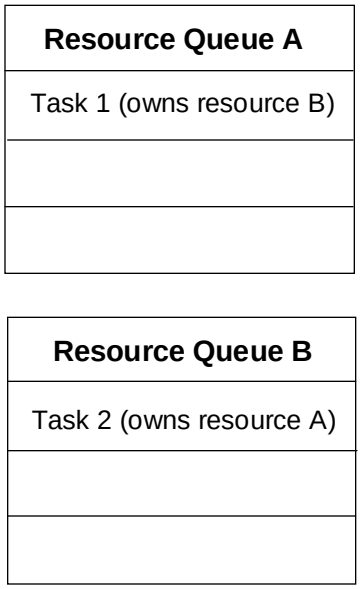

Figure 2-11: Deadlock or “deadly embrace”

To avoid the possibility of a deadly embrace or deadlock, resources already owned by a task should be considered when the task attempts to acquire an additional resource. If acquiring ownership of any of these resources could result in a deadly embrace situation, the resources should either be released or the acquisition of the additional resource should be conditioned on the resource’s availability. In the case shown above, this deadlock would have been prevented simply by both tasks requesting the resources in the same order.

Similar to events, resources are used by tasks. Only one task may own or control a resource at a time. Like an event, a resource is a synchronization primitive. Unlike an event, however, a resource is used to characterize an item that a task may want to use as being available or unavailable.

A task waiting for a resource becomes suspended (goes to sleep) if the resource is unavailable (if it is being used elsewhere). The task wakes when the resource becomes available. When a task using a resource finishes with and releases it, only one task waiting for the resource wakes. The task that wakes first is the one with the highest priority. When a resource is released by a task, the highest priority task waiting for the resource is given the resource and made runnable.

A resource is user defined. Examples of resources are a data buffer, a line printer, or a code segment. The number of resources available to the system is determined by the value of `NUMRES` in the file **mtcfg.h**.

See also: The discussion of ***block_preemption()*** for another method of protecting specific operations.

Memory Management

MT! has two methods for managing memory. Up to three heaps can be used to handle variable-length memory blocks, similar in behavior to ***malloc()*** and ***free()***. Alternatively, MT! can allocate memory from pools of fixed-length buffers. The fixed-buffer allocation operates considerably faster than the variable-size block memory allocation since it does not need to search for a block of adequate size, and when releasing a buffer no coalescing of fragments is needed. In the next few pages we will refer to the variable-size blocks as *memory blocks* and the fixed-size blocks as *buffers*.

MT! uses memory in blocks, which are contiguous groups of bytes. When a task requests a memory block, MT! returns a block that is at least as large as the memory block requested. MT! flags an error if available memory is inadequate. This process may require MT! to split a large memory block, potentially creating memory fragmentation. When a task releases a memory block, MT! recombines the released block with existing blocks, alleviating memory fragmentation.

MT! also provides simpler and faster fixed-size memory *buffer* allocation. These buffers are arranged as arrays of fixed-size memory

blocks called *pools*. Any number of memory pools may be defined, with each containing any number of buffers. The buffers within any pool are all of a fixed equal size. The memory fragmentation problem does not arise with this scheme and therefore no recombining of memory buffers is necessary.

Variable-Size Blocks

During initialization, blocks of contiguous memory may be released to any of the three heaps maintained by the heap manager. The free memory resides in RAM and is defined by the user through the ***MTmeminit2()*** function. The ***MTmeminit2()*** function must be called at least once before any memory will be available from the ***reqmem()*** function. These memory blocks are maintained by the memory manager in a linked list ordered from lowest to highest address. In other words, a memory block with a lower starting address is linked into the list ahead of a memory block with a higher starting address. See Figure 2-12.

NOTE: The current code requires that sections of memory be passed to ***MTmeminit2()*** in either ascending or descending order.

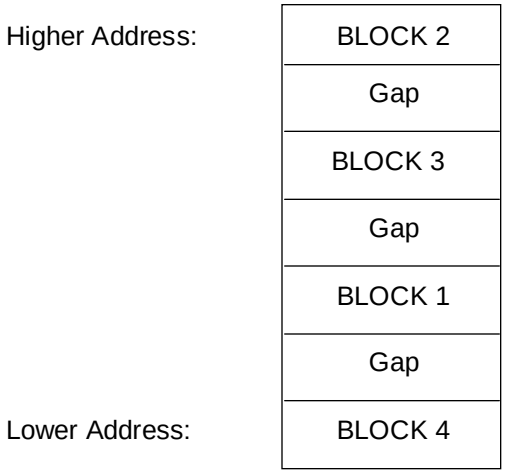

Figure 2-12: Memory after ***MTmeminit()***

When a request is made for memory, MT! checks the blocks in its linked list and processes the request with the first block larger than the requested size. The block is split and the piece at the high-memory-address end of the block is returned while the remaining memory is left in the linked list.

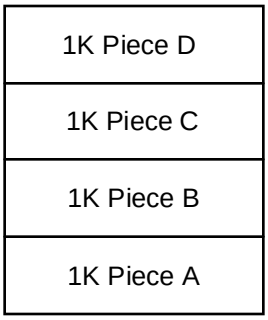

Figure 2-13: Diagram of a 4K block of memory

For example, assume that the 4K block of memory shown in Figure 2-13 is the only block being managed. MT! processes a series of four requests for 1K blocks of memory by returning pieces A, B, C, and D in that order. The following illustration shows what happens when the blocks are returned in the order A, C, D, and then B.

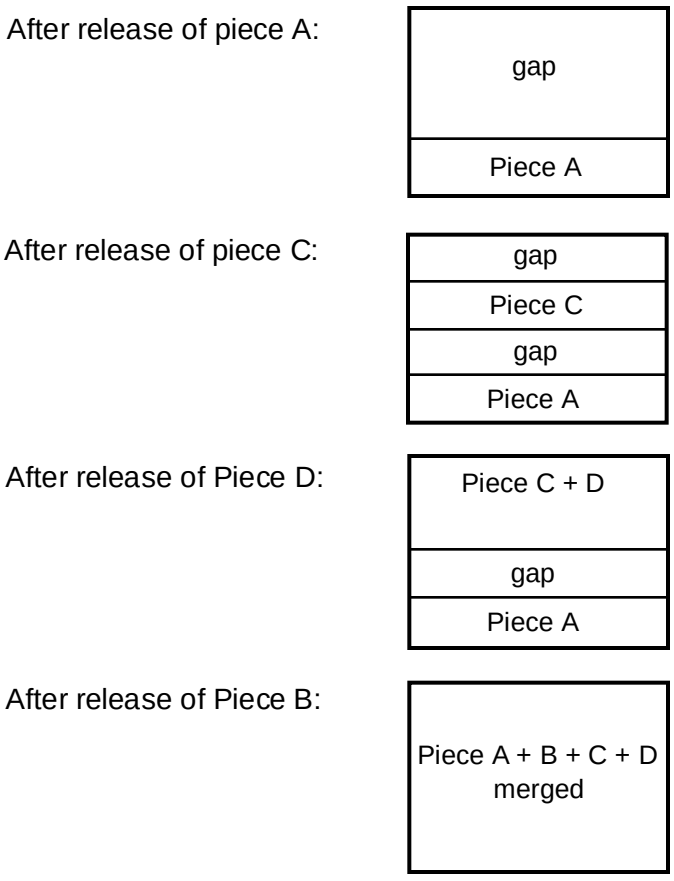

Figure 2-14: Blocks A, C, D, and B returned

When a memory block is released, it is recombined with as many free memory blocks as possible to limit free-memory fragmentation.

Variable-Size Block Memory Management Functions

In order to provide a distinction in terminology between the two types of memory allocation in MultiTask!, we refer to the fixed-size memory blocks as "buffers" and the variable-size allocation as "blocks."

MTmeminit2(*memory address, size, color*)
MTmeminit(*memory address, size*)
reqmem(*memory type, size*)
relmem(*memory address*)
chkmem(*color, info*)

MTmeminit2 Adds a new memory block to the managed memory pool.

MTmeminit Adds a new memory block to the *COLOR0* memory pool.

reqmem Requests a free memory block.

relmem* Releases a free memory block.

chkmem Performs an integrity check and reports.

The ***reqmem()*** function requests a block of memory from the system, the ***relmem()*** function releases a block of memory to the system, and the ***chkmem()*** function checks the memory integrity and reports the amount of free memory currently available for the *COLOR* (pool) specified.

NOTE: The * indicates that ***relmem()*** can be invoked from an ISR by calling ***MTqcmd_c(****RELMEM, address****)***.

The ***reqmem()*** and ***relmem()*** functions provide a memory management facility similar to the C functions ***malloc*** and ***free***. MT! maintains a linked list of the free blocks of memory, and fills a request for memory by finding the first block large enough to satisfy the request. MT! initially has zero bytes of free memory available. The ***MTmeminit2()*** (or ***MTmeminit()***) call must be used to release at

least one block of memory to MT! before using ***reqmem().*** Since the ***runtsk()*** call uses ***reqmem()*** to allocate stack space for a new task, an initial ***MTmeminit()*** call is required to release some memory before a successful call to ***runtsk()*** can be made.

Each memory block managed by MT! is preceded by a structure of type *MEM_DEF* (defined in the file **mtcfg.h**). The structure maintains the block size, owner, and a link to the next block in the chain. In general, this header structure can be ignored since MT! returns a pointer to the actual usable memory for a ***reqmem()*** call, and the same pointer is passed back to ***relmem()*** to release the memory. If you request a block of memory with ***reqmem()*** and then write outside the bounds of that block, however, you will likely be overwriting the *MEM_DEF* structure for that block and will corrupt the memory system.

The size of a *MEM_DEF* structure is two times the `sizeof(void *)` if local memory tracking is turned off, or four times `sizeof(void *)` if it is on. If you do not use ***reqmem()/relmem()*** in your code, you may want to use ***runtskss()*** to launch tasks. See the description of ***runtskss()*** in the *MultiTask! Library Reference* Chapter.

There are two types of memory that a running task may request: `GLOBAL` or `LOCAL`. `LOCAL` memory belongs to the task that requests it and normally should not be accessed by another task. If local memory tracking is enabled, then `LOCAL` memory is automatically released back to the free memory pool when the owning task is terminated (i.e., the task is killed by ***klltsk()*** or comes to the closing brace of the task's ***main()*** function).

`GLOBAL` memory, on the other hand, is not automatically released to the system when the task that requested it terminates.

Any time it is desirable to release a block of memory that was requested by a different task, the request should be made for the `GLOBAL` type of memory. If you want a block of memory to remain accessible after the task that requests it terminates, request `GLOBAL` memory.

Example: Task A requests a block of `GLOBAL` memory to hold a data buffer (perhaps some input from a sensor). Task A then sends a

pointer to the data buffer as a message to Task B. Task B receives and processes the data buffer. When Task B is finished processing the data buffer, it releases the data buffer memory back to the system with the ***relmem()*** call.

If Task A always used the same fixed address to hold its data buffers, it would need to use a *resource* or *event* mechanism to know when Task B had finished processing the data so that Task A could again use the buffer. If, however, Task A requests a new block of memory for each data buffer, it may operate without this overhead and may at times have several data buffers queued up waiting to be processed by Task B.

Memory that is to be used only by the task that requests it should be requested as `LOCAL` memory. Both `LOCAL` and `GLOBAL` memory can be released at any time with a call to ***relmem().*** In the case of `LOCAL` memory, any requested `LOCAL` memory that is not explicitly released by the requesting task will be automatically released by the ***klltsk()*** function when the task terminates.

The variable-size memory allocation functions ***reqmem()*** and ***relmem()*** now support three different pools of memory, referred to as *colors*. Each color can actually contain multiple discontiguous blocks of memory at any address. When memory is allocated, memory blocks are split as necessary on a first-fit basis, and memory is recombined into the largest contiguous block possible when it is released. Multiple colors allow you to designate one color as normal RAM, another as battery-backed-up RAM, etc. Another possible use is in a design where you have requests for many small blocks of memory and also requests for large blocks. In this situation you may be able to avoid fragmentation problems by making the large requests from one pool (color) and the small requests from another. You might also want to divide memory into different pools so that if requests from one are depleted, it will still leave memory available in another pool assigned to more vital functions. The new memory algorithm forces a limit of three colors specified by the predefined labels `COLOR0`, `COLOR1`, and `COLOR2`.

MT! itself always requests memory from *COLOR0*. Task stacks, MTFILE structures, and serial I/O and pipe buffers are allocated from *COLOR0* by the system.

The behavior of LOCAL memory is a *compile-time* option controlled by bit 1 of the #define label *STCFG*, which is passed in the *CFLAGS* by the makefile. When bit 1 of *STCFG* is set, LOCAL memory is active. When a task dies (or is killed), any memory blocks requested as LOCAL memory by the task are automatically released along with the task's stack space. Tracking LOCAL memory doubles the size of the header on each block.

When bit 1 of *STCFG* is zero at compile time, LOCAL memory requests are not linked to the requesting task, and consequently memory blocks requested with the LOCAL attribute do not get automatically released when the task dies. (The task stack space, however, is still released.)

This case has two performance advantages: First, the request and release are both faster, and second, the number of bytes of overhead allocated for each block is cut in half, making for more efficient use of memory. You should consider this option if you are making heavy use of dynamic memory allocation (such as when using ***reqmem()/ relmem()*** in place of ***new/delete*** in C++).

It is possible for ***chkmem()*** to perform an integrity check of both allocated and free memory blocks at the expense of taking more time. ***Chkmem()*** is able to spot most memory corruption situations.

To initialize the multiple colors of memory, an ***MTmeminit2()*** function has been added that includes the color specification. If you are using only one color, or referring to *COLOR0*, you do not need to specify the color, which is done by using ***MTmeminit()***.

The ***relmem()*** function does not require the color of the pool to which the memory block will be returned. The block will automatically be put back into the color pool from which it was requested.

Examples

```
char freemem[10000];
char *p;

/* initialize color 0 */
MTmeminit(freemem, 10000);
MTmeminit2(freemem, 10000, 0);   /* same */

/* init color 1 */
MTmeminit2(freemem, 10000, COLOR1)

/* get GLOBAL COLOR0 memory */
p = reqmem(GLOBAL, 100);

relmem(p);     /* release memory */

/* get COLOR1 memory */
p = reqmem(GLOBAL|COLOR1, 200);
relmem(p);

/* get local COLOR0 memory */
p = reqmem(LOCAL|COLOR0, 17);
relmem(p);
```

Existing calls to ***MTmeminit(), reqmem()***, and ***relmem()*** do not require any change. Whenever the color is not specified, *COLOR0* will be used. The configuration parameter *NUMCOLORS* in **mtcfg.h** specifies the maximum number of colors you will be using. This can be set to any value between 1 and 3. (If you remove all use of heaps, then a value of 0 can be used.)

Fixed-Size Blocks

In order to provide a distinction in terminology between the two types of memory allocation in MultiTask!, we refer to the fixed-size memory blocks as “buffers” and the variable-size allocation as “blocks.”

Functions for Managing Fixed-Size Memory Buffers:

```
init_mem_pool(pool ID, address, block size, block count,
              type)
del_pool(pool ID)
reqbuf(pool ID)
getbuf(pool ID, timeout)
relbuf(pool ID, buffer address)*
ireqbuf(pool ID)*
ireqbuf_c(pool ID)*
chkbuf(pool ID)
```

init_mem_pool Initializes a memory pool.

del_pool Deletes a memory pool.

reqbuf Requests a buffer from a pool.

getbuf Requests a buffer from a pool, and waits if one is not available.

relbuf* Returns a buffer to its pool.

ireqbuf* Requests a buffer from a pool (for use only by an ISR written in assembler).

ireqbuf_c* Requests a buffer from a pool (for use only by an ISR written in C).

chkbuf Returns the number of buffers currently available in a memory pool.

NOTE: The * following a function name indicates that this function can be used from an ISR. In fact, ***ireqbuf()*** and ***ireqbuf_c()*** can only be called from an ISR. The following descriptions of these functions explain how they are used. As an example, to call ***relbuf()*** from an ISR, one must use ***MTqcmd_c***(`RELBUF`, `ID`).

Memory buffers are allocated from *pools*. Within each pool, all buffers are of the same fixed size. Different pools are independent and may have different buffer sizes. The buffers in a pool are essentially an array of fixed-size memory blocks.

You must call the ***init_mem_pool()*** function to initialize each memory pool before calling any of the other functions for that pool. The ***init_mem_pool()*** call is passed the number of buffers in the pool, their size, the memory address where the pool begins, and the pool type. The number of memory pools in the system is user defined by the parameter `NUMPOOLS` in the file **mtcfg**.**h**. The pool *type* defines the pool as either a `TASK_POOL` or an `ISR_POOL`, meaning request for buffers can be made either by ordinary tasks or by ISRs only. In either case, the buffer can be later released by either a task or an ISR.

The ***reqbuf()*** function requests a buffer from a pool and returns either a pointer to the buffer or a null pointer if a buffer is not available.

The ***ireqbuf*** and ***ireqbuf_c()*** functions act the same as ***reqbuf()*** except that they are for use only by an ISR written in either assembler or C respectively. These functions operate faster than the ***reqbuf()*** call and do not check if the pool number passed to them is valid. (Note: ***ireqbuf()*** is an optimized version of ***ireqbuf_c()*** to be called by an assembly language ISR rather than a C language ISR, and may not be present on all platforms. If it is not present, you may construct it by hand by optimizing a copy of the ***ireqbuf_c()*** function.)

The ***getbuf()*** function performs the same as ***reqbuf()*** except that if a memory buffer is not immediately available, the requesting task will wait until one becomes available. You cannot use this call from an ISR, which would not make any sense anyway.

The ***relbuf()*** function returns a buffer to its memory pool. Buffers must be returned to the same pool from which they were requested, but do not have to be returned by the same task that requested them.

The ***chkbuf()*** function returns the number of buffers currently available in a memory pool.

The ***del_pool()*** function de-initializes a memory pool, so that any further requests for memory buffers by a task will not be satisfied.

Time Management

Time management in the MT! system is based on the system clock. Any task may delay or wake a task or read the system clock. The time management functions, ***dlytsk()*** and ***period(),*** are based upon a number of system clock ticks from the time the call is made. Two calls, ***delay_until()*** and ***oneshot()***, are based upon a designated system tick count independent of when the call is made. The system time is kept internally as a count of clock ticks (at *CLOCKHZ* rate) since system initialization. The function ***get_sys_time()*** returns the current system tick time. By getting the current system time and adding a number of ticks to this time, a new future time is computed. The ***delay_until()*** function behaves exactly as ***dlytsk()***, except that it takes a system tick time as argument instead of the number of ticks or seconds, etc., to delay. The call ***oneshot()*** increments an event exactly like the ***period()*** function except that it takes an absolute tick time to increment the event and this happens only once. The ***period()*** and ***oneshot()*** functions are described further under the *Event Functions* section earlier in this chapter.

Time Management Functions:

dlytsk(*task ID, units, time*)
delay_until(*task ID, time*)
get_sys_time(*void*)
oneshot(*event ID, time*)
period(*event ID, period*)
waktsk(*task ID*)

dlytsk	Delays a task for a period of time.
delay_until	Delays a task until a specified system time.
get_sys_time	Returns the current system time.
oneshot	Increments an event at a specific system time.
period	Increments an event each time the period specified elapses.
waktsk*	Wakes a delayed task -- if task is not asleep, will prevent next task wait.

NOTE: The * following ***waktsk*** indicates that it can be used from an ISR. To call ***waktsk()*** from an ISR, one must use ***MTqcmd_c***(`WAKTSK`, `ID`).

These time functions make it easy to program a sequence of events that must occur at specific intervals from a starting point, when the system clock tick resolution is adequate.

Example: The system clock tick time is 10 milliseconds (i.e., *`CLOCKHZ`* = 100). At the start of a manufacturing process, additional steps must be taken at intervals of 50, 200, 250, and 800 milliseconds. The following code sequence should accomplish this.

```
void start_process(void)

{
tick_cnt_t start_time;
   start_time = get_sys_time();
   process1();
   delay_until(cur_task, start_time+5);
   process2();
   delay_until(cur_task, start_time+20);
   process3();
   delay_until(cur_task, start_time+25);
   process4();
   delay_until(cur_task, start_time+80);
   process5();
}
```

The system clock is stimulated by a timer interrupt supplied by the application. Each task receives a time-slice equal to the frequency of the clock tick times the user-specified parameter *NUMTCK*.

The parameter *CLOCKHZ* specifies the number of clock interrupts per second, i.e., the clock frequency in hertz.

NUMTCK and *CLOCKHZ* are defined in **depends.h**. For more information, refer to the section on *Configuring MultiTask!* later in this chapter.

The ***dlytsk()*** function moves a task from the run queue to the time delay queue. When the task's specified delay expires, the task will move back into the run queue.

The ***waktsk()*** function also moves a task from the time delay queue to the run queue. If the task is not in a wait queue, then a flag will be set so that the next attempt to delay the task will not succeed. This is particularly useful when the task will be woken by an ISR.

Miscellaneous Functions

Profiling

The MT! task profiling functions ***clr_profile()*** and ***get_profile()*** are not included in the library in the default configuration. They can be included by selecting them with the configuration program, or editing **mtcfg.h,** or setting bit 2 in the *STCFG* makefile variable before compiling the MultiTask! library.

When profiling is enabled, MT! will keep track of which task is running each time the clock tick occurs. The function ***clr_profile()*** allows you to reset this count and ***get_profile()*** enables you to get a copy of this information. These counts will allow you to compute the percentage of CPU time taken by each task and an approximate execution time for any task. Since these counts are of a statistical nature, you can obtain greater accuracy by allowing a set of tasks to operate continuously for a relatively long period of time before inspecting the profile information.

The profile count for task 0 represents the amount of idle time when no task is running. Profile counts are maintained by MT! in an array of type `profile_t` (32-bit values). The number of entries equals *NUMTSK*+1.

The profile numbers can be distorted when some tasks are synchronized to the clock ticks and others are not. In this case, some tasks may never be active during the clock tick and are therefore not counted.

Environment

With the release of the USFiles file system, a new dynamic task load capability (load task from disk) was added. This introduced a need for a method for the loaded task (which was compiled separately from the main application) to find a valid free mailbox number or event number, etc., or to find if a certain task was running in the main application or some similar detail.

When the entire application was linked together during development, these things could always be statically defined with no difficulty. This may be difficult to do when a task is loaded separately (perhaps developed after the main application was installed). To address this need, we have added an *environment variable* mechanism similar to DOS or UNIX and bit-map allocation (*scoreboard*) functions.

The environment mechanism differs slightly from what is in either DOS or UNIX in that the environment variable value is a (`void *`) type rather than a (`char *`). This makes it simpler to pass numeric or other information besides strings. For example, you might want to put a task name and the mailbox number it is using in the environment space. To do this, you might pass a (`char *`) to the task name string, and an (`int *`) to an `int` containing the task slot number or its mailbox number, etc. The actual strings or other data pointed to are not copied, rather only their pointers are stored in the environment table. This means that the data pointed to must remain constant while the string is in the environment. The environment pointers should point to (`const`) or global storage that will remain constant throughout the time that the environment entry exists. The number of variables that can be stored in the environment is set by the configuration program in the parameter *MTENVSIZE* in **mtcfg.h**.

Scoreboard

The ***acquire()*** and ***release()*** functions implement bit-map oriented bit set and clear for any user-defined table. The ***acquire()*** function returns the bit number (0..*n*) of the next clear bit in the specified table and sets that bit. The ***release()*** function clears the specified bit in the specified table. This can be used if you want to implement dynamic assignment of mailbox numbers, resources, or any other item. For example, you might initialize a table to represent available mailbox numbers and then do an ***acquire()*** call to find the next available one for use. This again is mostly applicable when dynamically adding a separately compiled task to a running system.

Time of Day

MT! provides a sample ***time_keeper()*** task that maintains a 24-hour system clock with hours, minutes, and seconds based on a clock interrupt provided by the application. The ***time_keeper()*** task must be started with a ***runtsk()*** call before the clock will function. You can set or read the system clock using the ***setclk()*** or ***getclk()*** functions.

setclk() Sets the time-of-day clock.

getclk() Reads the time-of-day clock.

The **coretest.c** program starts the ***time_keeper()*** task and tests ***setclk()*** and ***getclk().*** You can examine this program for an example of usage.

The get_tcb() function

The ***get_tcb()*** function returns a copy of the task control structure (`typedef TASK_DEF`) for any task. The most useful structure members for debugging purposes are usually `task_que` and `que_link`, which are the queue index the task is currently in, and the link to the next task `TASK_DEF` in the queue. You should note that the *`task_sp`* value will not be current if you inspect the task control structure of the currently running task.

The ***get_tcb()*** function is of little or no use in current releases, since this information is usually better obtained by calling the ***mtdbg()*** debug function.

How to Design Your Application

Real-Time Application Guidelines

A number of guidelines should be kept in mind when writing multitasking applications. The processing work should be divided into small (generally single function), manageable tasks. Careful consideration should be given to task priority level with no more inter-task communication than necessary.

If a great deal of communication is required between tasks, it indicates either poor task splitting or an exceptional processing requirement that is not efficiently handled in a multitasking environment.

Information that must be shared between tasks should be passed as messages and events wherever possible, or through pipes. It's generally not a good practice to share variables between tasks. However, if a variable in memory must be shared, that variable should be treated as a system resource requiring each task to request exclusive access to the variable when required and releasing the variable when no longer needed.

By paying attention to inter-task communication and data sharing, you can eliminate the most common cause of bugs in a multitasking system.

Before You Start

You should have already read the MultiTask! *Overview* section to gain a basic understanding of the OS features before proceeding.

You should also have built all the test programs supplied and tried to run at least one of these on your target.

You should have a clear idea of what your application is required to do.

When planning your application, keep in mind that the **mtcfg.h** file is used to configure the MultiTask! environment by defining things like number of tasks, number of events, etc. The file **usrasign.h** should be used to statically define ID numbers for items such as mailboxes and events (among others).

The **mtbench.c** program supplied with the MultiTask! delivery will perform MT! function timings on your target. You should run this to get a good idea of how long various operations will take on your target platform. You will need to make sure the ***usrclk()*** interrupt is set to occur at the same rate as described by the *CLOCKHZ* parameter in order for the timings to be accurate. The timings are mostly for combinations of functions, the way they will be used. For example, if one task is waiting for an event to be set, and another sets the event, the timing loop includes the ***wteset()*** and ***setevt()*** functions as well as two task switch times that will be used to go full cycle. You will note that using events is generally a little faster than sending and receiving messages, which is generally faster than running and killing a new task. This information is useful when you are deciding how to structure your most time-critical operations.

Running **mtbench** will also provide RAM requirements for applications of differing complexity as well as sizes for particular MT! structures. If the symbol `STACK_FILL` is defined in the file **depends.h** (e.g. `#define STACK_FILL '+'`), then **mtbench** will determine how much stack space is left unused. This will allow you to determine a minimum stack size to use for your application. ROM sizing must be done by examining a map file, and this would be done independently of **mtbench**.

Defining Tasks

Before you can begin building your multitasking application, you must define what the individual *tasks* will be. This is an important design phase. Take your best stab at it; you may later need to refine this by splitting or combining some tasks.

The multitasking implemented by MT! is sometimes referred to as multi-threading, or lightweight multitasking where tasks share global memory, as compared to multiprocessing or UNIX-style multitasking where a process' memory is completely separate from other processes. As this is the case, the task switching and communication features are much faster and more efficient and more suitable for an embedded real-time application. With this architecture, tasks are often quite small, with some being only a few lines of code. As a rule of thumb, each task performs only one simple function and waits for one condition to stimulate it. There will, of course, be exceptions, but this is the breakdown of tasks that will most naturally fit the OS features.

When deciding what should be a task, use the criteria in the following sections.

What actions must be performed?

Usually each separate action naturally dictates a separate task, for instance updating a display, reading user controls (keypad, etc.), or controlling an external operation (servo, valve, etc.). Each interface (RS-232, GPIB, front panel, etc.), each process, and any shared functions (e.g. command processing) should have separate tasks. Interfaces that are bidirectional may need a task for each direction. See Figure 2-1 for a graphical representation of how these tasks might interact.

What stimulus will dictate those actions?

Even more important than dividing separate actions into tasks is the division by what the stimulus for those actions will be. Most of the MT! functions allow you to wait for a single condition and also optionally timeout if that condition does not occur within a specified time. The group event functions allow waiting for a combination of stimuli. Occasionally, to wait for a complex combination of things to occur, you may need to implement one or more helper tasks whose only duty is to wait for some condition and signal another task to do the real work.

What interrupt sources are available?

It is generally advantageous to use input interrupts rather than polling to stimulate tasks whenever possible. For instance, if you must read a switch closure, polling will require that some CPU time be expended to watch for the switch closure; but if the switch closure causes an interrupt that you can respond to, then no CPU time will be consumed until the switch is actually closed. Of course, if you must poll, you can easily set up a task using a periodic event to do this at a reasonable period.

The frequency of the interrupt will generally dictate how you should handle it. You don't want to cause task switching at a very high frequency or you will end up *thrashing* (using up all CPU time merely switching tasks). For example, if you have a serial port receiving at a high baud rate, it is better for the ISR to buffer a number of characters and then wake the task waiting for those characters rather than wake the task (cause a task switch) for every byte. This is exactly what the serial stream drivers supplied will do if a task requests a read of, say, 100 bytes (with ***mt_fread()***). The driver ISR will buffer each byte until the 100th and then wake the task. If the task only requests a read of one byte, then the ISR will wake it as soon as the byte is available.

Some very high-speed requirements may be handled directly by the ISR without using any MT! function calls. ISRs written in this way

are completely independent of the task activity and, in a sense, are higher priority than all tasks since normally interrupts are always enabled.

If the process to be initiated by an interrupt is relatively lengthy and infrequent, it may be most convenient and suitable to have the ISR merely wake a waiting task by setting an event and allowing the task to perform all the processing. Remember to disable that interrupt in the ISR and reenable it in the task.

In any case, system function calls from an ISR must always be made indirectly with the ***MTqcmd_c()*** function. With the exception of ***ireqbuf()***, only non-waiting functions that accept information may be used. See the *Interrupt Service Routine* section in the *MultiTask! I/O* chapter for details.

What is the most time-critical path?

The more time-critical operations require careful consideration. Usually, human interfaces are not that time critical. For instance, if you are updating a display for someone to read, the person looking at the display will not notice a 50-millisecond delay. But if you are controlling an electrical signal to another machine, 50 microseconds delay might be significant. Task priorities will affect their response time. Generally, a higher priority (relative to other tasks) will give quicker response times. Studying the timing information given by the **mtbench.c** program will help you estimate if an operation can be handled by a task or must be coded outside the operating system in an ISR.

What are the priorities?

Now is the time to make those “pie-in-the-sky” specifications realistic. If you are one of the lucky few who still has leftover CPU time after meeting all the design specs, then you will have plenty of latitude as to your actual implementation. If you determine that some tasks do not meet the desired response time, and/or all or nearly all of

the CPU bandwidth is used up, then the first thing to do is reevaluate your specification. Do all of the response times need to be as stringent as originally specified? Can you change the hardware to a faster CPU, or faster memory, or maybe a different processor altogether? When the answer to both of these questions is no, then the only fix left is to make the code more efficient.

To squeeze more efficiency out of the code, you need to look both at the way you code your tasks and the frequency of task switching. You may be able to merge two tasks into one and thereby reduce the number of task switches that take place. For example, if you have several things that must be done periodically at the same interval, it would be more efficient to have one periodic task perform all functions rather than a separate periodic task performing each function.

More predictable performance can be achieved by making the application completely preemptive, at least for high-priority tasks. In other words, every task will have a different priority, so time-slicing will not occur. When all tasks have different priorities, you have more control over how much of a task will execute before it yields the CPU (by waiting for an event or message, doing a timed delay, etc.). During time-slicing, you cannot predict if a task will get a full time-slice each time it runs. This is because a task may be scheduled very near the end of a time-slice due to the previous task's yielding. In this case, it will get only a fraction of the time-slice until the clock ticks again and it is rescheduled. If you have many tasks of the same priority, that task will wait until all of the others have run again before it gets any more CPU time. Because of this, you may find that a task appears not to be running. The more tasks you have of the same priority engaged in time-slicing, the worse this phenomenon may become.

Do not equate the importance of what your tasks are doing with priority. Just because the duties performed by all of your tasks seem of equal importance does not mean that they should have equal priority. Of course, round-robin time-slicing may be just what you want under some circumstances; however, to avoid unexpected behavior, we would recommend you avoid it except in cases where it really makes sense. In cases where you want equal priorities but no

time-slicing, you can configure MT! for this with a compile-time option.

Stack Sizing

The symbol `STACK_FILL` can be used to help determine a minimum stack size for tasks in your application. If you define `STACK_FILL` in **depends.h** (e.g. `#define STACK_FILL '+'`), then ***runtsk()*** will fill the stack with the symbol indicated. When ***MTterminate()*** is called, it will search through the stack of each remaining task and determine how much stack space was unused. Since this value will be printed via ***iprintf()***, console I/O must be operational to do this.

Reentrancy Considerations

Since a task switch may be instigated by an interrupt service routine, you should consider every task as if it were itself an interrupt routine when considering code reentrancy problems.

All operation system functions will block task switching in critical code sections to avoid these problems. This allows you to call any system service call from any task at any time without concern about interference with other tasks.

If you have more than one task accessing common data structures, a conflict can arise. This includes the case where two tasks call a common function, either one that you have written or a standard library function that accesses global data. (Examples of this are the standard C library functions ***malloc/free***, ***strtok***, and use of `errno`, among others.)

Consider the case where a linked list is being updated. If an interrupt causes a task switch to another task that will also update the linked list, there will be a critical window in the operation where the links could be corrupted.

One way to prevent this from happening would be to mask interrupts during the code that was manipulating the list. This would have the undesirable affect of preventing tasks that have nothing to do with the list from preempting during the time that interrupts were masked, as well as preventing interrupts from being processed for that time. If the time is very small, this is of little significance, but if the operation is lengthy it could have a detrimental effect on system performance.

Another way is to allow interrupts but to prevent preemption. This can be done via ***block_preemption()*** and ***unblock_preemption()***, and it will only work if the memory is shared between tasks, and no ISR accesses it.

A better way to control reentrancy for lengthy operations would be to assign a *resource* for the operations. This would block only another task from entering the critical area of code at the same time it was already in use. Task switching would still occur and non-conflicting tasks could operate unhindered. Interrupts would also be processed without delay.

The resource functions exist to take care of this sort of reentrancy problem. They are used not only to prevent reentering non-reentrant code, but also to prevent two tasks from trying to control the same hardware port simultaneously, or any similar situation. A resource provides *mutual exclusion* in execution of portions of tasks since only one task may possess the resource at any time.

Of course, if the resource can be accessed by an ISR, you must disable interrupts, either globally or just those specific to the device in question. If your critical code can be nested, you must save and restore the mask/enable state. Please see ***MTqcmd_c()*** and **depends.h** for assistance.

Task Activation

In an application designed with multitasking, all tasks operate more or less asynchronously much as if they were interrupt routines. In general, a task will be inactive, i.e., not using any CPU time, until it receives some signal that there is something to do. The signals can be thought of as interrupts. Just as a hardware interrupt activates an ISR, the software signal activates a task.

MultiTask! provides many types of signals that can activate a task. These include *events, messages, I/O,* and *time***.**

To make a task be activated by one of the types of software signals, the task must first be run and the task itself must perform a function call to cause itself to wait for the desired signal.

As you can see, in both ISRs and tasks, there is an initialization sequence that must be done before the interrupt or signal will cause the desired code to be run.

The various types of signals have properties far more versatile than an interrupt. When a task is waiting for any type of signal, it can also specify a time limit it is willing to wait. If the timeout (time limit) passes before the signal arrives, then the task will resume and can take alternative action.

If any of the types of signals are present when the task requests to wait for them, then the task immediately continues and no context switch takes place.

Task Activation via Events

The simplest form of signal is the *event*. The event simply signals that some condition is present. The condition is whatever arbitrary meaning you want to assign to the event. An event differs in nature from a hardware interrupt in that it is capable of activating any number of tasks simultaneously. Any number of tasks may wait for an event to be set or clear and, when the condition arises, all will be

activated (returned to the run queue) simultaneously. The highest priority of these tasks will run to completion, and then the next, and so on.

Each of the many tasks waiting for the event could have a different timeout period if desired.

An ISR often sets an event to signal a task to proceed. (Note: This is done in the manner described in the previous section on interrupts.)

Task Activation via Group Events

When it is desired to have a task wait for a combination of signals, or for any one of several signals, the *group event* is used. Like the event, the group event can activate any number of tasks waiting for the same conditions simultaneously.

A task waiting for a group event is like an ISR that only responds when several specific interrupts are pending simultaneously. Alternatively, it can act like an ISR that services any of several interrupts. More complex combinations are possible also, such as certain conditions present, some others absent, and one or more of yet some others.

Task Activation through Mailboxes

The *mailbox* provides a means of both activating tasks and passing them some data via a *message* or *packet.* Unlike the event and group event, only one task waiting at a particular mailbox will be activated when a message arrives. This will be the highest priority task waiting at that mailbox, which is not necessarily the first one to wait. Generally, however, only one task is made to wait at a particular mailbox.

The data can be passed as just a pointer by using a message, or as a copy of the original data with a packet.

The messages are queued in the mailbox, so several can be waiting. In this case, they are queued in order of the priority of the message that is specified when they are posted. Higher-priority messages will move ahead of those already in the queue.

Messages may be sent from many sources to the same mailbox, where they are queued for processing by a task.

Task Activation via Time

Time can be the activating signal for a task in a number of ways. All time-related services in MultiTask! are derived from the system clock interrupt. The smallest unit of time used is the *clock tick*, the period of time between two successive clock interrupts, which is 1/`CLOCKHZ` seconds.

Whenever a task is waiting for an event, group event, resource, message or packet, a timeout period expressed in clock ticks can be specified. After the specified timeout period passes, if the signal the task was waiting for has not occurred, then the task is reactivated by a time-derived signal with an indication that the timeout has expired.

A task can be made to wait only for a time signal by using ***dlytsk()*** to delay a specified number of ticks, seconds, or minutes. It can be made to wait until a certain system time (tick count) is reached with the ***delay_until()*** function.

A task can be made to run at regular intervals with the ***period()*** function. In this case, the task actually waits for an event, but the event is being automatically incremented at regular intervals based on the system clock.

A task can wait forever by using ***dlytsk()***. The wait (whether finite or infinite) can be canceled with ***waktsk()***. The ***waktsk()*** can occur before the ***dlytsk()***. If this happens, then the next call to ***dlytsk()*** for the task in question will be ignored. In essence, each task has a wakeup bit associated with it. If this bit is set and ***dlytsk()*** is called, the task will not enter the time delay queue, and the bit will be turned off.

Task Activation through I/O Functions

The *stream I/O* functions signal the task requesting I/O when the I/O is complete. A task may request to read data through a serial port or a *pipe* using the stream I/O functions, and the task will become inactive and wait when necessary for the data to arrive. The device driver sends a wakeup signal to the task when its requested data has arrived.

A task processing I/O through the stream functions can usually be coded very simply to read the data and process it. The task will be active only when it has something to do without needing to use any other signal mechanism.

System Initialization

All MultiTask! initialization is normally accomplished in the program ***main()*** function. The minimum initialization consists of:

1. Calling ***MTinitialize();*** to initialize MT!'s data structures and variables.
2. Calling ***MTmeminit()*** to initialize the primary heap. ***MTmeminit2(`COLOR0`)*** may be used as well. If ***runtskss()*** is being used and no other functions make use of the heap, then this step may be omitted.
3. Initializing the system clock interrupt with ***usrclk_init();*** or a user-supplied modification of this routine. (Newer ports have this step divided into three separate pieces for finer control).
4. Executing any initialization required by your application (if necessary).
5. Placing at least one task in the run queue by calling ***runtsk();*** (or ***runtskss()***).
6. Calling ***MTstart().***

NOTE: When developing a real embedded application, you can omit the use of ***usrclk_term()***.

The order of the above commands is significant. The ***MTstart()*** function call will start execution of the highest priority task. ***MTstart()*** only returns when multitasking is terminated by some task calling ***MTterminate()***.

Individual stack space is allocated for each task from `COLOR0` memory by the ***runtsk()*** function call. This is why `COLOR0` memory must be initialized before ***runtsk()*** is called. `COLOR0` memory is also used by the system to allocate `MTFILE` structures and serial device buffers when a path is opened with the ***mt_fopen()*** function.

Starting the Code

We suggest you copy the `#include` statements and ***main()*** function from the **coretest.c** program and edit these to suit your application. You should have run the **coretest** program on your target by this point, if this is feasible. The **coretest *main()*** provides all the necessary initialization commands.

Compiling and Linking with the MultiTask! Library

The **makefile** supplied builds the MultiTask! library correctly for the compile and model options you set with the various **make** variables in the file. Any time you change the configuration limits with either the configuration program or by editing **mtcfg.h** or **depends.h** directly, you will need to rebuild the MT! library.

The library naming conventions are:

MultiTask! library: **Mt*n_m*.lib**

where *n* is the value of the *`STCFG`* configuration variable set in the makefile

and *m* is the value of the *`MODEL`* configuration variable set in the makefile (the *`MODEL`* variable does not exist on some platforms, for which a `NULL` value is used instead).

Some compilers may use a library extension other than ***.lib,*** such as ***.a*** (for archive).

Refer to the makefile for examples of building MultiTask! programs. The **coretest** program is an example of a program using all MT! functions except the stream I/O. The **siotest** program is an example of a program including stream I/O. The only difference in linking here is the inclusion of the object file containing the properly configured device table. The device table contains an entry with the name of each device it is possible to open with ***mt_fopen()***.

Configuring MultiTask!

The executable program **stconfig.exe** simplifies setting configuration parameters. This is a simple program to run under DOS in the directory where you have your SuperTask! source files. The source for the configuration program is also provided. If you are running on a platform other than DOS (e.g., UNIX), you may be able to recompile **stconfig.c** to produce an executable to run on your system.

Using the Configuration Program

To run the program, make your development source directory your current directory, and then type:

stconfig

The program reads the **depends.h** and **mtcfg.h** files and others if instructed. These files contain instructions to **stconfig** encoded in comments in the files. The necessary configuration parameters are displayed, along with their current setting in [square braces]. After each such display, press <Return> to keep the value shown, or enter a new numeric value and press <Return>.

As the ***.h** files are processed, an equivalent **include** file for assembly programs is produced, so that there are no longer any parameters that need to be set in two places.

It is also possible to run **stconfig** and have it produce the assembly output file only, without prompting for changes, by using the syntax:

stconfig -a

There are some parameters in the ***.h** files that **stconfig** will not show, as these are rarely modified. If you need to change one of these, you must do it manually, after which you should run **stconfig -a** to regenerate the assembly **include** file. Comments in the configuration

files **depends.h** and **mtcfg.h** will give additional detail on specific parameters.

Using the `-d` option will reset the configuration to the default settings coded in the file. Example:

stconfig -d

When **stconfig** is run, it starts by renaming the file **depends.h** to **depends.bak**. It then prompts for configuration changes and makes a new **depends.h** file containing the new configuration settings. It proceeds to **mtcfg.h,** first renaming it to **mtcfg.bak**, and then constructing a new version of the file. As it does this, it generates an assembler **include** file containing the equivalent configuration definitions for use in the MT! assembly files. The name of the assembler **include** file varies by platform.

CAUTION: If you interrupt the operation of **stconfig** by pressing <Ctrl/Break> or otherwise terminate the program prematurely, it will leave the **depends.h** or **mtcfg.h** file incomplete. In this case you must copy the ***.bak** files over the "**.h**" files to restore them before you compile or run **stconfig** again.

Configuration Parameters

Parameters in mtcfg.h

The user configures MT! for a particular application by setting system parameters in the user configuration file, **mtcfg.h**. The following table summarizes the system parameters that may be configured for specific applications. These parameters define system table sizes, which in turn impose numerical limits on the system services involved. The parameters are described in detail following the table.

Table 2-1: Parameters in **mtcfg.h**

Parameter	Description	Minimum
NUMTSK	Number of tasks	255
NUMEVT	Number of events	max. unsigned
NUMPER	Number of periodic events	< = *NUMEVT*
NUMGEVT	Number of group events	max. unsigned
NUMRES	Number of resources	max. unsigned
NUMMBX	Number of mailboxes	32767
NUMMSG	Total active messages limit	max. unsigned
MBXLIMIT	Maximum messages per mailbox	65535
NUMCOLORS	Number of variable memory heaps	3
NUMPOOLS	Number of fixed memory pools	max. unsigned
NUMSTREAMS	Number of open streams allowed	max. unsigned
MTENVSIZE	Maximum entries in environment	max. unsigned
INC_KLLTSK	If zero, excludes use of ***klltsk()***	0 or 1
INC_PROFILING	If zero, excludes use of profiling	0 or 1

**max. unsigned* means the value of the largest number that can be represented by an unsigned *int* with the compiler in use. Memory layouts and limits may be exceeded by very large values.

NUMTSK specifies the maximum number of tasks that MT! will handle at any given time. Tasks are generally referenced by task table slot number. A task TCB structure is preallocated in RAM for each task. The slot number is a unique number (1..255) that is assigned by the function ***runtsk()***.

NUMEVT specifies the maximum number of user-defined events MT! will handle. Events are referenced by event numbers that range from 0 to *NUMEVT*-1. For example, if *NUMEVT* equals 5, the events would be referenced as 0 to 4. Each event takes one byte of RAM that indicates whether an event is set or clear, plus the size of two TCB pointers for the set and clear queue heads.

If events are not required by your application, the functions ***setevt()***, ***clrevt()***, ***chkevt()***, ***wteset()***, ***wteclr()***, and the table `event_tab` may be deleted from MT! by setting *NUMEVT*= 0. For more information on the functions ***setevt()***, ***clrevt()***, ***chkevt()***, ***wteset()***, and ***wteclr()***, refer to the *MultiTask! Library Reference* chapter.

NUMGEVT specifies the maximum number of user-defined group events handled by the system. Group events are referenced by group event numbers that range from 0 to *NUMGEVT*-1. For example, if *NUMGEVT* equals 5, the group events would be referenced as 0 to 4. Each group event takes one word of RAM, the size of a TCB pointer, and three words per task.

NUMRES specifies the maximum number of user-defined resources you will use. Each resource requires one byte of RAM plus the size of a TCB pointer. Resource numbers are zero-based and range from 0 to *NUMRES*-1.

If resources are not required by your application, the functions ***reqres()***, ***getres()***, ***relres()***, ***chkres()***, and the table space may be deleted from MT! by setting *NUMRES* = 0. For more information on the functions ***reqres()***, ***getres()***, ***relres()***, and ***chkres()***, refer to the *Library MultiTask! Reference* chapter of this manual.

NUMMBX specifies the number of user-defined mailboxes. Each mailbox requires enough RAM for an MBX_DEF structure (6 to 12 bytes). Mailboxes are referenced by mailbox numbers that range from 0 to *NUMMBX*-1.

If mailboxes are not required by your application, the mailbox RAM can be eliminated by setting *NUMMBX* = 0.

NUMMSG specifies the maximum number of active messages that will be handled by the system. This is the maximum number of messages that can be sent that have not yet been received. Internally, a message header is allocated for each active message to link it into the mailbox. This is done automatically by the ***putmsg()*** and ***sndmsg()*** functions. The header is freed automatically when the message is received.

MBXLIMIT specifies the maximum number of messages that can be sent to any one mailbox. *MBXLIMIT* is usually set to *NUMMSG/NUMMBX*, which prevents all the message headers from being consumed by a task sending messages to a mailbox from which they are not being received.

NUMCOLORS specifies the maximum number of variable-size allocation memory pools that can be used by ***reqmem()*** and ***relmem()***. *NUMCOLORS* is normally required to have a value between 1 and 3. If ***runtskss()*** is used, then *NUMCOLORS* may be 0.

NUMPOOLS specifies the maximum number of fixed-size memory pools that you will be using. The pool numbers range from 0 to *NUMPOOLS*-1.

NUMSTREAMS is the number of I/O streams that can be opened at one time. A stream is opened each time ***mt_fopen()*** is called.

MTENVSIZE is the maximum number of environment variables that can be entered in the environment table accessed by ***get_mtenv()*** and ***put_mtenv()***.

INC_KLLTSK normally has a non-zero value (nominally 1). If set to zero, it excludes the internal use of the ***klltsk()*** function. In this case, care must be taken to ensure that ***klltsk()*** and ***MTterminate()*** are never called and tasks do not terminate.

INC_PROFILING normally is defaulted to zero. If set to one, then task profiling will be enabled.

Memory requirements can be displayed by using the **mtbench** test program.

Parameters in depends.h

Table 2-2: Parameters in depends.h

Parameter	Description	Maximum
NUMTCK	Number of clock ticks/time slice	255
CLOCKHZ	Clock interrupt frequency in hertz	max. unsigned
MAX_CMD_CNT	Maximum number of delay queue commands	processor dependent

**max. unsigned* means the value of the largest number that can be represented by an unsigned *int* with the compiler in use.

NUMTCK specifies the number of clock interrupts the system processes before rescheduling tasks. This number depends on the application and the frequency of the clock interrupt. For example, if a clock interrupt occurs every 5 milliseconds and *NUMTCK* is set to 4, then tasks are rescheduled every 20 milliseconds. This means the “time-slice” each task runs will be 20 milliseconds. To deactivate time-slicing, read the description of `STCFG` found in the **makefile**.

CLOCKHZ specifies the number of clock interrupts that occur each second. This number provides the system with the basis for maintaining a clock. In the clock interrupt example above, *CLOCKHZ* would be set to 200 (200 x 5 milliseconds = 1,000 milliseconds = 1 sec.). In most cases, the clock interrupt code provided will automatically reprogram the interrupt rate to match this definition. Please check the clock interrupt code for your particular processor to verify its operation.

MAX_CMD_CNT specifies the number of entries that can reside in the command queue at any given time. If the queue fills up then entries will be overwritten and certain commands will not be executed. If you require a large number (> 255) of queued commands, please contact U S Software, and we will help you determine the maximum safe value for your processor. Most applications will not need such a large number of queued commands.

MT! User-Configurable RAM Usage

To determine the RAM necessary for your application, we recommend you run **mtbench** on your system. This will display the amount of memory required for three applications of differing complexity. It also indicates the amount of RAM size each MT! structure requires, so you are able to compute the amount of RAM needed for 5 tasks (for instance).

IMPORTANT NOTE for 80x86 TARGETS: In order to utilize more than 64K of memory with the memory management functions on an 80x86 (real mode) target, the *HUGE_MEMORY* parameter in **depends.h** (which can be set by `stconfig`) must be set to 1.

Parameters in the makefile

The following variables are normally set in the makefile and passed to the compiler and assembler as command line options to define the variable.

STCFG is a variable used to pass multiple configuration parameters to the compiler and assembler. Each bit is mapped to another variable as follows:

`Bit 0 = TSL`	Time-slicing enabled
`Bit 1 = INC_LOCAL_MEM`	Local heap tracking
`Bit 2 = INC_PROFILING`	Include profiling

Example: If *STCFG* = 5, then *TSL* =1, *INC_LOCAL_MEM* = 0, and *INC_PROFILING* = 1.

TSL is the Time-SLicing compilation flag. This flag is normally set in the makefile, and its value is passed through *CFLAGS* and *AFLAGS* to all C and assembly modules. When *TSL* = 1, round-robin time-slicing among tasks of equal priority is enabled. This implements the time-slicing behavior described in the manual. If *TSL* = 0, then no round-robin time-slicing will occur. In this case, if two (or more) tasks of equal priority are in the run queue, the first one to run will run to completion or until it is preempted by a higher-priority task. After the value of TSL is changed, **mtcore.c** and **mtsched.*** must be recompiled for the new setting to take effect. This is a compile-time option and cannot be dynamically switched on and off. When *TSL* = 0, task switching time will be reduced.

INC_LOCAL_MEM
is a compile-time option that specifies the behavior of the local memory type. If compilation is with *INC_LOCAL_MEM* set to zero, then the local memory attribute is ignored, and task requested local memory is not automatically released when the task dies. If compilation is with the *INC_LOCAL_MEM* non-zero, then local memory behaves as described in the manual.

TRG_ID should be set to match the evaluation board that you are using. Please see your particular **makefile** for details.

Parameters in usrasign.h

When defining ID numbers for MultiTask! facilities, you should use the file **usrasign.h**. By keeping ID numbers in one location, you will more easily be able to avoid errors from duplicate ID numbers. In addition, when MT! is integrated with other U S Software products (e.g. USFiles), certain ID numbers may already be defined in **usrasign.h**.

Parameters in userio.h

If you are using stream I/O or USFiles in your application, then the device table in userio.h needs to be configured. The C source file **dev_tab.c** `#includes` **userio.h,** and it is here that the actual device table is generated. Our makefiles are constructed to generate different versions of the object file containing the device table. Each version contains a different combination of devices needed to run a specific test program. Since different programs require different devices in the device table, we do not include the device table in the SuperTask! libraries. If we did, code would be linked from the libraries for *file managers* and *drivers* for the unused devices, which would be undesirable.

We derive the name of the object file containing a specific configuration of the device table from the code letters shown in the preceding table. For example, the object file **dev_ps.obj** contains the PCIO and SIO (disk drives and serial devices). The object file **dev_s.obj** contains only the serial port devices. The .**obj** extension varies by platform; it may be .**o** or some other extension on the platform you have selected.

The name of the object file containing the correct configuration of the device table (e.g., **dev_s.obj**) is given explicitly during the link operation.

Using mtdbg() for Debugging

The module **mtdbg**.**c** (along with **iprintf**.**c** and the module containing functions ***getchr()*** and ***putchr()***) can be linked to any MT! application code to provide a debugging display.

By altering the ***getchr()*** and ***putchr()*** simple character I/O routines to communicate with your target processor as appropriate (usually through a serial port), you can use this module with any target. Typically a dumb terminal would be connected to this target port to provide the display and user control.

The ***mtdbg()*** function code in the target needs to be executed in some manner to activate the debug display. There are several possible ways of doing this:

- From some source debuggers (e.g., Borland's Turbo Debug) you can force a function to be called on the target system for purposes of evaluating the return value from that function. In Borland's TD, this is done with the following menu sequence: DATA, EVALUATE, ***mtdbg(),*** EVAL. Some simple ROM monitors allow a call to be issued.
- Insert calls to ***mtdbg()*** in your code where needed, as you would imbed ***printf()*** statements for debugging purposes.
- Set up a high-priority task to call ***mtdbg()*** when some operator keystroke is entered, or periodically, etc.
- Set up some interrupt source, such as the abort button on the target, to activate task as in the item above.

When the ***mtdbg()*** function is entered, it displays a table showing all tasks that have been started in the system, showing what queue they are in and their status (running, waiting , etc.). ***Mtdbg()*** will then display the prompt `MTDBG`: and wait for you to enter a command.

Typing **`help`** will show other commands available. All pertinent MultiTask! system information can be displayed. Pressing <Return> or the ***Q*** command will return to the caller (i.e., debugger, etc.). Interrupts are masked while the ***mtdbg()*** function is active, so nothing else will run until you exit back to whatever called ***mtdbg***.

The ***HELP*** command displays the following command summary:

MTDBG Commands:

<Return>	Return to caller (debugger)
<n>	Run for <n> steps, then break
BR{E|G|R} *[-] <num>*	Break on *event|group|resource* change
BRM *[-] <num>*	Break when message arrives at mailbox
BRT *[-] <num>*	Break when task runs
CE *<evtnum> <value>*	Change event to value
CONFIG	Display configuration data
DS	Redisplay status screen
D{E|G|M|R|T} *<num>*	Display *event|group|mailbox|resource|task*
IGT *[-] <slot>*	Ignore step display for Task
MEM|LMEM|GMEM	Display *free|local|global* memory
PMEM [*num*]	Display buffer pool <*num*>
NAMES	Display task names table
NS	No step; run until breakpoint
T	Display time queue data
TK *<slot>*	Kill task at slot
TP *<slot> <priority>*	Change task priority
TR *<pri> <name> <stk>*	Run task
Q	Return to calling program

The ***NAMES*** command displays a list of task names in the program. In order for this command to work, the program must contain a table of the task names, which is an array of structures of type `TASK_NAMES`. This array must be named `dbg_tname`. See the **coretest.c** program for an example of this array. When you add a new task to your program, you must also add a name entry to this array; otherwise, the task name will be unknown to **mtdbg** and the status display will show the task name as "???".

The commands ***DE, DG, DM***, and ***DR*** display the event, group event, mailbox, or resource requested. The command is followed by the item number, and zero is assumed if no number is given.

Example

```
MTDBG:DE 1
Event[ 1 ] = 0
MTDBG:
```

The ***DT*** command displays detailed task information including current stack usage. This command can be followed by either the task slot number or the task name to indicate which task to display.

NOTE: The stack usage cannot be determined for the currently running task. For all other tasks, the display will show the current stack usage along with resources owned and local memory blocks owned.

There are several commands to display how MT!-managed memory is currently allocated. ***MEM*** displays a list of the currently free blocks for each color of memory managed by the ***reqmem()/relmem()*** functions. The blocks sizes are shown in decimal, and addresses are shown in hexadecimal. ***LMEM*** shows a list of currently allocated `LOCAL` memory blocks along with what task they are assigned to. ***GMEM*** shows the currently allocated `GLOBAL` memory blocks.

PMEM followed by the pool number will display the status of the selected buffer pool (memory managed by ***reqbuf()/relbuf(),*** etc.).

The ***T*** command displays the current contents of the time queue. The time queue contains entries for periodic events, delayed tasks, and tasks with a timeout active. The number of clock ticks remaining is shown along with other pertinent information.

The displayed output of **mtdbg.c** is made through a call to ***putchr()*** in a single location in each of these files. You may modify where the output is sent by changing the call in these locations. For instance, if you are using the new console/keyboard driver (`CON` or `VIEW` device) in the PC environment, you may want to open a VIEW window for the debug output and send the display there by changing the ***putchr()*** call to a ***mt_fputc()*** call. You may similarly want to change the ***getchr()*** function to redirect input.

3. MultiTask! Library Reference

Chapter Contents

Functions by Category

NOTE: Functions marked with a * can be used from an ISR, generally through the ***MTqcmd_c()*** function. The exception is the ***ireqbuf()*** function, which can be called directly.

System Control Functions

MTinitialize	Initializes system.
Mtstart	Starts multitasking.
MTterminate	Terminates all tasks and shuts down.

Task Control Functions

runtsk*	Initializes task and places in run queue.
klltsk*	Terminates task, deallocates resources.
pritsk*	Changes task priority.
dlytsk*	Delays task a specified time. Units can be ticks, seconds, minutes, hours, or forever.
delay_until	Delays a task until specific system time. A time up to 2^{31} ticks into the future may be used.
scdtsk	Reschedules (gives up time-slice).
suspend	Suspends task (prevents from running).
reanimate	Cancels task suspension.
waktsk*	Wakes task from time delay queue
wketsk	Wakes task from any queue (obsolete).
wketsk_nto	Wakes task without timeout (obsolete).

Event Functions and Variables

setevt*	Sets event to one.
clrevt*	Sets event to zero.
incevt*	Adds one to event.
decevt	Subtracts one from event.
chkevt	Gets current event setting.
wteset	Waits until event is non-zero.
wteclr	Waits until event is zero.
wteset_dec	Waits until event non-zero and subtracts 1.
period	Starts/stops periodic event increment.
oneshot	Increments event at specified time.
TIME_KEEPER_EVENT	ID number for event used for time keeping (see **usrasign.h**).

Group Event Functions

setgrp*	Sets bits in group event.
clrgrp*	Clears bits in group event.
waitgrp	Waits for specific group event setting.
chkgrp	Gets current group event setting.
GrpWakeValue	Reports wake up condition.

Memory (Heap) Functions and Variables

MTmeminit	Initializes variable memory *COLOR0*.
MTmeminit2	Initializes variable memory, any color.
reqmem	Requests variable size memory allocation.
relmem*	Releases memory obtained by reqmem.
chkmem	Checks memory integrity.
`COLOR0`	Primary heap
`COLOR1`	Secondary heap
`COLOR2`	Last allowed heap
`LOCAL`	Indicates local memory allocation
`GLOBAL`	Indicates global memory allocation

Memory (Buffer Pool) Functions and Variables

init_mem_pool	Initializes fixed-buffer pool.
del_pool	Terminates fixed-buffer pool.
getbuf	Waits for and gets fixed buffer.
reqbuf	Task requests fixed buffer.
ireqbuf*	ISR requests fixed buffer.
relbuf*	Releases fixed buffer.
chkbuf	Checks fixed-buffer availability.
`TASK_POOL`	Specifies that a memory pool is accessed by tasks.
`ISR_POOL`	Indicates that a memory pool is accessed by ISRs.

Message Functions and Variables

sndmsg*	Sends a message.
putmsg	Sends a message; waits if mailbox full.
rcvmsg	Receives message or packet with possible timeout.
chkmsg	Checks if message or packet is in mailbox (obsolete).
chkmbx	Determines number of messages in mailbox.
flushmbx	Discards all messages and packets in mailbox.
sndpkt	Sends a packet.
putpkt	Sends a packet; waits if mailbox full.
relpkt	Releases packet memory.
`MSGSUSPEND`	When `OR`ed with priority, suspends the task sending the message after the message is sent.
`SUPERPRI`	When `OR`ed with priority, forces a message to the head of queue.

Resource Functions and Variables

getres	Waits for and acquires a resource.
reqres	Acquires a resource if available.
relres*	Releases a resource.
chkres	Finds resource owner.
freeres	Unconditionally releases a resource.
`PCFM_RESOURCE`	Resource ID used to protect USFiles functions (see **usrasign.h**).

Interrupt Functions

Mtsched	System scheduler entry from ISR (ASM).
MTsched_c	System scheduler entry from ISR (C).
MTqcmd_c	System function access from ISR.

Timer Functions and Variables

dlytsk*	Delays a task for a specified time. Units can be ticks, seconds, minutes, hours, or forever.
waktsk*	Wakes a task. If task is not delayed yet, will cancel next attempt to delay task.
delay_until	Delays a task until a specific system time. A time up to 2^{31} ticks into the future may be used.
get_sys_time	Gets system timer tick count.
oneshot	Increments event at specified time.
period	Starts/stops periodic event increment.
`DLY_TICKS`	Specifies units for delay are ticks.
`DLY_SECS`	Specifies units for delay are seconds.
`DLY_MINS`	Specifies units for delay are minutes.
`DLY_HOURS`	Specifies units for delay are hours.
`CLOCKHZ`	Specifies the clock frequency.

Miscellaneous Functions

clr_profile	Clears task profile counts.
get_profile	Returns task profile counts.
get_tcb	Gets copy of task TCB.
getclk	Gets time kept by timekeeper task.
setclk	Sets time kept by timekeeper task.
put_mtenv	Sets environment variable.
get_mtenv	Gets environment variable.

Critical Code Protection Functions and Variables

MASK_INTS()	Masks interrupts (not nestable).
UNMASK_INTS()	Enables interrupts.
`++mt_busy`	Locks the kernel (nestable).
MTqproc()	Unlocks the kernel.
block_preemption()	Prevents task switching (nestable).
unblock_preemption()	Enables task switching.

Status Reporting Variables

`TASK_ID cur_task`	Variable with the current task ID number.
`errno`	Variable containing error code for current task (defined in **rtos1.h**).
`STACK_FILL`	If defined in **depends.h**, will report amount of free stack space when ***MTterminate()*** is called.
`cmdqerrors`	Errors from queued system calls (defined in **mtdata.h**).

Stream I/O Functions

Please see the *Stream I/O Library Reference* chapter for a discussion of Stream I/O functions.

Hooks Available for Error Recovery

CMDQUE_FULL_CHECK
Used in ***MTqcmd_c()***.

MTQPROC_TRAP Executed if illegal value of `mt_busy` encountered in ***MTqproc()*** in **MTinit.c**.

MTSTACK_TRAP Executed if stack overflow detected in ***MTqproc()*** in **MTinit.c**.

See also: The files **depends.h** and **MTinit.c.**

New Low-Level Functions

As we update MT!, we have been reworking our implementations of console I/O, interrupt control, and ticker control. The following functions replace the older functions ***usrclk_init()***, ***usrclk_term()***, ***putchr()***, and ***getchr()***. Not all ports will have these. Please examine the **usrclk** and **getput** files that are delivered for your CPU.

Console I/O Functions

ussDebugInit	Initializes debug device.
ussDebugTerm	Terminates use of debug device.
ussDebugGetTst	Reports if a character is available.
ussDebugGetChr	Behaves like ***getchar()***.
ussDebugPutChr	Behaves like ***putchar()***.
ussDebugPutStr	Outputs a string without newline (\n) character appended.
ussDebugPutBlk	Behaves like ***fwrite()***.

Interrupt Control Functions

ussIntrptInit	Initializes interrupts.
ussIntrptTerm	Ends interrupt availability.
ussIntrptPut	Installs a high-level interrupt handler routine.
ussIntrptSet	Installs a low-level interrupt service routine. This function is only available for hardware vector tables.

Ticker Control Functions

ussTickInit	Initializes the ticker.
ussTickTerm	Ends ticker operation.
ussTicks	System ‘up time’ in ticks.
ussTimeMS	System ‘up time’ in milliseconds.
uss_mSecToTicks	Converts milliseconds to ticks.
ussTick10s	Tick rate (in ticks per 10 seconds).

Include Files

Programs using any of the SuperTask! library functions should include the header file **rtoshdrs.h**, or copy the include statements from this header into your program. The file **rtoshdrs.h** does not exist on the distribution disks. When using MT! **rtoshdrs.h** is created by copying **rtos1.h**.

The **rtoshdrs.h** file will provide all necessary function prototypes and constant definitions as referenced in the library. Several of the functions listed above are defined as macros. These macros can typically be found in the **mtlib.h** or **depends.h** files.

The compilation "model" or other options that the library is compiled with must be compatible with the way you compile your code. On the 80x86, for instance, if your program is compiled in the `LARGE` model, then the library must be compiled the same.

It will be useful to keep a printed copy of the header files on hand as a reference.

Typedef Names

Atomic Typedef Names

The final five names are specified in the draft for ANSI C-99.

Name	Description
`byte`	unsigned `char` (8 bits)
`int16`	signed 16-bit integer
`int32`	signed 32-bit integer
`uint`	unsigned integer of same size as "`int`"
`uint16`	unsigned 16-bit integer
`uint32`	unsigned 32-bit integer

Derived Typedef Names

Name	Equivalent	Description
TASK_ID	byte	Task slot number (ID)
tick_cnt_t	uint32	System time tick count for delays, etc.
profile_t	uint32	Task profile tick count

System Structure Typedef Names

Name	Used for
ALLOCMEM_DEF	Allocated memory block header
CMDARG	Entry in command queue
ENV_DEF	MT environment entry
MBX_DEF	Mailbox structure
MEM_DEF	Free memory block header
MEM_INFO	Information returned by ***chkmem()***
MEM_POOL	Buffer memory control structure
MEMHEAD_DEF	Color pool root structure
MSG_DEF	Message structure linked into mailbox
MTtime_t	Used by timekeeper task
PKT_HDR	Message packet header
QARGS	Command arguments union in ***cmdque***
TASK_DEF	TCB (Task Control Block structure)
TASK_NAMES	Task name table used by ***mtdbg*** and ***protodbg***
TIME_DEF	Time queue entry

Function Descriptions

See also: *Error Codes* appendix for a complete list of return values.

acquire

Sets the first zero bit in designated table and returns bit number.


```
int acquire(byte *table, int limit);
```

table	pointer to the user-initialized table
limit	number of significant bits in the table

The ***acquire*** function searches the first *limit* bits of memory beginning at *table* and sets the first zero-bit found, returning its bit number. The table should be initialized (set to all zeros) before it is used. The table must be at least (*limit*+7)/8 bytes long. The bit-number of the bit set by ***acquire()*** will be between 0 and *limit*-1. If there were no more zero bits in the table, ***acquire*** will return an error value.

See also: ***release***

Return Value

{0..limit-1}	bit acquired
E_TABFULL	table full (all bits 1)

Example

```
byte mbx_aq_table[(NUMMBX+7)/8];
int i;

for(i=0; i<NUMMBX; i++)
  release(mbx_aq_table, i);     /* init table */

        /* assign first available number */
i = acquire(mbx_aq_table, NUMMBX);
```

block_preemption

Prevents task switches from occurring.

```
void block_preemption(void);
```

The ***block_preemption()*** function allows protection of critical code sections. Using ***block_preemption()*** still allows interrupts to occur, but no task switch will be performed until ***unblock_preemption()*** is called. This function is defined as a macro in the **mtlib.h** file, and calls to it can be nested.

See also: ***unblock_preemption, MASK_INTS, UNMASK_INTS***

Return Value

None

Example

```
/* Entering critical code section */
block_preemption();       /* Prevent task switches */
/* Execute critical code */
unblock_preemption();     /* Allow task switching */
```

chkbuf

Checks number of available buffers.

```
int chkbuf(uint poolid);
```

poolid ID number of pool {0..*NUMPOOLS*-1}

Returns the number of memory buffers currently available in the memory pool specified by *poolid*.

See also: ***del_pool, getbuf, init_mem_pool, relbuf, reqbuf***

Return Value

count	number of buffers available {0..*n*}
E_INVPID	invalid *poolid* specified

Example

```
#define POOL0 0           /* PoolID number */

if( chkbuf(POOL0) <= 0 )
  { /* no memory available or E_INVPID */ };
```

chkevt

Gets state of an event.

```
int chkevt(uint event);
```

event user-assigned event number (0..*NUMEVT*-1)

The system will return the status of *event*. If an invalid *event* is specified, an error indication will be returned.

See also: ***clrevt, decevt, incevt, period, setevt, wteclr, wteset, wtesetdec***

Return Value

0	event clear
1..255	event set
E_INVEVT	invalid event

Example

```
#define data_avl 50        /* assign event # */
int status;                /* return status */

status = chkevt(data_avl); /* check data avail */

if (status > 0)
  {
  process_data();          /* process data */
  clrevt(data_avl);        /* no data avail */
  }

else if (status == 0)
  {}    /* no data to process */

else
  {}    /* invalid event num */
```

chkgrp

Gets state of group event.

```
int chkgrp(uint evtgrp, uint *result);
```

evtgrp user-assigned event number. (0..*NUMGEVT*)

result address of variable to hold group event value

Copies the current bit pattern of the group event specified by *evtgrp* into the variable pointed to by **result*. If an invalid *evtgrp* is specified, an error indication is returned.

See also: ***clrgrp, setgrp, waitgrp, GrpWakeValue***

Return Value

SUCCESS OK

E_INVGRP invalid *evtgrp*

Example

```
#define data_avl 50
int status;
        /* event no. */

uint result;

status = chkgrp(data_avl,&result);
        /*get event bit pattern*/
```

chkmbx

Returns count of messages in mailbox.

```
int chkmbx(uint mailbox);
```

mailbox user-assigned mailbox number (0..*NUMMBX*-1)

The number of messages currently in *mailbox* is returned. If an invalid *mailbox* is specified, a negative error value is returned.

See also: ***flushmbx, putmsg, putpkt, rcvmsg, relpkt, sndmsg, sndpkt***

Return Value

0..+*n* number of messages waiting in *mailbox*

E_INVMBX invalid *mailbox* specified

Example

```
void *msgptr;                   /* message address */
uint tskmbx;                    /* task mailbox */
uint status;                    /* return status */
tskslt = cur_task;              /* get our slot */
tskmbx = tskslt;                /* use slot as mbx */

while (chkmbx(tskmbx) == 0) scdtsk();
        /* poll for a msg */

msgptr = rcvmsg(tskmbx,0);      /* rcv msg */
```

chkmem

Checks memory system integrity.

```
int chkmem(uint color, MEM_INFO *info);
```

color memory color of interest

info pointer to structure returned information

An integrity check of all memory blocks of the specified *color* is performed. Information about free memory of the specified *color* is returned by filling in the fields of the MEM_INFO structure at **info*. The MEM_INFO structure is defined in **mtlib.h** as:

```
typedef struct mem_info { /* information returned by
                             chkmem */
  uint fragments;         /* number of fragments */
  mem_size_t total_free;  /* total free bytes */
  mem_size_t largest;     /* size of largest fragment in
                             bytes */
  uint allocated_blocks;  /* number of allocated
                             blocks */
  uint sequence_no;       /* if memory is corrupt,
                             sequence_no will indicate at
                             what point corruption was
                             detected by chkmem */
} MEM_INFO;
```

See also: ***Mtmeminit, Mtmeminit2, relmem, reqmem***

Return Value

`SUCCESS`	information returned in *`info`* struct
`E_INVCOLOR`	invalid *`color`* specified
`E_CORRUPT`	memory structure is corrupted

Example

```
MEM_INFO meminfo;    /* memory info structure */

if( chkmem(COLOR0, &meminfo) )
   /* error */
```

chkmsg (obsolete)

Checks for message in mailbox.

```
void *chkmsg(uint mailbox);
```

mailbox user-assigned mailbox number (0..*NUMMBX*-1)

The system will return the message pointer of the first message in *mailbox*. If no message is present or an invalid *mailbox* is specified, a NULL pointer is returned. Even though the message pointer is returned, the message is not actually removed from the mailbox. This should be done with ***rcvmsg().***

See also: ***chkmbx, flushmbx, rcvmsg, sndmsg***

NOTE: This function should be considered obsolete. The ***chkmbx()*** function should be used instead

Return Value

NULL invalid *mailbox* or no messages available

!NULL address of first message

Example

```
void *msgptr;              /* message address */
uint tskmbx;               /* task mailbox */
uint status;               /* return status */
tskslt = cur_task;         /* get our slot */
tskmbx = tskslt;           /* use slot as mbx */

while (chkmsg(tskmbx) == NULL) scdtsk();
        /* poll for a msg */
```

chkres

Checks status of a resource.

```
int chkres(uint resrc);
```

resrc user-assigned resource number (0..*NUMRES*-1)

The system will return the status of *resrc*. If an invalid resource is specified, an error indication will be returned. Zero is returned if the resource is available. If the resource is owned, the TASK_ID of the owning task is returned (which will be a positive number).

See also: ***getres, relres, reqres***

Return Value

0	resource available
1..NUMTSK	resource owner's task slot
E_INVRES	invalid *resrc*

Example

```
int owned = 0;            /* own count */
int resrc = 0;            /* resource number */

while (resrc != NUMRES)
  {
  if (chkres(resrc) ==cur_task) /* check owner */
        {
        owned += 1;       /* count resources we own */
        resrc += 1;       /* next resource */
        }
  }
```

clr_profile

Clears task profile information.

```
int clr_profile(TASK_ID slot, int prof_type);
```

slot	slot number of task entry to clear, "don't care" when *prof_type* = PROF_ALL (1..NUMTSK, 0 = current task)	
prof_type	PROF_TASK	clear only entry of slot
	PROF_SYSTEM	clear slot and system total
	PROF_ALL	clear all slots and system

Clears some or all of the profiling counts. One count is maintained for each task defined by NUMTSK, plus one for the idle state when no task is running (accessed as slot 0). One count is also maintained for the system total, which is the sum of all tasks including slot 0. The value passed for *prof_type* determines whether only a single task count is cleared, a single task and the system total, or if all task counts including the system total will be cleared.

See also: ***get_profile***

Return Value

SUCCESS	successful operation
E_INVPRF	invalid *slot* or *prof_type*

Example

```
/* reset profile information for system total and current task */
if(clr_profile(cur_task, PROF_SYSTEM) != SUCCESS)
      {}    /* error handling */

/* reset profile for current task and not total */
if(clr_profile(cur_task, PROF_TASK) != SUCCESS)
      {}    /* error handling */

/* reset all slot profiles and system total */
if(clr_profile(0, PROF_ALL) != SUCCESS)
      {}    /* error handling */
```

clrevt

Clears an event.

```
int clrevt(uint event);
```

event user-assigned event number (0..*NUMEVT*-1)

The system will clear *event* by setting its value to zero. If an invalid event is specified, an error indication will be returned. Any task waiting for the event to be cleared will be moved to the run queue where it will preempt the running task if it has a higher priority than that task.

See also: ***chkevt, decevt, incevt, setevt, wteclr, wteset, wtesetdec***

Return Value

SUCCESS	*event* cleared
E_INVEVT	invalid *event*

Example

```
#define data_avl 50             /* assign event # */
int status;                     /* return status */

process_data();                 /* process data */
status = clrevt(data_avl);
        /* no data avail */

if (status != SUCCESS)          /* check errors */
  {}    /* invalid event num */
```

clrgrp

Clears bits in group event.

```
int clrgrp(uint evtgrp, uint clr_mask);
```

evtgrp	user-assigned group event number (0..NUMGEVT-1)
clr_mask	16-bit mask of bits to clear

The bits that are set in *clr_mask* are cleared in the group event specified by *evtgrp*. If an invalid value is passed for *evtgrp*, an error indication is returned. Any tasks that are waiting for the resulting group event bit pattern will be moved to the run queue where they will preempt the running task if they have a higher priority than that task.

See also: ***chkgrp, setgrp, waitgrp***

Return Value

SUCCESS	OK
E_INVGRP	invalid *evtgrp*

Example

```
#define data_avl 50
int status;
        /* event # */

process_data();           /* process data */

status = clrgrp(data_avl, 0xF120)
        /* clear bits */

if(status != SUCCESS)
  {}    /* invalid event no.  */
```

decevt

Decrements an event.

```
int decevt(uint event);
```

event user-assigned event number (0..*NUMEVT*-1)

The system will subtract one from the value of *event*. If an invalid *event* is specified, an error indication will be returned. Any task waiting for the event to be cleared will be moved to the run queue if ***decevt*** caused the event value to become zero. When a task is moved to the run queue, it will preempt the running task if it has a higher priority than that task. You should note that if you decrement an event whose value is zero, it will have a new value of 255, at which point the event is considered set.

3

See also: ***chkevt, clrevt, incevt, setevt, wteclr, wteset, wtesetdec***

Return Value

SUCCESS *event* decremented

E_INVEVT invalid *event*

Example

```
#define data_avl 50             /* assign event # */
int status;                     /* return status */

while(TRUE){
  status=wteset(data_avl,1000); /* wait for data */
  if (status != SUCCESS)        /* timed out ? */
     system_reset();            /* yes, must abort */
  process_data();               /* process it */
  (void)decevt(data_avl);       /* signal finished */
}                               /* wait for more */
        /* set data available */
```

del_pool

Deletes a memory pool.

```
int del_pool(uint poolid);
```

poolid ID number of pool {0..*NUMPOOLS*-1}

De-initializes the memory pool specified by *poolid*. Any later request for memory from the pool will return an error.

See also: ***init_mem_pool, reqbuf, getbuf, relbuf, ireqbuf_c***

Return Value

SUCCESS	pool successfully deleted
E_INVPID	invalid *poolid* specified

Example

```
#define POOL0 0           /* pool ID number */

(void)del_pool(POOL0);    /* assume valid poolid */
```

delay_until

Delays a task until a specified time.

```
int delay_until(TASK_ID slot,tick_cnt_t time);
```

slot	slot number of task to delay (1..NUMTSK, 0 = current task)
time	system time of wake up

The system will delay the task in *slot* if it is in the run queue, otherwise an error indication will be returned. The delay will be until the system tick count equals *time*. The value of time should be the current *time* plus 1 to 2^{31}. Times that wrap around the largest tick value are handled properly.

See also: ***get_sys_time, dlytsk, oneshot, period***

Return Value

SUCCESS	OK
E_INVSLT	invalid task ID
E_LATE	time is already less than or equal to current system time

Example

```
TASK_ID slot;              /* task slot */
int status;                /* return status */
tick_cnt_t now;

now = get_sys_time();      /* get reference time */

{ /* do some process here that takes an unknown amount
     of time, but less than 40 ticks */ }

status = delay_until(slot, now+40);
         /* delay 40 ticks from reference */

if (status != SUCCESS)     /* check errors */
  {}                       /* invalid parameter */
```

dlytsk

Delays a task.

```
int dlytsk(TASK_ID slot, byte dlytyp, uint dlytme);
```

slot	slot number of task to delay (1..NUMTSK, 0 = current task)
dlytyp	delay type (*DLY_TICKS*, *DLY_SECS*, *DLY_MINS*, or *DLY_HOURS*)
dlytme	delay count (0 = infinite delay, 1..maxvalue(uint) = timed delay)

The system will delay the task in *slot* if it is in the run queue, otherwise an error indication will be returned. The delay (based on *dlytyp*) will be in clock ticks, seconds, minutes, or hours, which are respectively delay types *DLY_TICKS*, *DLY_SECS*, *DLY_MINS* and *DLY_HOURS*. The delay will be for *dlytme* time periods (a number ranging from 1 to the maximum value of type uint). A *dlytme* of 0 will give the maximum possible delay. All delay types are converted internally into clock ticks, which for most processors are stored as an unsigned long. This means the maximum possible delay is 2^{32} clock ticks. If the clock interrupts every 5ms, then this translates to a maximum delay of approximately 248 days.

The accuracy of the delay for all delay types is from -1 to +0 clock ticks; e.g., if a delay of 2 *DLY_TICKS* is specified, the actual delay will be somewhere between 1 and 2 ticks.

Calling ***waktsk()*** will wake up a task that has been delayed. In addition, if the task specified in a call to ***waktsk()*** is not presently delayed, then the next call to ***dlytsk()*** for that task will be canceled.

See also: ***delay_until, period, waktsk***

Return Value

SUCCESS	OK
E_INVDLY	invalid parameter
E_INVSLT	invalid task slot

Example

```
TASK_ID slot;             /* task slot */
int status;               /* return status */

status = dlytsk(slot,DLY_MINS,5);
        /* delay for 5 minutes */

if (status != SUCCESS)    /* check errors */
  {}    /* invalid parameter */
```

find_pipe

This function can be found in the *Stream I/O Library Reference* chapter.

flushmbx

Discards all messages in a mailbox.

```
int flushmbx(uint mailbox);
```

mailbox mailbox number

Any messages (***sndmsg, putmsg***) or packets (***sndpkt, putpkt***) waiting in mailbox are discarded. Any tasks waiting for a message or packet from the mailbox and any tasks waiting for the mailbox to be not full are awakened.

Return Value

SUCCESS no errors

E_INVMBX invalid mailbox number

Example

```
flushmbx(1);
flushmbx(2);
```

freeres

Frees a resource.

```
int freeres(uint resrc);
```

resrc user-assigned resource number (0..NUMRES-1)

The system will set the resource assignment count to zero and release resource *resrc* to the highest priority task waiting for the resource. The resource will be released even if the task releasing the resource does not own it. This allows ***klltsk()*** to release a task's resources when it is killed. If an invalid resource is specified, an error indication will be returned.

NOTE: This routine is not for normal use, but may be used for error recovery.

See also: ***getres, reqres, relres***

Return Value

SUCCESS resource released

E_INVRES invalid resource number

Example

```
#define printer 6

if (!getres(printer,0)){
  /* use printer */
  freeres(printer);            /* free it */

}else
  { /* error handling */ }
```

NOTE: All resources owned by a task are freed automatically by ***klltsk()*** when a task is killed, whether explicitly or implicitly by coming to the function end.

get_mtenv

Returns an environment variable.

```
void *get_mtenv (char *member);
```

member pointer to the member name string to be searched for

The pointer to the value for environment variable *member* is returned. If *member* is not found in the environment list, a NULL pointer is returned. The string *member* must match exactly an entry in the environment; i.e., uppercase and lowercase characters are not treated as equal.

See also: ***set_mtenv***

Return Value

NULL	no match for *member* found in environment
non-NULL	pointer value stored as argument for *member*

Example

```
char *cp;
int *ip;

  cp = get_mtenv("Motor_task_status");
  if( !cp )
     {  /* some error */ }
  ip = get_mtenv("Motor_task_ID");
  if( !ip )
     /* error handling */
```

get_profile

Gets profiling information.

```
profile_t get_profile(int slot, prof_type *profile);
```

slot	slot number of task profile to copy or PROF_ALL for all, or PROF_SYSTEM for system total only
profile	address of array to copy data to slot = 0..NUMTSK : array size = 1 slot = PROF_SYSTEM : array size = 0 slot = PROF_ALL : array size = NUMTSK+1

Returns the total system clock tick count since system startup or the last time the count was reset by ***clr_profile().*** If the value of *slot* is PROF_SYSTEM, nothing is transferred to **profile*. If the value of *slot* is PROF_ALL, the entire profile array (NUMTSK+1 elements) is copied to **profile*; otherwise only the single entry for the task at *slot* is copied.

See also: ***clr_profile***

Return Value

E_INVSLT	invalid slot number
0..*n*	system total clock ticks

Example

```
profile_t total, task_data[NUMTSK+1], task1_prof;
total = get_profile(PROF_ALL, &task_data[0]);
        /*get all data */

if(total == (profile_t)-1)
  {}    /* invalid slot error */

total = get_profile(PROF_SYSTEM, NULL);
        /* get system total only */

  /* get profile of slot1 task & system total: */
total = get_profile(1, &task1_prof);    y
```

get_sys_time

Returns the current system time.

```
tick_cnt_t get_sys_time(void);
```

The current system time is returned. The system time is typically maintained as a long (32-bit) integer count that is incremented at each clock interrupt.

See also: ***dlytsk, delay_until, oneshot***

Return Value

`0..0xffffffff` current tick time

Example

```
tick_cnt_t now;

now = get_sys_time();      /* get reference time */
```

get_tcb

Gets a copy of Task Control Block.

```
int get_tcb(TASK_ID slot, TASK_DEF *task_status);
```

slot	slot number of task (1..NUMTSK, 0 = current task)
task_status	pointer to structure of type TASK_DEF to be filled with a copy of the TCB for the task at the slot requested.

Copies the contents of the task TCB structure for the task in *slot* to the structure pointed to by *task_status*. The structure definition for TASK_DEF can be found in the include file **mtlib.h**. This structure contains all status information about any task known to MT!, including what queue it currently resides in and its priority. The queue will indicate if a task is running or waiting for an event, resource, or other stimulus.

Return Value

SUCCESS	OK (task's TCB is copied to **task_status*)
E_INVSLT	invalid slot number

Example

```
TASK_DEF guard_status;

if( get_tcb( slttsk(guard_task), &guard_status) )
  error_exit();
```

getbuf

Gets a memory buffer.

```
void * getbuf(uint poolid, uint timeout);
```

poolid	ID number of pool {0..*NUMPOOLS*-1}
timeout	timeout period in clock ticks {0 = none}

Returns a pointer to a memory block from the memory pool specified by *poolid*. If the specified pool does not have a buffer available, the calling task will wait until one becomes available or until the specified *timeout* has elapsed. This call should only be used by "task" code and should never be made from an ISR.

See also: ***reqbuf, relbuf, init_mem_pool, del_pool***

Return Value

pointer	pointer to memory buffer
NULL	no buffer available, or E_INVPID | E_INVPT | E_TIMED_OUT

Example

```
#define POOL0 0           /* PoolID number */
char *buffer;             /* pointer to buffer */

buffer = (char *)getbuf( POOL0, 50 );
if( buffer == NULL )
   { /* error handling */ };
```

getclk

Gets the time of day maintained by the ***time_keeper()*** task.

```
void getclk(MTtime_t *time);
```

time pointer to structure to receive the time

The time of day maintained by the ***time_keeper()*** task is copied into the structure pointed to by *time*.

NOTE: The ***time_keeper()*** task must have been started for the time of day clock to keep time.

See also: ***setclk***

Return Value

`time->hour`	set to system hour	0..23
`time->minute`	set to system minute	0..59
`time->second`	set to system second	0..59

Example

```
MTtime_t clock_time;            /* time structure */
getclk(&clock_time);            /* get time */
```

getres

Waits for and gets a resource.

```
int getres(uint resrc, uint timeout);
```

resrc	user-assigned resource number {0..NUMRES-1}
timeout	timeout delay in ticks, 0 = infinite

The system will check the status of resource *resrc*. If the resource is available, it will be assigned to the calling task. If the resource is unavailable, the calling task will be placed in a resource wait queue until the resource becomes available to the task. If an invalid resource is specified, an error indication will be returned. If a non-zero value is given for *timeout* and the resource is not available within *timeout* scheduling ticks, then the function returns an error.

Resouce nesting is permitted up to 255 levels.

3

See also: ***chkres, reqres, relres***

Return Value

SUCCESS	resource acquired
E_INVRES	invalid *resrc*
E_TIMED_OUT	timeout expired (resource not acquired)

Example

```
#define printer 6          /* assign resource # */
int status;                /* return status */

/*get the printer, but wait no longer than 1000 ticks*/
status = getres(printer,1000);
if (status == 0)           /* check printer avail */
  {}    /* print report now */

else
  {}    /* invalid resrc */
```

GrpWakeValue

Returns the group event pattern that signaled the task to wake up.

```
uint GrpWakeValue;
```

This function is defined as a macro in the file **mtlib.h**. It returns the pattern of the group event that caused the task to wake up.

See also: ***setgrp, clrgrp, waitgrp, chkgrp***

Return Value

Group event bit pattern that satisfied the ***waitgrp()*** function call

Example

```
waitgrp(id, 0, 0, 7);   /* Wait for any of three bits */
if( GrpWakeValue & 1 ){
   /* bit 0 caused wake up */
}
if( GrpWakeValue & 2 ){
   /* bit 1 caused wake up */
}
if( GrpWakeValue & 4 ){
   /* bit 2 caused wake up */
}
```

incevt

Increments an event.

```
int incevt(uint event);
```

event user-assigned event number (0..*NUMEVT*-1)

The system will add one to the value of *event*. If an invalid *event* is specified, an error indication will be returned. Any task waiting for the *event* to be set will be moved to the run queue if ***incevt*** caused its value to be non-zero. When a task is moved to the run queue, it will preempt the running task if it has a higher priority than that task. You should note that if the event is repeatedly incremented with no task decrementing it, that after a value of 255 the next ***incevt*** operation will wrap the value back to zero, at which point the event is considered "cleared."

3

See also: ***chkevt, clrevt, decevt, wteclr, wteset, wtesetdec***

Return Value

SUCCESS	event incremented
E_INVEVT	invalid event

Example

```
#define data_avl 50             /* assign event # */
int status;                     /* return status */

status = incevt(data_avl);
        /* set data available */

if (status != SUCCESS)          /* check errors */
  {}    /* invalid event num */
```

init_mem_pool

Initializes a memory buffer pool.

```
int init_mem_pool(uint poolid, Mtmem_t *base_ptr,
                  uint buf_size, uint16 buf_count,
                  uint16 pool_type);
```

poolid	ID number of pool {0..*NUMPOOLS*-1*}*
**base_ptr*	base address of pool memory
buf_size	size of each block (see Note below)
buf_count	number of blocks in pool
pool_type	{TASK_POOL or ISR_POOL}

Initializes the memory pool identified by *poolid* for use. The memory beginning at *base_ptr* is divided into *buf_count* blocks each of size *buf_size*. All blocks are linked together. The pool will be for use either by a task(s) or by a single ISR (Interrupt Service Routine) depending upon whether the value of *pool_type* is TASK_POOL or ISR_POOL. A memory pool of type TASK_POOL can be shared by any number of tasks. A memory pool of type ISR_POOL should by used only by a single Interrupt Service Routine, and cannot be accessed by task code.

See also: ***chkbuf, del_pool, getbuf, ireqbuf_c, reqbuf, relbuf***

Return Value

SUCCESS	pool initialized
E_INVPID	invalid *poolid*
E_INVBSZ	invalid *buf_size* {not multiple of sizeof(void *)}
E_INVPT	invalid *pool_type*

Example

```
void *pool0[4*10];      /* Let C compiler allocate memory */

if( init_mem_pool(0, &pool0[0], 4*sizeof(void*), 10, TASK_POOL))
      { /* error routine */ };
```

Defining the memory to be used by `pool0` above as an array of pointers puts the burden of assuring proper memory alignment upon the C compiler and linker.

NOTE: The size of each buffer in a memory pool is required to be a multiple of `sizeof(void *)`. The memory used for the pool should be allocated as above or in some other manner to assure that the memory alignment is proper for accessing a pointer in memory. On 8-bit processors, no alignment is required, but 16- and 32-bit processors usually require memory alignment at least for efficiency.

ireqbuf_c

Requests a memory buffer from ISR.

```
void *ireqbuf_c(uint poolid);
```

poolid ID number of pool {0..*NUMPOOLS*-1}

Returns a pointer to a memory block from the memory pool specified by *poolid*. This call should only be used by an Interrupt Service Routine written in C, or the portion of the routine that is written in C. (See ***ireqbuf*** in the appendices for requesting a block from assembly code in an ISR.) For proper operation, the memory pool specified by *poolid* must have only one ISR making ***ireqbuf_c*** (or ***ireqbuf***) calls from it. The block may be returned to the pool with the ***relbuf*** call from either the ISR or from a task.

NOTE: Since this routine is for use by an ISR only, no error checking is done. You generally must ensure that the pool has a sufficient number of buffers available so the ISR will never be denied one.

See also: ***chkbuf, del_pool, getbuf, init_mem_pool, relbuf, reqbuf***

Return Value

pointer pointer to memory buffer

NULL no buffer available

Example

```
#define POOL0 0      /* PoolID number */
#define MBOX_A 1     /* mailbox for task communication */

char *buffer;        /* pointer to buffer */

/* ISR entry */

buffer = (char *)ireqbuf_c( POOL0 );

/* Fill buffer with data */
/* send message to task which will release the buffer */
MTqcmd_c(SNDMSG, MBOX_A, (MTmsg_t *)buffer, 100);

/* ISR exit */
```

klltsk

Kills a task.

```
int klltsk(TASK_ID slot)
```

slot task slot number (1..NUMTSK, 0 = current task)

The system will remove the task from the specified slot, and release any resources and *LOCAL* memory (at least the stack space) owned by the task. If an invalid system slot is specified, an error indication is returned. If the task being killed is the current task (suicide), ***klltsk()*** will not return. In this case, the next ready task is run instead.

Any resources owned by the task are released and any open streams owned by the task are closed. Memory requested as *LOCAL* memory by ***reqmem()*** is released, but memory obtained as *GLOBAL* memory from ***reqmem()*** as well as any buffers obtained by ***reqbuf()*** or ***getbuf()*** are not released.

WARNING: Do not use ***klltsk()*** on a task that was started with ***runtskss()***.

See also: ***runtsk***

Return Value

SUCCESS	task killed
E_INVSLT	invalid slot number specified

Example

```
uint slot;                /* task slot */
int status;               /* function status */
slot = 3;                 /* use slot 3 */
status = klltsk(slot);    /* kill task in slot */

if (status != SUCCESS)    /* check errors */
  {}    /* invalid slot number */
```

NOTE: When a task reaches the end of its code (closing function brace or “return”), it will automatically execute a ***klltsk()*** of itself unless `INC_KLLTSK` was set to zero in **mtcfg.h**, in which case the system will likely crash.

MASK_INTS

Masks interrupts.

```
void MASK_INTS(void);
```

The ***MASK_INTS()*** macro is defined in the **depends.h** file. It will allow you to prevent interrupts from occurring to protect critical code. Calls to ***MASK_INTS()*** should not be nested.

See also: ***UNMASK_INTS, block_preemption, unblock_preemption***

Return Value

None

Example

```
/* Entering critical code section */
MASK_INTS();        /* Mask interrupts */
/* Execute critical code */
UNMASK_INTS();      /* Unmask interrupts */
```

MTinitialize

Initializes system global tables and variables.

```
void MTinitialize(void);
```

MTinitialize() is called in the program ***main()*** function to initialize the MT! system tables and variables. This call is part of the normal system startup sequence. Call ***MTinitialize()*** before ***usrclk_init()***.

See also: ***MTmeminit, MTmeminit2, MTstart***

Example

See example for ***MTstart.***

MTmeminit

Adds memory blocks to dynamic *COLOR0* pool.

```
int MTmeminit(void *memory_ptr, mem_size_t size);
```

memory_ptr address of memory to add to pool

size number of bytes to add

The system will add *size* contiguous bytes of memory starting at *memory_ptr* to the *COLOR0* pool of free memory managed by ***reqmem()*** and ***relmem()***. ***MTmeminit()*** can accept multiple blocks of memory, but they must be provided in either ascending or descending order. On certain tool chains, you may be able to have the linker and/or startup code pass the actual available memory into your code. Otherwise, use an array of type `long`.

NOTE: ***MTmeminit()*** is implemented as a macro that calls ***MTmeminit2*** for *COLOR0*. On processors where memory alignment can affect access efficiency, you should ensure that *memory_ptr* is aligned on the appropriate boundary (even, quad word, etc.) for greatest efficiency.

See also: ***MTmeminit2***

Return Value

`SUCCESS` memory added to dynamic pool

`E_RELMEM` corrupt memory block header (should never happen)

Example

```
long free_memory[2048];   /* allocate aligned memory */

MTmeminit(free_memory, sizeof(free_memory));

/* absolute memory designated */
MTmeminit((MTmem_t)0xc000, 0x4000);
```

MTmeminit2

Adds memory blocks to a dynamic pool.

```
int MTmeminit2(void *memory_ptr, mem_size_t size,
               uint color);
```

memory_ptr	address of memory to add to pool
size	number of bytes to add
color	pool to add block to {*COLOR0*, *COLOR1*, or *COLOR2*}

The system will add *size* contiguous bytes of memory starting at *memory_ptr* to the *color* pool of free memory managed by ***reqmem()*** and ***relmem().***

NOTE: On processors where memory alignment can affect access efficiency, you should ensure that *memory_ptr* is aligned on the appropriate boundary (even, quad word, etc.) for greatest efficiency.

Return Value

SUCCESS	memory added to dynamic pool
E_RELMEM	corrupt memory block header (should never happen)

Example

```
long free_memory[2048];   /* allocate aligned memory */

MTmeminit2(free_memory, sizeof(free_memory),COLOR0);

/* absolute memory designated */
MTmeminit2((MTmem_t)0xc000, 0x4000, COLOR1);
```

MTqcmd_c

Queues a system call from an interrupt handler.

```
void MTqcmd_c (void (*qfunc)(QARGS *a), ...);
```

qfunc	function label for function to queue
...	arguments taken by the function *qfunc*

The command specified by *qfunc* is added to the system command queue along with the arguments given by .... One of the following predefined labels is used for *qfunc*:

Function	Arguments
CLREVT	uint *event*
CLRGRP	uint *evtgrp*, uint *clr_mask*
DECEVT	uint *event*
DLYTSK	TASK_ID *slot*, byte *dlytyp*, uint *dlytme*
FREERES	uint *resrc*
INCEVT	uint *event*
PRITSK	TASK_ID *slot*, byte *priority*
REANIMATE	TASK_ID *slot*
RELBUF	uint *poolid*, void **bufptr*
RELMEM	void **memptr*
RELRES	uint *resrc*
RUNTSK	uint *priority*, void (**tskptr*)(void), uint *stksze*, . . .
SCDTSK	void
SETEVT	uint *event*
SETGRP	uint *evtgrp*, uint *set_mask*
SNDMSG	uint *mailbox*, void **msgptr*, uint *msgpri*

Function	Arguments
SUSPEND	TASK_ID *slot*
TIKTOK	void
WAKTSK	TASK_ID *slot*

The following three functions are available, but their use is not recommended:

FLUSHMBX	uint *mailbox*
KLLTSK	TASK_ID *slot*
MTTERMINATE	void

These labels have the obvious correspondence to the functions of the same name (in lowercase). The arguments normally given for the function being queued are supplied as the variable arguments to ***MTqcmd_c()***. ***MTqcmd_c()*** is meant to be called from an ISR; although there is no harmful side effect from calling it directly, the reverse is not true. This is the only valid way to call a system function from an interrupt routine! The function is actually executed after the ISR returns from the interrupt or enters ***MTsched*** or ***MTsched_c()*****.** Since the function is called from the command queue and not directly, the ISR cannot get the return value (error status) of the function so called. If the queued command returns an error status when it is called, a bit corresponding to the function called will be set in a long status word, *cmdqerrors***,** which can be tested for diagnostic purposes or by a watchdog task. The bit mask for each function is defined as the same name used for *qfunc* with "_BIT" appended. The mask for SETEVT, for example, is SETEVT_BIT.

Example

```
void interrupt ISR(void)
{
  mt_busy++;        /* prepare for MTsched entry */
  MTqcmd_c(SETEVT, 3);    /* set event 3 */
  MTsched_c();
}
```

MTsched (assembly code only)

Jumps entry point to scheduler from ISR.

Not a function; entry is by assembly-level jump.

ISRs must jump to the system label ***MTsched*** after processing an interrupt. Usually this is the last instruction in the ISR.

The stack must be clean at the time of the jump; i.e., only the processor state is saved by the processor interrupt acknowledged on the stack, unless otherwise specified for the platform being used. ISRs may, at their option, perform a normal return from an interrupt sequence instead of this jump under certain specific conditions.

There is no reason to perform this jump from an ISR that has not used the ***MTqcmd_c()*** function call unless nested interrupts are possible. Its purpose is to provide the quickest possible execution of the commands in the command queue.

The system variable *mt_busy* must be incremented once by the ISR before jumping to ***MTsched*** for proper operation.

See also: ***MTsched_c, MTqcmd_c***

MTsched_c

Scheduler entry from C-level ISR.

void MTsched_c(void);

Provides a scheduler entry from an ISR written in C. This call should be the last call made from an ISR, and the system variable *mt_busy* must be incremented by one before this call is made. This call does not need to be made from an ISR that does not use the ***MTqcmd_c()*** function except under some circumstances detailed in the section on interrupts in the chapter on *MultiTask! Internals*.

The function placing this call must be an interrupt service routine in which all processor registers are saved on entry and restored on exit (after return from ***Mtsched_c()***). If the compiler in use does not allow for the generation of such code, then ISR entry and exit must be from the assembly level.

Because of the way nested interrupts are detected on the 68xxx and 68HC16 processors, this function cannot be used on those platforms. For these, an assembly entry and exit must be made.

See also: ***Mtsched, MTqcmd_c***

Example

```
void interrupt ISR(void)
{
  mt_busy++;        /* prepare for MTsched entry */
  MTqcmd_c(SETEVT, 3);    /* set event 3 */
  MTsched_c();
}
```

MTstart

Starts multitasking operation.

```
void MTstart(void);
```

The function is called in the ***main()*** function to begin executing the first task. Before this function is called, ***MTinitialize()*** and either ***MTmeminit()*** or ***MTmeminit2()*** must be called at least once to define the *COLOR0* memory pool used by the system, and ***runtsk()*** must be called to define at least one task. This call begins multitasking operation by starting the highest priority task defined by a previous call to ***runtsk().*** If ***runtskss()*** is used instead of ***runtsk()***, then the ***MTmeminit()*** call can be omitted. The call to ***MTstart()*** will not return until ***MTterminate()*** is called. In fact, most embedded designs will never call ***MTterminate()*** and therefore will never return from this call.

NOTE: The stack in use by the ***main()*** function is separate from all task stacks, and is used only while the ***main()*** function is executing. This stack is usually defined by the C startup code or the linker command file.

See also: ***MTinitialize, MTmeminit, MTmeminit2, runtsk, MTterminate***

Example

```
#define MEMLOC    0x10000     /* heap start */
#define MEMSIZE   0x8000      /* heap size */
void fAR firsttask(void);     /* task prototype */
int firstid;                  /* save ID of task here */
void main(void)
{

      MTinitialize();   /* initialize system tables */
      usrclk_init();    /* initialize timer interrupt */
      more_init();    /* do additional user intializations*/
```

```
    MTmeminit((MTmem_t)MEMLOC,MEMSIZE); /* init memory */
    firstid = runtsk(100, firsttask, 1024); /* define task */
    if( firstid < SUCCESS )
          exit(1);

    MTstart();              /* we expect no return */
    usrclk_term();          /* stop timer interrupt */
}
```

MTterminate

Stops multitasking and returns to ***main()***.

```
void MTterminate(void);
```

Kills all tasks defined to the system, and performs a return to the main function to the point after the call to ***MTstart().*** The stack is restored to the original stack that was in use in the ***main()*** function. This call must be made by a task or from an ISR via the ***MTqcmd_c(***`MTTERMINATE`***)*** call. Most systems that are designed to operate continuously will never use this call. It is used mainly for systems that are run under another OS, such as an application launched from DOS. In this case, making this call would shut down your application in preparation for return from the ***main()*** function to DOS (or other code that called ***main()***).

See also: ***MTstart***

Example

```
/* control task */
{
  if( message == “exit”)
        MTterminate();
}
```

mt_xxx() (Stream I/O functions)

Functions of the type ***mt_xxx()*** are stream I/O functions and can be found in the *Stream I/O Library Reference* chapter. These include:

mt_clearerr	***mt_fclose***	***mt_feof***	***mt_ferror***
mt_fflush	***mt_fgetc***	***mt_fgetpos***	***mt_fgets***
mt_fopen	***mt_fprintf***	***mt_fputc***	***mt_fputs***
mt_fread	***mt_fseek***	***mt_fsetpos***	***mt_ftell***
mt_fwrite	***mt_mkdir***	***mt_printf***	***mt_remove***
mt_rename	***mt_rmdir***	***mt_sprintf***	***mt_sscanf***
mt_vsprintf			

oneshot

Increments an event at a specified time.

```
int oneshot(uint event, tick_cnt_t time);
```

event	event to increment (1..*NUMPER*)
time	system time to wake up (***get_systime()*** + 1 ..2G)

When the system time equals *time*, the event number specified by *event* will be incremented. (This occurs only once, in contrast to the ***period()*** function, which causes the event to increment repeatedly at a regular interval.)

NOTE: The valid values of *event* are from 1 to *NUMPER*, not *NUMEVT*.

If a second ***oneshot()*** function call is made to the same event before the original *time* is reached, the previous *time* value is canceled and replaced by the new value. It is possible to cancel the ***oneshot*** *event* before the event is set by performing another ***oneshot()*** call with the *time* value less than the current system time, such as (***get_sys_time()***-1). In this case, an error code of E_LATE is returned, but the *time* value of the original call is canceled and no new ***oneshot*** *time* is set, thus clearing the ***oneshot***.

See also: ***delay_until, get_sys_time, period***

Return Value

SUCCESS	OK
E_INVEVT	invalid event number
E_LATE	time is already less than or equal to the current system time

Example

```
int status;               /* return status */
tick_cnt_t now;

now = get_sys_time();     /* get reference time */

status = oneshot(1, now+40);
  /* increment event 1 40 ticks from reference */

if (status != SUCCESS)    /* check errors */
  {}    /* invalid parameter */
```

period

Activates or deactivates a periodic event.

```
int period(uint event, uint period);
```

event event number to be periodically incremented (1..*NUMPER*)

period number of clock ticks of the period

Activates or deactivates a periodic event. If *period* is non-zero, then the periodic *event* will be activated. If *period* is zero, then the periodic *event* will be deactivated. Each active periodic event will be incremented each and every time the associated number of clock ticks has elapsed. For example, if the period for event 1 is set to 10 clock ticks, event 1 will be incremented after every 10 clock interrupts.

NOTE: The valid values of *event* are from 1 to *NUMPER*, not *NUMEVT*.

This function is used in conjunction with ***wteset()*** and ***decevt(),*** or ***wtestedec(),*** to implement periodic tasks.

See also: ***delay_until, dlytsk, oneshot***

Return Value

SUCCESS	period set
E_INVEVT	invalid event number

Example

```
if (period(1,2) != SUCCESS)
        /* inc event 1 after every 2 clock ticks */

  {}    /* error handling code */

/* This task is synchronized to periodic event #1 and
therefore is run every two clock ticks.  */

void period_task (void)
{

  while (TRUE) {
   (void)wtesetdec(1,0);   /* wait for event 1 set */
   { /* etc. */ }
  }
}
```

pritsk

Changes priority of a task.

```
uint pritsk(TASK_ID slot, byte priority);
```

slot	slot number of task to prioritize (1..NUMTSK, 0 = current task)
priority	new priority of the task (0..255)

The system will change the priority of the task identified by *slot* to *priority*. If an invalid system slot is specified, an error indication is returned.

Return Value

SUCCESS	task prioritized
E_INVSLT	invalid slot number specified

Example

```
int status;               /* function status */

status = pritsk(cur_task, 200);
        /* prioritize ourselves */

if (status != SUCCESS)    /* check errors */
  {}    /* invalid slot number */
```

put_mtenv

Sets an environment variable.

```
int put_mtenv(char *member, void *value);
```

member	pointer to the member name string to be added/ updated
value	value to be returned by a later call to ***get_mtenv****(member)*

The environment variable *member* is added to the environment. The value returned by a later call of ***get_mtenv***(*member*) will be *value*. If the variable member previously existed in the environment, its value will be reset to the new value passed in the call. If *value* is NULL, *member* is removed from the environment.

NOTES: Only the pointer values of *member* and *value* are stored in the environment table; the items pointed to are not copied. Consequently, the items pointed to must persist throughout the time that member will reside in the environment table.
Environment table space is configured by the *MTENVSIZE* parameter in **mtcfg.h**.

See also: ***get_mtenv***

Return Value

SUCCESS	member added/updated/deleted from environment
E_ENVFULL	no more space in environment table

Example

```
static int Motor_ID;

  if( put_mtenv(“Motor_task_status”,”Running”) )
     {  /* error handling */ }
  if( put_mtenv(“Motor_task_ID”,&Motor_ID) )

  Motor_ID = cur_task;
     /* error handling */
```

putmsg

Sends a message to a mailbox and waits if mailbox is full.

```
int putmsg(uint mailbox, void *msgptr, uint msgpri,
           uint timeout);
```

mailbox	user-assigned mailbox number (0..NUMMBX-1)
msgptr	points to a message
msgpri	priority of the message (0..255), or *SUPERPRI*, or (0..255)|*MSGSUSPEND*
timeout	ticks to wait if no room in mailbox

The message pointer *msgptr* will be linked into mailbox number *mailbox* at *msgpri* priority. If an invalid mailbox is specified, an error indication will be returned. If the mailbox is full (contains *MBXLIMIT* messages), the sending task will wait until the message can be sent or *timeout* clock ticks elapse. If the value of *msgpri* is *SUPERPRI*, the message will be forced to the front of the mailbox even if a previous *SUPERPRI* message resides there. If (*msgpri* & *MSGSUSPEND*) evaluates as true, then the calling task will be suspended immediately after the message is sent.

See also: ***sndmsg, rcvmsg***

Return Value

SUCCESS	message sent
E_INVMBX	invalid mailbox
E_NOROOM	no more message headers available
E_MBXFULL	MBXLIMIT messages reside in mailbox
E_TIMED_OUT	mailbox still full after timeout

Example

```
int somtsk();              /* task function */
int tskmbx;                /* task mailbox */
int status;                /* return status */

tskslt = runtsk(somtsk,100,200);
        /* run some task */

tskmbx =cur_task;          /* use slot as mailbox */

status = putmsg(tskmbx,(MTmsg_t*) “message to
somtsk”,20,10);
        /* snd msg */
if (status == E_TIMED_OUT)/* check errors */
  {}    /* error handling */
```

putpkt

Sends a message packet and waits if mailbox is full.

```
int putpkt(uint mailbox, void *msgptr, uint msgpri,
           uint size, uint mempool, uint timeout);
```

mailbox	mailbox number to send packet to
msgptr	pointer to message to be sent
msgpri	priority of message
size	size of message in bytes
mempool	memory pool to allocate packet buffer from
timeout	timeout in ticks if mailbox full

The ***putpkt()*** function behaves the same as ***sndpkt()*** except that when the *mailbox* is full, ***putpkt()*** will wait up to *timeout* clock ticks for space to become available. If space is available within *timeout* clock ticks, then the packet is sent. Otherwise an E_TIMED_OUT error status is returned. A *timeout* of zero indicates an infinite timeout. See ***sndpkt()*** for other details.

See also: ***sndpkt, rcvmsg, relpkt***

Return Value

SUCCESS	packet sent successfully
E_INVMBX	invalid mailbox number
E_TIMED_OUT	mailbox full and *timeout* expired
E_NOROOM	no more message headers
E_NORAM	could not allocate memory

Example

```
int status;
status = putpkt(1,”This is a packet”,100,17,COLOR0,20);
switch(status){
  case SUCCESS:
      break;
  case E_TIMED_OUT:
      /* etc. */
```

rcvmsg

Waits for and receives next message in mailbox.

```
void *rcvmsg(uint mailbox, uint timeout);
```

mailbox	user-assigned mailbox number (0..NUMMBX-1)
timeout	timeout delay in ticks

The task will receive a message from *mailbox*. If no message is available at the mailbox, then the task will be placed in a wait-for-message queue. The task remains in the queue until it becomes the highest priority task waiting for a message and a message arrives. If an invalid *mailbox* is specified, a NULL pointer will be returned. The return value is the pointer to the message. If a non-zero value is given for *timeout* and the message is not received within *timeout* scheduling ticks, then a NULL value (0) is returned. If the value PKTRCV is ORed with *mailbox*, ***rcvmsg()*** expects a packet (sent by ***sndpkt()*** or ***putpkt()***) from the mailbox rather than a message, and will return an error if it finds an ordinary message instead.

NOTE: It is not possible to poll a mailbox to retrieve a message.

See also: ***sndmsg, putmst, sndpkt, putpkt***

Return Value

NULL	invalid mailbox, or timeout occurred; *errno* contains the error code
!NULL	address of message, *errno* set to zero

Example

```
PKT_HDR *pkt;        /* pointer if receiving packet */
char *msgadr;        /* received message pointer */
uint tskmbx;         /* task mailbox */

tskmbx = cur_task;   /* use slot as mbx */
msgadr = (char *) rcvmsg(tskmbx,0);

if (msgadr == NULL){ /* check errors */
  if(errno == E_MSGTYPE){ /* is packet, not message */
      pkt = rcvmsg(tskmbx|PKTRCV,0);  /* get packet */
      /* etc. */
  }

}else
  {
  if (*msgadr == “message to somtsk”)
        {           /* chk message content */
        /* process based on message */
        }
```

reanimate

Reactivates a suspended task.

```
int reanimate(TASK_ID slot);
```

slot	slot number of task to suspend (1..NUMTSK, 0 = current task)

The task specified by *slot* will have its suspension flag removed. If the task is in the suspend queue, it will be moved to the run queue. If the task was in any other queue, it will remain there until its wait condition is satisfied.

See also: ***suspend***

Return Value

SUCCESS	task reactivated
E_INVSLT	invalid slot number specified
E_NOTSUS	task was not suspended

Example

```
int     motor_taskid;

motor_taskid = runtsk(200,motor_task,1024);

if( suspend(motor_taskid) < SUCCESS)
  {  /* error process */ }

/* more processing here */

(void)reanimate(motor_taskid);
```

relbuf

Releases a memory buffer.

```
int relbuf(uint poolid, void *bufptr);
```

poolid ID number of pool {0..*NUMPOOLS*-1}

bufptr pointer to memory buffer being returned

Returns the memory buffer pointed to by *bufptr* to the memory pool specified by *poolid*. The memory buffer must have been previously extracted from the pool specified by *poolid* and the value of *bufptr* must be exactly the pointer value that was returned by the ***reqbuf(), getbuf(), ireqbuf(),*** or ***ireqbuf_c()*** function that allocated the buffer. If either of these conditions are not met, the memory pool will become corrupted.

See also: ***getbuf, reqbuf, ireqbuf_c***

Return Value

SUCCESS buffer was successfully released

E_INVPID invalid *poolid* specified

Example

```
#define POOL0 0           /* PoolID number */
char *buffer;             /* pointer to buffer */

if( relbuf(POOL0, (MTmem_t*)buffer) < SUCCESS )
  { /* error handling */ };
```

NOTE: If the buffer is released to the wrong pool, or the value of the *bufptr* released is not one that was previously extracted from the pool, this function will likely return SUCCESS even though the memory pool has been corrupted and will cause a problem later as it is used.

release

Clears a bit in specified table.

```
void release(byte *table, int bitno)
```

table	pointer to the user initialized table
bitno	bit number in the table to clear

The ***release()*** function clears bit number (7 - (*bitno* % 8)) in memory location *table*[*bitno*/8]. The value of *bitno* should be between zero and the number of bits in the table less one. No error checking is done.

See also: ***acquire***

Example

```
byte mbx_aq_table[(NUMMBX+7)/8];
int i;

for(i=0; i<NUMMBX; i++)
  release(mbx_aq_table, i);     /* init table */

  /* assign first available number */
i = acquire(mbx_aq_table, NUMMBX);

  /* use mailbox acquired */

release(mbx_aq_table, i);
```

relmem

Releases memory that was acquired by ***reqmem().***

```
int relmem(void *memptr);
```

memptr address of memory being released

The system will return the memory specified by *memptr* to the free memory pool recombining free memory as possible. The value of *memptr* must be a value that was previously returned by a call to ***reqmem().***

See also: ***Mtmeminit2, reqmem***

Return Value

SUCCESS	released memory to free memory
E_CORRUPT	corrupt memory block header
E_INVCOLOR	corrupted memory block header
E_NULLPTR	*memptr* is NULL
E_UNALLOC	redundant release

Example

```
byte *memptr;                    /* memory pointer */
uint status;                     /* return status */

memptr = (byte *) reqmem(GLOBAL,(mem_size_t)1000);
        /* request memory */

if (memptr != 0)                 /* utilize memory */

  usemem(memptr);                /* use memory */
  status = relmem(memptr);       /* release memory */
  if (status != SUCCESS)         /* check errors */
  {}    /* invalid release */
}
```

relpkt

Releases packet memory.

```
int relpkt(PKT_HDR *pkt);
```

pkt pointer to packet to release

The packet memory allocated by ***sndpkt()*** or ***putpkt()*** is released. Normally this would be done by that task using the ***rcvmsg()*** function to receive the packet after it is finished using the packet.

See also: ***sndpkt, putpkt***

Return Value

SUCCESS no errors

E_INVPID invalid *poolid*

Example

```
PKT_HDR *pkt;
typedef struct ourpkt_s{
  int  count1;
  int  count2;
  double data[16];
  }OURPKT;
OURPKT *ourpkt;

pkt = rcvmsg(MBX,100);          /* get next message */
if( !pkt ){   /* error */
}

ourpkt = &pkt[1]; /* point to our structure in packet */
process(ourpkt);          /* do some processing */
relpkt(pkt);              /* release packet memory */
```

relres

Releases a resource.

```
int relres(uint resrc);
```

resrc user-assigned resource number (0..NUMRES-1)

The system will release resource *resrc* to the highest priority task waiting for the resource. If an invalid resource is specified or the calling task is not the owner of the resource, an error indication will be returned. The ***getres()*** and ***reqres()*** functions may be nested, in which case ***relres()*** will only release the resource when the same nesting level is reached; i.e., if a task calls ***getres()*** and then calls a subroutine that calls ***getres()*** for the same resource, then the resource will remain in the possession of the task until two calls to ***relres()*** for that resources are made.

See also: ***chkres, getres, reqres***

Return Value

SUCCESS	resource released
E_INVRES	invalid *resrc*
E_NOTOWNER	*resrc* not owned by calling task

Example

```
#define printer 6

getres(printer,0);              /* acquire resource */

{  /* use the resource */  }

relres(printer);                /* release resource */
```

reqbuf

Requests a fixed-size memory buffer.

```
void *reqbuf(uint poolid);
```

poolid ID number of pool {0..*NUMPOOLS*-1}

Returns a pointer to a memory block from the memory pool specified by *poolid*. This call should only be used by task code. To request a memory block from an ISR, see ***ireqbuf_c()*** and ***ireqbuf().***

See also: ***chkbuf, getbuf, ireqbuf_c, relbuf***

Return Value

pointer pointer to memory buffer

NULL no buffer available, or E_INVPID | E_INVPT

Example

```
#define POOL0 0           /* PoolID number */
char *buffer;             /* pointer to buffer */

buffer = (char *)reqbuf( POOL0 );
if( buffer == NULL )
   { /* error handling */ };
```

reqmem

Requests a variable-sized memory block.

```
void *reqmem(int memory_type, mem_size_t reqsize);
```

reqsize size of requested memory in bytes

memory_type type of memory requested and its color
{*LOCAL* = memory belongs to task
GLOBAL = memory available to all tasks}
|
COLOR{*0..2*} = memory color

The system will search the free memory blocks, allocating and returning a pointer to memory of a size designated by *reqsize* from the first block that is large enough. If *reqsize* contiguous bytes are not available, a NULL pointer will be returned. The *memory_type* should be one of the values *LOCAL* or *GLOBAL* that are defined in **mtlib.h**. If memory is to come from other than *COLOR0*, the color label defined in **mtlib.h** is ORed with the *memory_type* to specify this.

NOTE: The actual size of the memory block returned may be rounded up to a minimum boundary size to avoid memory fragmentation. You are assured that the size of the block returned is at least *reqsize* bytes. The block returned is preceded by a header linking the memory block with other memory blocks, so avoid writing to any area outside of the block you are returned.

See also: ***Mtmeminit2, relmem***

Return Value

NULL insufficient memory

address pointer to memory

Example

```
byte *memptr;                   /* memory pointer */
uint status;                    /* return status */

memptr = reqmem(LOCAL|COLOR2, (mem_size_t)512);
        /* get 512 bytes */

if (memptr == NULL)             /* for current task */
  {}    /* check errors */
        /* 512 bytes not avail */
```

reqres

Gets a resource if it is available.

```
int reqres(uint resrc);
```

resno `user-assigned` resource number *(0..*`NUMRES`*-1)*

The system will check the status of resource *resrc*. If the resource is available, the resource will be assigned to the task. If the resource is unavailable, an error indication will be returned. If an invalid resource is specified, an error indication will be returned.

See also: ***chkres, getres, relres***

Return Value

`SUCCESS`	resource acquired
1..`NUMTSK`	resource owner's task slot
`E_INVRES`	invalid *resrc*

Example

```
#define printer 6          /* assign resource # */
int owner;                 /* owner of resource */

if(owner=reqres(printer)==SUCCESS)
        /* check printer avail */

  {}    /* print report now */
else
  {}    /* display owner */
```

runtsk

Adds a new task to the run queue.

```
int runtsk (uint priority, void (*tskptr), uint stksze,
            ...);
```

priority	task priority (0..255)
tskptr	address of the “task” function
stksze	stack size for task
...	if present, up to four arguments are passed to task

MT! will place the task pointed to by *tskptr* into a system slot at the priority designated by *priority*, with a task stack of *stksze* bytes and return the slot number. If no slot is available, an error indication is returned. Zero is the lowest priority, 255 is the highest priority. Up to four arguments may be passed to the task as the variable arguments appearing after *stksze*. If the task takes arguments, it will be necessary to cast *tskptr* as a function without arguments to avoid compiler warnings (or errors for C++).

NOTE: Some ports have additional information passed with the priority.

See also: ***klltsk, runtskss***

Return Value

E_NOSLOT	no slot available
E_NORAM	no RAM available for task stack
1..NUMTSK	task’s assigned slot number

Example

```
void task1(void);               /* task function */
void task2(char *state, int count);
int slot1,slot2;                /* task slot */

  slot1 = runtsk(100,task1,1024); /* run the task */

if (slot < SUCCESS)             /* check errors */
  { }   /* no stack or no slot */

slot2 =
runtsk(120,(void(*)(void))task2,1500,"Active",5);
        /* run with task args */
if (slot < SUCCESS)             /* check errors */
  { }   /* no stack or no slot */
```

runtskss

Adds a new task with a static stack to the run queue.

```
int runtskss (uint priority, void (*tskptr),
              uint stksze, void *stkbase, ...);
```

priority	task priority (0..255)
tskptr	address of the “task” function
stksze	stack size for task
stkbase	starting stack base
...	if present, up to four arguments are passed to task

MT! will place the task pointed to by *tskptr* into a system slot at the priority designated by *priority*, with a task stack starting at *stkbase* with *stksze* bytes and return the slot number. If no slot is available, an error indication is returned. Zero is the lowest priority, 255 is the highest priority. Up to four arguments may be passed to the task as the variable arguments appearing after *stksze*. If the task takes arguments, it will be necessary to cast *tskptr* as a function without arguments to avoid compiler warnings (or errors for C++).

NOTES: Some ports have additional information passed with the priority.
This function is not compatible with *LOCAL* memory tracking.

See also: ***klltsk, runtsk***

Return Value

E_NOSLOT	no slot available
1..NUMTSK	task’s assigned slot number

Example

```
#define TASK_STKSIZ 400

uint32  ssstack1[(TASK_STKSIZ+3)/4],
        ssstack2[(TASK_STKSIZ+3)/4];

void task1(void);               /* task function */
void task2(char *state, int count);
int slot1,slot2;                /* task slot */

/* run the task */

slot1 = runtskss(100, task1, sizeof(ssstack1), ssstack1);

if (slot1 < SUCCESS)            /* check errors */
  { }   /* no slot */

/* run with task args */

slot2 = runtskss(120, (void(*)(void))task2,
        sizeof(ssstack1), sstack2, "Active", 5);

if (slot2 < SUCCESS)            /* check errors */
  { }   /* no stack or no slot */
```

scdtsk

Reschedules tasks in run queue.

```
void scdtsk(void);
```

The system will schedule the next runnable task. This function allows a task to relinquish the remainder of its time slice without using a time delay, event wait, message wait, or a resource wait.

NOTE: Time-slicing is always only among tasks of equal priority and may be switched off by a compile-time option. The only situation when this function is of any use is when there is at least one other task of the same priority as the task making this call. The ***scdtsk()*** function will never cause a lower priority task than the caller to run, and any higher priority task would already be running. To give other priority tasks an opportunity to run, use ***dlytsk(****0, DLY_TICKS, 1****)***.

Return Value

`SUCCESS` OK

Example

```
#define warmup 2          /* define warmup event */

while (chkevt(warmup) == 0)
  {
  scdtsk();               /* let other tasks run */
  }
```

setclk

Sets time of day clock.

```
int setclk(byte hour, byte minute, byte second);
```

hour	hour of the day (0..23)
minute	minute of the hour (0..59)
second	second of the minute (0..59)

The time of day clock implemented by the ***time_keeper()*** task will be set to the hour, minute, and second specified. If an invalid value is specified, an error indication will be returned. A 24-hour clock is used. The valid range for hours is 0 to 23. The valid range for minutes or seconds is 0 to 59.

NOTE: The ***time_keeper()*** task must have been started for the system clock to keep time.

Return Value

SUCCESS	OK
E_INVTME	invalid parameter

Example

```
int status;                     /* return status */

status = setclk(10,35,20);
        /* set time 10:35:20 */

if (status != SUCCESS)          /* check errors */
  {}    /* invalid parameter */
```

setevt

Sets an event.

```
int setevt(uint event);
```

event user-assigned event number (0..*NUMEVT*-1)

The system will set *event* by setting its value to one. If an invalid *event* is specified, an error indication will be returned. Any task waiting for the event to be set will be moved to the run queue where it will preempt the running task if it has a higher priority than that task.

See also: ***clrevt, decevt, incevt***

Return Value

SUCCESS	event set
E_INVEVT	invalid event

Example

```
#define data_avl 50             /* assign event # */
int status;                     /* return status */

status = setevt(data_avl);
        /* set data available */

if (status != SUCCESS)          /* check errors */
  {}    /* invalid event num */
```

setgrp

Sets bits in a group event.

```
int setgrp(uint evtgrp, uint set_mask);
```

evtgrp	user-assigned group event number (0..NUMGEVT-1)
set_mask	16-bit mask of bits to set

The bits that are set in *set_mask* are set in the group event specified by *evtgrp*. If an invalid value is passed for *evtgrp*, an error indication is returned. Any tasks that are waiting for the resulting group event bit pattern will be moved to the run queue where they will preempt the running task if they have a higher priority.

See also: ***clrgrp, waitgrp***

Return Value

SUCCESS	OK
E_INVGRP	invalid *evtgrp*

Example

```
#define data_avl 50
int status;
        /* event # */

status = setgrp(data_avl, 0x2315);
        /* set bits */
if(status != SUCCESS)
  {}    /* invalid event number */
```

slttsk

Returns the slot number (TASK_ID) of a task.

```
int slttsk( void (*tskptr)(void) );
```

taskptr function pointer to task function

If *tskptr* is zero (NULLFP), the current task's slot number is returned. Otherwise the slot number of the first task slot whose starting address equals *tskptr* is returned. If no slot is found with a task starting address that matches *tskptr*, an error indication is returned. The ***slttsk()*** function is mostly an historical remnant. The slot number of a task is better obtained by storing the return value from ***runtsk()*** when the task is started, or for the currently running task using the value in the global variable *cur_task*.

Return Value

1..NUMTSK slot number of matching task slot

E_INVSLT no task slot found that matches *tskptr*

Example

```
void fndtsk();            /* task function */
int ourslot;              /* task slot */
int fndslot;              /* task slot */
ourslot = slttsk(0);      /* find our slot */

fndslot = slttsk(fndtsk);
        /* find fndtsk slot */

if (fndslot < 0)          /* check errors */
  {}    /* invalid tskptr */
```

sndmsg

Sends a message to a mailbox.

```
int sndmsg(uint mailbox, void *msgptr, uint msgpri);
```

mailbox	user-assigned mailbox number (0..NUMMBX-1)
msgptr	points to a message
msgpri	priority of the message (0..255)), or *SUPERPRI*, or (0..255)|*MSGSUSPEND*

The message pointer *msgptr* will be linked into *mailbox* at the priority specified by *msgpri*. If an invalid *mailbox* is specified, or the mailbox is full, or no more message headers are available, an error indication will be returned. If the value of *msgpri* is *SUPERPRI*, the message will be forced to the front of the mailbox even if a previous *SUPERPRI* message resides there. If (*msgpri* & *MSGSUSPEND*) evaluates as true, then the calling task will be suspended by the ***suspend()*** function immediately after the message is sent.

See also: ***putmsg, rcvmsg***

Return Value

SUCCESS	message sent
E_INVMBX	invalid mailbox
E_NOROOM	no more message headers available
E_MBXFULL	mailbox is full (configured limit reached)

Example

```
int somtsk();              /* task function */
int tskslt;                /* task slot */
int tskmbx;                /* task mailbox */
int status;                /* return status */

tskslt = runtsk(somtsk,100,200); /* run some task */

tskmbx = tskslt;           /* use slot as mailbox */

status = sndmsg(tskmbx,(MTmsg_t*)"message to somtsk",20);
        /* snd msg */

if (status != SUCCESS)     /* check errors */
  {}    /* invalid mailbox num */
```

sndpkt

Sends a message packet.

```
int sndpkt(uint mailbox, void *msgptr, uint msgpri,
           uint size, uint mempool);
```

mailbox	mailbox number to send packet to
msgptr	pointer to message to be sent
msgpri	priority of message
size	size of message in bytes
mempool	memory pool to allocate packet buffer from

A memory block is allocated from the source indicated by *mempool*, and the message pointed to by *msgptr* of *size* bytes is copied to the new memory block with a packet header prepended. The packet (consisting of the packet header followed by a copy of the message) is placed in *mailbox* at the priority *msgpri*. Packets use the same mailboxes as messages, and are received by issuing a ***rcvmsg()*** function call with the macro PKTRCV ORed to the *mailbox*.

The *mempool* can be one of the variable-size pool names: *COLOR0*, *COLOR1*, or *COLOR2*, or it can be the buffer pool ID for a fixed-size buffer pool. If the variable-size pool names are used, memory will be allocated with a ***reqmem()*** function call. If a buffer pool ID is used, a ***reqbuf()*** call will be used to allocate the memory. If you are using a fixed-size buffer array, you must initialize it before using ***sndpkt()*** with the ***init_mem_pool()*** function, and the size of the buffers must be at least *size*+sizeof (*PKT_HDR*).

See also: ***putpkt, rcvmsg, relpkt***

Return Value

SUCCESS	packet sent successfully
E_INVMBX	invalid mailbox number
E_MBXFULL	mailbox message limit reached
E_NOROOM	no more message headers
E_NORAM	could not allocate memory

Example 1

```
if( sndpkt(1,"This is a packet",100,17,COLOR0) )
      { /* some error occurred */
```

Example 2

```
/* Using fixed size buffers for message packets: */
#define MBX      2
#define POOL0    0
#define MSGSIZE 20            /* message size we want */

/* add sizeof(PKT_HDR)*/
/* and round up to sizeof(void*) multiple */
#define BUFSIZE
((MSGSIZE+sizeof(PKT_HDR)+sizeof(void *)-1)&(~(sizeof(void *)-1)))

/* Let C compiler allocate aligned memory */
void *pool0[BUFSIZE*10/sizeof(void*)];

char *msg1[MSGSIZE];

if( init_mem_pool(POOL0, &pool0[0], BUFSIZE, 10, TASK_POOL))
      { /* error routine */ };

{ /* code to fill in msg1 with something useful */ }

if( sndpkt(MBX,msg1,200,MSGSIZE,POOL0) )
      { /* error handling */ }
```

suspend

Suspends a task.

```
int suspend(TASK_ID slot);
```

slot slot number of task to suspend (1..NUMTSK, 0 = current task)

The task specified by *slot* will be flagged as suspended. If the task was in the run queue, it is moved to the suspend queue where it will remain until reactivated by the ***reanimate()*** function. If the task was waiting in some other queue (i.e., delayed, or waiting for and event or message, etc.), it will stay in that queue until its delay expires, event is set, etc. When the condition it was waiting for is met, it will move to the suspend queue instead of the run queue. When you reanimate such a task, it will receive its proper status from such a wait (i.e., "message received" or "timeout").

See also: ***reanimate***

Return Value

SUCCESS	task suspended
E_INVSLT	invalid slot number specified

Example

```
int     motor_taskid;

motor_taskid = runtsk(200,motor_task,1024);

if( suspend(motor_taskid) < SUCCESS)
  {  /* error process */ }
```

timed_xxx (Stream I/O functions)

Functions of the type ***timed_xxx()*** are Stream I/O functions and can be found in the *Stream I/O Library Reference* chapter. These include:

timed_getc ***timed_read*** ***timed_readln***

unblock_preemption

Enables task switching.

```
void unblock_preemption(void);
```

The ***unblock_preemption()*** function enables task switching to occur. This should be used to counter the effects of a ***block_preemption()*** call.

See also: ***block_preemption, MASK_INTS, UNMASK_INTS***

Return Value

None

Example

```
/* Entering critical code section */
block_preemption();       /* Prevent task switches */
/* Execute critical code */
unblock_preemption();     /* Allow task switching */
```

UNMASK_INTS

Removes interrupt mask.

```
void UNMASK_INTS(void);
```

The ***UNMASK_INTS()*** macro is defined in the **depends.h** file. It enables interrupts from occurring after a call to ***MASK_INTS()*** has prevented them.

See also: ***MASK_INTS, block_preemption, unblock_preemption***

Return Value

None

Example

```
/* Entering critical code section */
MASK_INTS();        /* Mask interrupts */
/* Execute critical code */
UNMASK_INTS();      /* Unmask interrupts */
```

waitgrp

Waits for a group event.

```
int waitgrp(uint evtgrp, uint set_mask, uint clr_mask,
            uint or_mask, uint timeout);
```

evtgrp	user-assigned group event number (0..NUMGEVT-1)
set_mask	bit mask of bits to wait until set
clr_mask	bit mask of bits to wait until clear
or_mask	bit mask of bits to wait for any set
timeout	timeout delay in scheduling ticks 0 = infinite delay (no timeout)

The calling task is put to sleep (deactivated) until either the specified group event bit pattern matches, or the timeout delay period specified by *timeout* expires. The group event bit pattern matches when all of the bits set in *set_mask* are also set in the event variable and all of the bits set in *clr_mask* are clear in the event variable, and any one or more of the bits set in *or_mask* are set in the event variable. Any bits that are not set in *set_mask* or *clr_mask* or *or_mask* are ignored in the event variable. If ***waitgrp()*** returns SUCCESS, then the value of the macro GrpWakeValue contains the group event value that woke the task.

See also: ***chkgrp, clrgrp, setgrp***

Return Value

SUCCESS	specified group event has occurred
E_INVGRP	invalid *evtgrp*
E_TIMED_OUT	*timeout* has expired

Example

```
#define data_avl 50                 /* event name */
int status;

status = waitgrp(data_avl, 0x0234, 0x0481, 0xf000, 1000);
if(status == SUCCESS){              /* event occurred */
      switch(GrpWakeValue & 0xf000){
        case 0x8000:
        case 0x4000:
      /* etc. */
```

waktsk

Returns a delayed task to the run queue.

```
int waktsk(TASK_ID slot);
```

slot slot number of task to wake (1..NUMTSK)

The system will move the delayed task designated by *slot* from the time delay queue back into the run queue. If an invalid slot is specified or the task is in the undefined queue, an error indication will be returned. The ***waktsk()*** function will immediately wake up a task from the time delay queue, except one that has been suspended by the ***suspend()*** function. If the task is not in the time delay queue when ***waktsk()*** is called, then the next attempt to delay the task will be canceled.

See also: ***delay_until***, ***dly_tsk***

Return Value

SUCCESS OK

E_INVSLT invalid slot

Example

```
TASK_ID slot;                   /* task slot */
int status;                     /* return status */
status = waktsk(slot);          /* wake sleepy task */
if (status != SUCCESS)          /* check errors */
  {}    /* inv slot or queue */
```

wketsk (obsolete)

Returns a delayed or waiting task to the run queue.

```
int wketsk(TASK_ID slot);
```

slot slot number of task to wake (1..NUMTSK)

The system will move the delayed task designated by *slot* from any wait queue back into the run queue. If an invalid slot is specified or the task is in the undefined queue, the run queue, or is suspended, an error indication will be returned. The ***wketsk()*** function will immediately wake up a task from any queue, except one that has been suspended by the ***suspend()*** function. If the task was waiting for a resource, message, or event, the status return value that task receives will indicate that the timeout expired (i.e., E_TIMED_OUT).

NOTE: We recommend that you avoid this routine unless you are very certain of its effects.

Return Value

SUCCESS	OK
E_INVSLT	invalid slot

Example

```
TASK_ID slot;                   /* task slot */
int status;                     /* return status */
status = wketsk(slot);          /* wake sleepy task */
if (status != SUCCESS)          /* check errors */
  {}    /* inv slot or queue */
```

wketsk_nto (obsolete)

Returns a delayed or waiting task to the run queue.

```
int wketsk_nto(TASK_ID slot);
```

slot slot number of task to wake (*1..NUMTSK*)

This call is identical to the ***wketsk()*** call, except the task that is awakened will not return the (E_TIMED_OUT) status. (The ***_nto*** stands for "no timeout.")

The system will move the delayed task designated by slot from any wait queue back into the run queue. If an invalid slot is specified or the task is in the undefined queue, an error indication will be returned. The ***wketsk()*** function will immediately wake up a task from any queue, except one that has been suspended by the ***suspend()*** function. The task status will not indicate a timeout. This is mainly for use with tasks waiting in I/O queues, but may have other applications.

NOTE: This is even more risky to use than ***wketsk()***. Use with extreme caution.

Return Value

SUCCESS	OK
E_INVSLT	invalid slot

Example

```
TASK_ID slot;  /* task slot */
int status;    /* return status */
status = wketsk_nto(slot); /* wake sleepy task */
if (status != SUCCESS)     /* check errors */
  {}    /* inv slot or queue */
```

wteclr

Waits until event is clear.

```
int wteclr(uint event, uint timeout);
```

event	user-assigned event number (0..*NUMEVT*-1)
timeout	timeout delay in scheduling ticks

The calling task will wait until the value of *event* is zero or *timeout* clock ticks have elapsed. If the value of *event* is set (i.e., non-zero), then the task will be placed in a queue to wait for the event to be cleared. The task will continue when the event has been cleared or decremented to a zero value. If the event value is zero when the call is made, the call will immediately return SUCCESS (i.e., the task does not wait). A *timeout* value of zero specifies no timeout. If an invalid event is specified, an error indication will be returned. If *timeout* is non-zero, ***wteclr()*** will return an error indication after *timeout* scheduling ticks if the event was not cleared.

See also: ***wteset, wteset_dec***

Return Value

SUCCESS	event clear
E_INVEVT	invalid event
E_TIMED_OUT	timeout expired (event not clear)

Example

```
#define data_avl 50             /* assign event # */
int status;                     /* return status */

status = wteclr(data_avl, 500);
        /* wait for event clear, but no
        more than 500 ticks */

if (status != SUCCESS)          /* check errors */
  {}    /* invalid event num */

else
  {
  gather_data();                /* gather more data */
  setevt(data_avl);             /* data avail */
  }
```

wteset

Waits until event set.

```
int wteset(uint event, uint timeout);
```

event	user-assigned event number (0..*NUMEVT*-1)
timeout	timeout delay in scheduling ticks

The calling task will wait until the value of *event* is non-zero or *timeout* clock ticks have elapsed. If the value of *event* is clear (i.e., zero), then the task will be placed in a queue to wait for the event to be set. The task will continue when the event has been set or incremented to a non-zero value. If the event value is non-zero when the call is made, the call will immediately return SUCCESS (i.e., the task does not wait). A *timeout* value of zero specifies no timeout. If an invalid event is specified, an error indication will be returned. If *timeout* is non-zero, then ***wteset()*** will return an error indication after *timeout* scheduling ticks if the event was not set.

See also: ***wteclr, wteset_dec***

Return Value

SUCCESS	event set
E_INVEVT	invalid event
E_TIMED_OUT	timeout expired (event not set)

Example

```
#define data_avl 50             /* assign event # */
int status;                     /* return status */

status = wteset(data_avl, 0);
        /* wait on data avail */

if (status != SUCCESS)
  {}    /* check errors */

else
  {     /* invalid event num */
  process_data();               /* process data */
  clrevt(data_avl);             /* no data avail */
  }
```

wteset_dec

Waits until event set and decrements it.

```
int wteset_dec(uint event, uint timeout);
```

event event number {0 .. *NUMEVT*-1}

timeout timeout clock ticks (0 = no timeout)

The calling task will wait until the value of *event* is non-zero or *timeout* clock ticks have elapsed. If the *event* is set at the time of the call, or becomes set before *timeout* clock ticks have elapsed, then the function returns SUCCESS and decrements the *event*. A *timeout* value of zero specifies no timeout. If an invalid event is specified, an error indication will be returned. If *timeout* is non-zero, ***wteset()*** will return an error indication after *timeout* scheduling ticks if the event was not set.

This function eliminates the need to perform a separate ***decevt()*** call to recycle the event, thus saving time; however, no check is made when the event is decremented for tasks waiting for this condition. You should, therefore not have any tasks calling ***wteclr()*** for any *event* you are specifying in a ***wteset_dec()*** call.

See also: ***wteclr, wteset***

Return Value

SUCCESS event set

E_INVEVT invalid event

E_TIMED_OUT timeout expired (event not set)

Example

```
for(;;){                    /* repeat forever: */
  wteset_dec(50, 0);        /* wait for event 50 */
  process_data();           /* process data */
}
```

4. MultiTask! Internals

Chapter Contents

Overview

This chapter will largely deal with handling various interrupt situations with MultiTask!. Following the interrupt discussion are sections describing more detailed internal issues that may prove useful in debugging an application. These sections will also provide a more complete understanding of MultiTask!'s operations.

Interrupt Basics

Interrupts are a hardware engineer's idea of multitasking. On most processors, these can be coupled to the RTOS (MultiTask!) to provide communication from the hardware devices to the software tasks.

In a single-tasking system, interrupts are implemented as sub-routine calls triggered by hardware status lines -- called Interrupt Request Lines. When the processor sees such a line active, and an internal Interrupt Enable Flag is `TRUE`, then the processor injects a synthesized subroutine call into the instruction stream. The interrupt routine is then run.

Since processors differ greatly, and looking at a few will provide a basis for understanding the issues that MultiTask is faced with.


```
8080_isr:
   DI    ; not automatic
   <work>
   EI
   RET
```

In the code above, the 8080 did not provide any way to read the Interrupt Enable Flag. In addition, an interrupt entry did not disable interrupts although it did guarantee that one instruction would be executed before another interrupt occurred.

```
generic_isr:
   ; PC & flags are already saved, interrupts disabled
   <work>
   RetI  ; special instruction
```

The flags registers, or a special 'systems' section thereof, contains the Interrupt Enable Bit(s). There may also be a 'User Mode' bit, which is also turned off when an interrupt or trap occurs.

```
ColdFire_isr:  ; supports 7 levels in hardware
   ; PC & flags & vector are pushed on stack
   ; interrupt mask level is set to the current level
   <work>
   RTE   ; return from exception (interrupt or trap)
```

The 68K and ColdFire (and 68HC16) support 7 levels of interrupts. Since this is hardware supported, MultiTask! requires special code to detect when a nested situation occurs.

Multilevel Interrupts – MT! Visibility

Some applications require an interrupt with an extremely fast response. On processors that support multiple levels of hardware, it is possible to modify MultiTask!'s use of interrupt levels to avoid interfering with this requirement. Basically, MT!'s view of interrupts is based on the macros defined in **depends.h.** If the user has a separate set, then changing MT!'s macros can isolate or split out some interrupt levels.

For example, let's say that a 68K/ColdFire system has a single level 6 interrupt. This interrupt does not interface with MT!, meaning that ***MTqcmd_c()*** and ***MTsched*** are not used. We can then cheat on MT! by changing `UNMASK_INTS()` to raise the interrupt level only to 5 instead of 7. This will still block any interrupt that may interface to MT!, but it will allow the special interrupt to occur – even inside MT!'s critical code sections!

Unfortunately, most RISC processors also have reduced interrupt hardware.

Interfacing to MultiTask!

There are three parts to consider when hooking an interrupt to MultiTask!. The first is how to send a MultiTask! task something - event, message, wakeup, etc. The second is how to let an interrupt routine get something from MultiTask!. The third is how the interrupt entry and exit must be changed for MultiTask!.

Talking to MultiTask! Objects

Interrupt Service Routines (ISRs) may only call MultiTask! functions indirectly through the ***Mtqcmd_c()*** function. Interrupt routines must not call MultiTask! functions directly! Doing so will result in crashes. The ***MTqcmd_c()*** function allows a subset of the MultiTask! system calls to be triggered from an ISR. This triggering is accomplished by placing a request for the desired system call into the command queue. The requests are executed just before a task is resumed - see below.

Requests are made through ***MTqcmd_c(****id,. . .****)***. These system calls are available:

```
SETEVT event #
CLREVT event #
INCEVT event #
DECEVT event #
SETGRP group #, bits
CLRGRP group #, bits
SNDMSG mailbox #, address, priority
WAKTSK task #
DLYTSK task #, unit, amount
SUSPEND task #
REANIMATE task #
PRITSK task #, priority
RUNTSK priority, address, stack size
KLLTSK task #
```

```
RELRES resource #
RELBUF pool #, address
RELMEM address
SCDTSK
TIKTOK
```

(these next 3 calls are not recommended)

```
FREERES resource #
FLUSHMBX mailbox #
MTTERMINATE
```

(these next 3 calls are obsolete)

```
WKETSK task #
WKETSK_NTO task #
PULSEVT event #
```

Example:

```
MTqcmd_c(SETEVT,5);
```

Getting Something from MultiTask!

The only facility that an interrupt routine can use from MultiTask! is ***ireqbuf()***, which acquires a buffer from a buffer poll set aside for use by interrupts.

Entry/Exit Adjustments

In most processors, interrupts are distinct and atomic. That is, once an interrupt routine starts, it runs to completion with no other interrupts able to interrupt it. Thus, there is no preemption among interrupts. Those systems where either the hardware or software allow nesting of interrupts require slightly different adjustments. The figure below shows this simple situation, which most designers are probably familiar with:

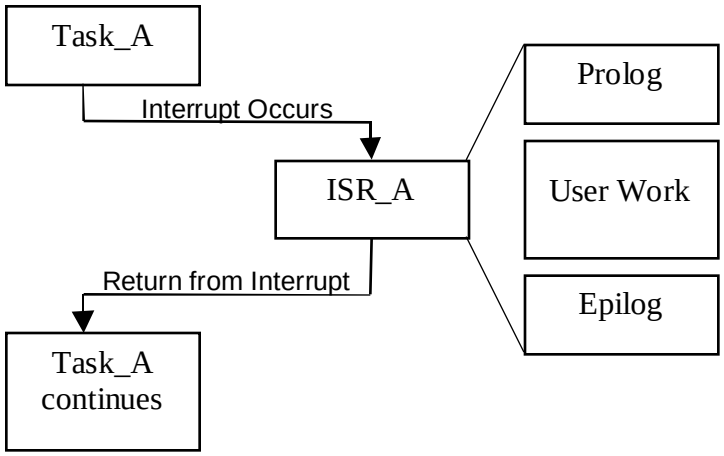

Figure 4-1: Simple interrupt situation

Some task (Task_A) is running, and the interrupt occurs and is acknowledged, causing the processor to vector to ISR_A.

The ISR completes its processing and returns (usually with a special interrupt return or exception return instruction).

The original task (Task_A) resumes where it was interrupted.

For assembly language interrupt routines on a simple processor, you can increment the variable *mt_busy* and replace the interrupt return instruction with a jump to ***MTsched***. ***MTsched*** will handle all the MultiTask! issues. If you have an extended C compiler with the `__interrupt` keyword, you can use C and add `++mt_busy; MTsched_c();` to the very end of your interrupt routine.

```
generic_isr:
   ; PC & flags are already saved, interrupts disabled
   <work>
   INC   mt_busy
   JMP   MTsched     ; to MultiTask

void __interrupt my_isr(void){
   <work>
   ++mt_busy; MTsched_c();
}
```

Depending upon the processor involved, the processor registers are either automatically saved when the interrupt occurs, or it will be the responsibility of the ISR to do this and then restore them before returning. In either case, the machine state will be saved upon entry to the ISR and restored upon exit, which will cause Task_A to resume with a machine state identical to when it was interrupted.

The next figure shows a slightly more complicated scenario. In this case, instead of the ISR performing a return from interrupt, it queues some operating system service that will result in a task preemption, and then branches to the MultiTask! scheduler entry point ***MTsched_c()***. The scheduler then causes a task switch to occur and resumes execution of Task_B, rather than Task_A. This is the common case where an ISR will cause a task switch to happen. The clock interrupt service routine ***usrclk()*** does just this whenever a task wakes up from a timeout or delay, or when time-slicing takes place.

Example ISR installed via a wrapper layer for handing ISR and MT! interfaces:

```
void my_handler(uint dev_num){
   <do processing>
}
```

Example ISRs installed into hardware interrupt vector table:

```
void __interrupt my_handler(void){
/* Note: a few processors provide vector # on entry */
   ++mt_busy;
   <do processing>
   MTsched_c();      /* does MT! work */
}

my_handler:
   PUSH <some registers>
   <do processing>
   INC mt_busy
   POP <some registers>
   JMP MTsched
```


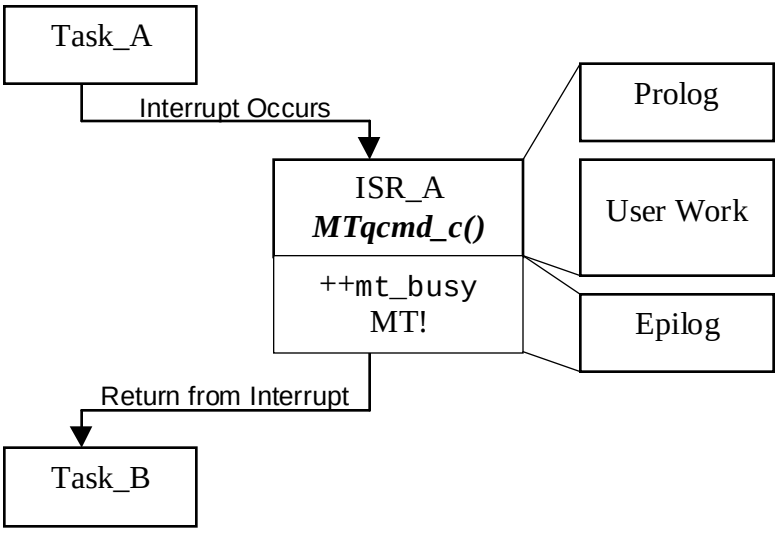

Figure 4-2: Interrupt with task switch

At first, the thought that a return from interrupt is not issued by the ISR may seem a little alarming. The scheduler may actually perform the return from interrupt if necessitated by the machine architecture or perform a functionally equivalent operation in the course of switching tasks. Otherwise, it changes from the hardware interrupt disabled state to a lower, kernel state controlled by *mt_busy*.

The actual "return from interrupt" performed when switching to Task_B will actually be restoring the context from when Task_B was interrupted. When Task_A is interrupted, its context (registers) are saved (generally on a stack frame) and these will not be restored until we return again to Task_A. This will often be on another interrupt, as is the case with tasks that are time-slicing, i.e., the clock interrupt is initiating the task-switch.

Nested Interrupt Issues

The next figure illustrates a simple case where interrupt service routines are nested. On some processors this can only occur if ISR_A executes instructions to allow further interrupts to be processed (i.e., re-enables interrupts). On machines with multiple interrupt priority levels, such as the 68000 family and i960, this action is controlled by the interrupt priority built into the CPU, and a nesting situation can only be avoided by assigning all interrupt sources the same priority. This is not always possible.

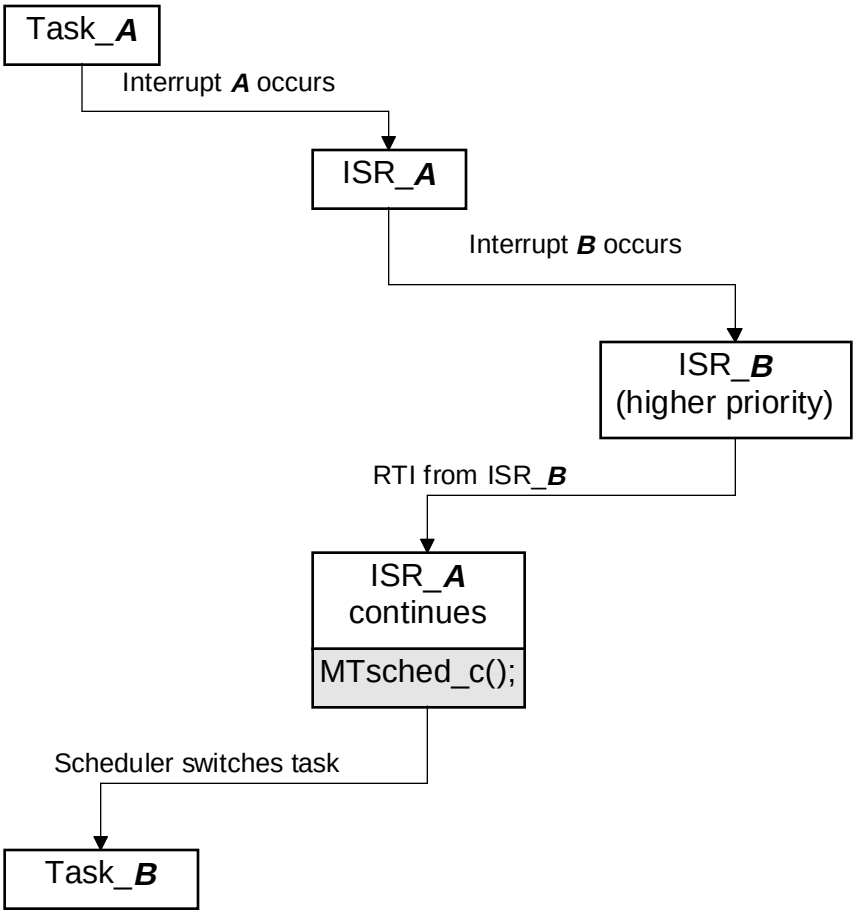

Figure 4-3: Nested interrupt routines

With the arrangement shown here, there is really no side effect of having ISR_B nested within ISR_A except that the execution of ISR_A takes longer than it would otherwise.

The next figure shows a situation that can be a problem. Task_A is interrupted by interrupt A. A higher-priority interrupt (B) then occurs and ISR_B begins execution. On processors with built-in interrupt priorities (68000, i960, and others), it is possible that interrupt A has been acknowledged by the processor but not one instruction of ISR_A has been executed before the processor acknowledges and vectors to ISR_B.

If ISR_B now queues a command that will cause task preemption and then exits by branching to the scheduler as illustrated, then the scheduler may start some other task (Task_B). From this point on, it is possible that other tasks may run an indefinite period of time before conditions dictate that the scheduler resume execution of Task_A. ISR_A is at this point considered a part of Task_A, since it is the context that was saved when Task_A was last preempted. So when the scheduler resumes Task_A, ISR_A is continued, and ISR_A's return from interrupt returns into Task_A at the point the original interrupt A had occurred.

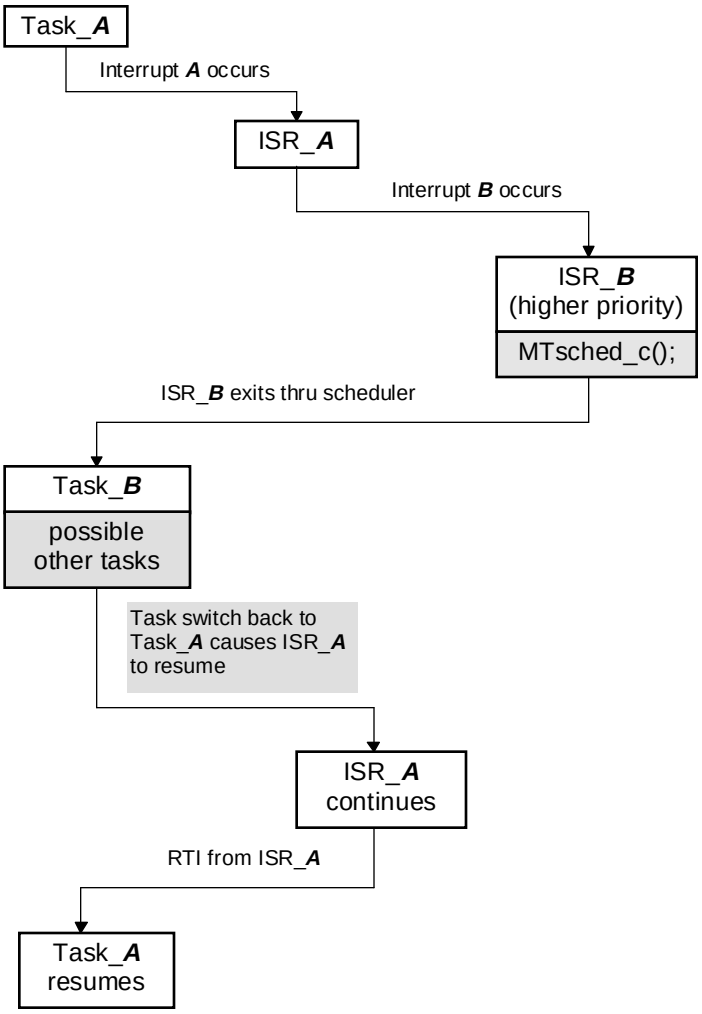

Figure 4-4: Possible interrupt problem

It is also entirely possible that Task_B could kill Task_A, in which case ISR_A can never be completed and interrupt A goes unprocessed.

This is not always a problem and may be in fact what you intended when assigning interrupt priorities, but you must design your interrupt handling with this in mind.

For hardware nested interrupts, you must copy and expand ***MTsched_c()*** into your routine. In addition, you must be able to access, from C, the saved flags that the C-generated prolog has pushed onto the stack. Finally, you must make sure the C-generated prolog saves all required registers.

```
void __interrupt my_isr(exception_frame_t X){
  <work>
  MASK_INTS();          /* *NO* interrupts allowed */
  /* interrupted a task? (vs system call) */
  if( 0 == mt_busy
  /* Something in command queue? */
  &&  cmdadd != cmdprc
  &&  X.interrupt_level == fully_enabled
  ){
        UNMASK_INTS();  /* kernel level */
        ++mt_busy;
        MTqproc();
        /* required by some RISC machines */
        MASK_INTS();
  }
}
```

For software nested interrupts, we rearrange things a bit:

```
void __interrupt my_isr(void){
   ++mt_busy;
   <disable interrupts in device>
   <acknowledge interrupt controller>
   UNMASK_INTS();
   <work>
   MASK_INTS();
   <reenable interrupts in device>
   MTsched_c();
}
```

We must disable the device's interrupts, otherwise an infinite call loop occurs. The *mt_busy* variable locks the kernel and stops other interrupts from triggering a task switch when they exit.

Avoiding Task Switching from Nested Interrupts

MultiTask! provides a simple mechanism for preventing a task switch from happening from a nested interrupt. This mechanism requires that you carefully adhere to the following coding rules in all interrupt service routines.

Rule 1: Always balance `mt_busy`. Remember that ***MTsched_c()*** includes --`mt_busy`.

Rule 2: Every ISR that queues a command should exit to the scheduler (e.g. ***Mtsched_c();***).

Rule 3: On CPUs with multiple priority levels (i960, 68xxx, 68HC16, ColdFire) or on other CPUs when nesting is explicitly allowed, if any ISR of a given priority level uses ***MTqcmd_c()***, then all ISRs at lower priority levels must contain `mt_busy`++; and end with ***MTsched_c();*** (or equivalent code). (A few processors may not allow for ***MTsched_c()***, and then the ISR must be in assembly language, but it may call C code.)

Rules 2 and 3 do not have to be strictly followed; however, if they are not, delays may occur before the command queue is processed when an ISR uses ***Mtqcmd_c()*** to instigate a command. If you want to ensure the most prompt execution of commands queued by an ISR and avoid any previously described problems with task switching from a nested interrupt, then we can distill the above three rules into one rule.

Interrupt Latency

As can be seen from the examples, the *++mt_busy* can sometimes occur early in the interrupt routine and will add to latency. Most systems allow it to be near the end, executing after the application-related code is completed.

Low-level Versus High-level Interrupt Routines

In many systems, it is desirable to have the 'handler' portion of a device driver be independent of the processor. The Stream-I/O serial (UART) drivers are a good example. It is desirable to split the interrupt support into a hardware-dependent portion that handles the processor/chip/board-specific interrupt situation, and a hardware-independent portion that handles only the device (e.g., UART). The first layer may be in extended-C with `__interrupt` used, or in assembler. The second portion is in plain, portable C.

One issue that arises is that there may be two or more UARTs present in a system. Then the ISR handler routine needs to know which UART generated the interrupt. The MultiTask! product keeps a small integer as an argument for each installed handler.

A few processors do not support multiple Interrupt Request Lines (e.g., PowerPC) and so all interrupts call a single ISR. It is then the job of that ISR to read some status information and dispatch the interrupt to the correct handler. Obviously, such a dispatch routine can be enhanced to pass a device number to the handler and, upon return, do the MultiTask! interfacing.

Processors that vector but also pass the vector can be made to use a common wrapper routine. An extra is that non-handler oriented ISRs can still be used. Thus, a mix of high level handlers and low level ISRs is possible.

The x86 vectors in hardware but loses the vector number. On machines like this, each interrupt vector must have a separate wrapper routine. These will typically look like:

```
void __interrupt isr8(void){
  drv0_isr( 0 );    /* COM1 is device[0] */
  ++mt_busy; MTsched_c();
}

void __interrupt isr9(void){
  drv0_isr( 1 );    /* COM2 is device[1] */
  ++mt_busy; MTsched_c();
}
```

As mentioned above, some processors (such as the PowerPC) do not vector directly and a software dispatch layer is required. In such situations, supporting low level jumps may be counter-productive since the dispatch code is more easily written in C and the MultiTask interfacing code (see above) can be handled in the same routine. This may require a three-layer approach on processors that are too hard for C to do the interrupt prolog/epilog. In that case, you will see a single assembly routine that does the prolog/epilog, a second routine in C that does the dispatching to the handlers and then interfaces to MultiTask!, and the device-specific code in various handler routines, also in C.

The Ticker

MultiTask! expects a timer interrupt (periodic [PIT], reload timer channel, compare timer, or real time clock [RTC]), which it uses for time-slicing, timed delays, and timeouts. This interrupt should perform any user-required function and then jump to the ***MTtick*** assembly routine as shown in appendix on *Platform-Specific Information*. Alternatively the routine may call ***MTqcmd_c(***`TIKTOK`***)***. A working version of this ISR is supplied in the file **usrclk.asm** or **usrclk.c**. If the clock interrupt will be from a different source than the supplied routine, you will need to make appropriate modifications.

Example 1 (Assembly Timer Interrupt):

```
my_tick:
   <Acknowledge the hardware>
   JMT MTtick
```

Example 2 (C Timer Interrupt, low level):

```
void __interrupt my_tick(void){
   <Acknowledge the hardware>
   if(clkon) MTqcmd_c(TIKTOK);
   ++mt_busy; MTsched_c();
}
```

Example 3 (C Timer Interrupt, high-level 'handler'):

```
void ticker_isr(unsigned dev){
   <Acknowledge the hardware>
   if( clkon ) MTqcmd_c(TIKTOK);
}
```

Older ports lumped much of the MultiTask! required startup/ shutdown into the ticker's two routines: ***usrclk_init*** and ***usrclk_term***. This included any special system startup, interrupt controller/system startup, ticker startup, and console I/O startup (for **iprintf**). In newer ports, all this support has been split up into three modules and `usrclk_init/term` are now macros that call the individual startups in the correct order.

In such cases, you should have:

ussTickInit	initializes the ticker
ussTickTerm	shuts down the ticker
ussTicks	counts ticks seen (similar to ***get_sys_time***)
ussTimeMS	milliseconds seen (may be from RTC)
ussTick10s	number of ticks in 10 seconds (182 on IBM-PC)
uss_mSecToTicks	converts milliseconds to ticks

If time-slicing, timed delays, periodic events, and timeouts are not needed, then the timer interrupt source is not required. In this case, the ***delay_until()***, ***oneshot()***, and ***period()*** functions cannot be used and timeouts are ineffective. The ***dlytsk()*** function can operate only with an infinite timeout. Task switching among equal-priority tasks can still be forced by calling the ***scdtsk()*** function from each task, which will cause cooperative round-robin type scheduling to occur. Preemption will still occur when a higher-priority task is returned to the run queue.

NOTE: Future versions of MultiTask! may use different functions. Please see the text files provided with your release for current information.

Dynamic Memory Routines – the Heap

It is always nice when everything works, but sometimes errors occur. When you see the heap disappear, or strange crashes occur, it is time to check your memory usage.

The primary heap corruption happens with stack overflow. Most ports now have a C level check at task switch time that checks for a magic bit pattern (`0xDEADBEEF`) that ***runtsk()*** places at the bottom of the stack when it allocates it. If this is no longer correct, then it is very likely that the current task has overflowed its stack. What MT! does when it detects stack corruption is based on the `MTSTACK_TRAP` macro in **depends.h**. Typically, this is an `ASM` statement with a breakpoint instruction in it. You can replace it with a call to your own routine – or with whatever you want.

You can check all active tasks with this code:

```
{ unsigned j; for( j = 1; j <= NUMTSK; ++j){
  TASK_DEF pTask = task_tab[j];
  if( pTask->task_que ){ /* only alive tasks */
        if( (uint32*)task_ptr->task_stk != 0xDEADBEEF ){
            iprintf("! task %u has a trashed stack !\n",j);
        }
  }
}}
```

You can check the entire heap by using ***chkmem()***. This does several checks of the heap, reporting available space and possible errors. By calling this at various times, you can narrow down and locate the problem. A good time to call it is just before and after calls to ***reqmem()*** and ***relmem()***.

Do not forget you have source code. Modify the routines to stop and ***iprintf()*** information when they see an error.

If you are really deep into a heap problem, here is some information on how it is organized. First, when you initialize memory via ***MTmeminit2()***, a 'bumper' is placed at each end. This bumper looks like a tiny block of allocated memory. Since it is allocated, ***relmem()*** will never merge it with a block being freed. Hence, the term 'bumper'. All remaining memory in the chunk being released is organized as a free block. It will contain a size, 2 pointers, free space, and a size. The size is the number of bytes in the entire block forced to a multiple of 4 plus a 2-bit code for 'free'. (The bumpers mentioned above are size = 4 & not free. Typically, this is 0x5.)

If local memory tracking is on (see STCFG in the **makefile**), then all blocks, free or allocated, follow that format. If it is off, allocated blocks do not have the 2 pointers in them, and the overhead shrinks to half.

So, ***chkmem()*** starts by checking each bumper using the pointers in `mem_rootptr[]`. Then it starts just after the lower bumper and looks at the first block size. It uses that to determine where the size field at the top end of the block is and checks that it is the same value. By stepping through all the blocks, it should reach the ending bumper.

The Scheduler

The scheduler is used at the end of a system call or interrupt routine to catch up on any queued commands and do a task switch. It is implemented between several routines:

Routine	**File**
MTqproc()	**mtinit.c**
MTsched_c()	**mtschedc.c**
MTsched:	**mtsched.asm/s**

The scheduler only operates if it will be exiting to a task. Otherwise, it is nested and does nothing. It operates with interrupts enabled, so interrupts can occur while it is processing. The final check, when it decrements *mt_busy*, is performed with interrupts disabled. This prevents an interrupt sneaking something into the command queue while it is testing the exit conditions.

```
if( 1==mt_busy ){
startover:
   process Command Queue
   if( run_queue empty ){
         spin loop or low power mode
         goto startover;
   }
   if( preemption allowed && different task ){
         rsched();  /* CPU specific context sw in asm */
         ++mt_busy;
   }
   MASK_INTS();
   if( cmdprc != cmdadd ){  /* one last check of cmd Q */
         UNMASK_INTS();
         goto startover;
   }
   –mt_busy;
   UNMASK_INTS();
}
```

The code used in interrupt routines increments *mt_busy* and calls ***MTsched_c()***:

```
if( 1==mt_busy ){   /* interrupted a task? */
   UNMASK_INTS();
   MTqproc();
   MASK_INTS();
}else{
   –mt_busy();
}
```

Finally, the routine ***rsched:*** is an assembly language routine that saves the current context, switches the stack pointers, and restores the context from the next task. It also decrements *mt_busy* in case the next task is new (was just set up via ***runtsk()***).

5. Stream I/O

Chapter Contents

5

ANSI C Functions

The *stream I/O* portion of the SuperTask! package implements the following ANSI C stream I/O functions in source form:

fopen	*fread*	*fwrite*	*fgetc*
fgets	*fputc*	*fputs*	*printf*
fprintf	*sprintf*	*vsprintf*	*sscanf*
fgetpos	***fsetpos***	***fseek***	***ftell***
fflush	*fclose*	***mkdir***	***remove***
rewind	***rmdir***	***feof***	***ferror***
clearerr			

The functions shown in the lighter font are present in the SuperTask! package but perform no function for serial port streams and pipes. These additional functions are only meaningful with the disk file manager in the USFiles® package. The function names shown above (***fopen***, etc.) are actually all coded with a prefix of ***mt_,*** so the function names are listed in the *Stream I/O Library* chapter under the names ***mt_fopen***, ***mt_fread***, etc. The reason for these modified names is to allow the use of another I/O library simultaneously with the MT! stream functions. For instance, when running under MS-DOS, you might want to use the standard C library ***fopen*** to open a disk file through DOS, and ***mt_fopen*** to open a serial port through the MT! stream system. Normally each of the function names (***mt_fopen***, etc.) has an alias without the ***mt_*** prefix defined in the include file **mtstdio.h,** so that the two names both refer to the MT! library function. If you `#include` <**stdio**.**h**> before including **mtstdio**.**h**, however, then these aliases are not defined, thus allowing you to link in another library with ***fopen,*** etc. Keep in mind, however, that if you switch off the `defines` in **mtstdio**.**h**, you will be referring to them as ***mt_fopen, mt_fread,*** etc., instead. With the `#defines` switched off, the file descriptor type for MultiTask! I/O becomes `MTFILE` rather than `FILE`. Throughout this document, we may refer to the *streamio* functions both with and without the ***mt_*** prefix; unless specifically stated otherwise, we are referring to the same MT! functions.

MultiTask! Stream I/O is built upon a three-layer code structure that allows additional devices (ports) to be added with the minimum of coding. The top layer is composed of the ANSI C functions already mentioned, most of which are contained in the source file **streamio.c**. The device numbers corresponding to *stdin, stdout,* and *stderr* are also defined in **mtstdio.h** and may be changed freely. The paths to *stdin, stdout,* and *stderr* are not automatically opened. They must be opened explicitly with a call to ***fopen()*** before they are used. If you are using your C compiler library I/O in conjunction with the MultiTask! I/O functions, you must not use ***mt_printf***, or use the *stdin, stdout,* and *stderr* macros with the ***mt_*** functions, since the values defined in **stdio.h** will not be compatible with the **mtstdio.h** values.

The ***fprintf()*** and ***sscanf()*** functions will be generated as integer-only versions (not supporting floats and doubles) unless the label `PF_FLOATS` is defined when you compile these modules.

Whether or not "text" mode stream I/O differs from "binary" mode depends upon the specific driver being used by the stream.

The ***mt_fopen()*** function searches the device table for the *`filename`* requested as the first argument to ***mt_fopen().*** The device table is an array of structures of type `DEVICE`, which is defined in **userio.h**. There is an entry in the device table for each device (port, pipe, disk drive, etc.) that can be opened in your configuration. The **userio.h** file is usually supplied with a number of devices predefined for our test environment. The inclusion of different devices is controlled by conditional `#ifdefs` in **userio**.**h**. There is more information on this in the comments of the makefile supplied for your platform and in comments in **userio**.**h**. The first (perhaps only) serial port entry in the `device_table` will have the device name COM1. This is done on all platforms so there will be a common name for the serial port that can be used by the test programs **siotest** and **tintest**. To add or delete devices or change the default port settings (baud rate, etc.), you will need to edit **userio.h.**

Devices

Serial Ports

For an interrupt-driven device, a task waiting for I/O to complete will automatically sleep in the I/O queue until I/O is complete, and then reawaken. This allows other tasks to use the processor time that would be wasted in polling the port. The wait for I/O sleep works as follows:

- For a read operation, if the input buffer does not contain enough bytes to satisfy the read, then the task will sleep until the requested number of bytes are put into the input buffer by the interrupt service routine for the device. The driver will then wake the task and the transfer of bytes from the interrupt input buffer to the user's buffer will be completed. If the requested number of bytes to be input is larger than the buffer size, the task waiting for input will sleep and wake for each filled buffer, until the total requested input is satisfied.
- For a write operation, the data are placed in the output buffer and the write interrupts for the device are enabled, allowing the output interrupt service routine to immediately commence the output of data. If the output buffer becomes full, then the task will be put to sleep until the buffer is nearly empty again. When the C-level write call that caused the output to occur returns, there may still be a (large) number of bytes in the output buffer that have not actually been transmitted yet. If the task must wait until all the output has actually been transmitted, it may do so by issuing a ***fflush()*** call for the device.

The *sfm* serial drivers can handle one task performing writes and another task performing reads simultaneously on the same port. For proper operation, you must ensure that there is not more than one task requesting reads and one task requesting writes from the same port at any time. You can arbitrate this with a resource if necessary. The same task can perform both reading and writing if desired.

The device is initialized each time the ***mt_fopen()*** function is called. The specific device parameters such as port address, interrupt vector location, ISR address, baud rate, etc., are defined as tables of initialized data in the file **userio.h**. Study this file for more information about how specific drivers supplied with MT! were implemented. You will need to modify this file to change any of these parameters or when you add another device. This will require recompiling **dev_tab.c**.

Pipes

Pipes provide a method of synchronizing the transfer of information from one task to another. As such, they provide an alternative to using a mailbox to pass messages or packets. Pipes transfer a copy of the data from the task writing to the pipe to the task reading the pipe. The **streamio** calls such as ***mt_fread()*** and ***mt_fwrite()*** are used to transfer the data after the pipe is opened by a call to ***mt_fopen().*** Reading or writing to the pipe will cause the calling task to automatically wait as appropriate. In other words, if a task performs an ***mt_fread()*** of a pipe, and no data are currently in the pipe buffer, then the task will wait until the task at the other end of the pipe transfers the number of bytes requested by the read.

SuperTask! pipes were implemented to be as fast as possible. They are implemented entirely by the pipe file manager in **pipefm.c**, with no associated driver. "Text mode" has no significance for pipes, i.e., there is no translation of the end-of-line character if you open the pipe in text mode as there would be for an `sfm` or `pcfm` device.

Whether you use pipes or mailboxes for a given situation will depend on several factors. The main limitation of pipes is that when you open a pipe, you may have only one task reading the pipe and one task writing to it. The other limitation is that there is no timeout available on reading a pipe as there would be with the ***rcvmsg()*** function. If this is suitable for the situation, then pipes bring the advantage of providing faster data transfer than messages, and the passing of a copy of the data rather than just a pointer. Since they pass the data by value, they are equivalent in function to packets transferred through a mailbox. An added advantage of pipes is that

since they use **streamio** functions for the data transfer, the transfer can easily be redirected to another device, such as a serial port (or disk file with USFiles).

To use a pipe, it must be opened by a task by using one of the following *filename* forms for ***mt_fopen()***:

PIPE:	Anonymous pipe
PIPE:mypipe	Named **pipe**
PIPE:/10	Anonymous pipe with buffer size of 10
PIPE:yourpipe/200	Named pipe with buffer size of 200

As shown, pipes can be either anonymous or named. If a named pipe is opened and another request is made to open a pipe with the same name (case is significant), then an error is returned. Each ***mt_fopen()*** of an anonymous pipe will create a new pipe. If the optional "/" followed by an integer number is supplied in the name, then this is used as the size of the buffer for the pipe, which is dynamically allocated from global `COLOR0` memory when the pipe is opened. The buffer will be released when the pipe is closed (by ***mt_fclose()***). If the buffer size is not given, the value of the `#define DEFAULT_PIPE_SIZE` is used.

The first task to read from the pipe becomes locked as the task allowed to read, and any other task requesting a read after this will be denied. The first task writing to the pipe is locked as the writing task in a similar fashion.

The ***find_pipe()*** function is provided so another task can locate the file handle (`MTFILE *`) for an opened named pipe.

Any of the functions ***mt_fread()***, ***mt_fwrite()***, ***mt_fgets()***, ***mt_fputs()***, ***mt_getc()***, and ***mt_putc()***, may be used to communicate through the pipe. The size of the pipe buffer does not affect the speed in a manner you might expect. This is largely due to the implementation, which will transfer multi-byte blocks directly from one task's memory to the other whenever this is possible, instead of going through the buffer. Because of this, a one-byte buffer will in some circumstances give faster throughput than a large buffer. The **pipetest**

program will give timings for various combinations of task priority and buffer size relative to the size of the block being transferred. You should study the output of this test program if you intend to use pipes.

The reading task will wait whenever the pipe buffer is empty and there are still bytes to be read to satisfy the last read request. The writing task will wait whenever the pipe buffer is full and there are remaining bytes to write.

Disk File System

The PC-compatible disk file system is not a part of the SuperTask! package. The USFiles package provides this file system, which will integrate smoothly with SuperTask! as an additional stream device. The USFiles file system can also be used in a single-tasking environment or with an operating system other than SuperTask! USFiles supports any size diskette, hard disk, or RAM disk with an MS-DOS-compatible format, providing complete interchangability of media between your target and any PC. Additional support is available for USFiles for CompactFlash or for CD-ROM media.

Other Devices

The 80x86 version of SuperTask! contains console and keyboard drivers for a standard PC keyboard and a text mode CGA display. The keyboard is completely interrupt driven, allowing a task to wait for keyboard input without using any CPU time in polling as well as giving a true <control>-break interrupt to the application. The display implements non-overlapping screen views (windows) that can each be controlled independently by separate tasks. Complete details on this driver are contained the *Platform-Specific Information* appendix or the **cpunotes.txt** file delivered with the x86 MultiTask! release.

See the next section for information on adding customized drivers for additional devices.

Customizing Stream I/O

Functions for Customizing Stream I/O

These lists show the relationships of particular stream I/O functions to file manager and device driver functions. The relationships are explained in the following sections.

Connections

ANSI Stream I/O	File Manager	Device Driver
mt_fopen	***open***	***init***
mt_fclose	***close***	***term***

Data Transfer

ANSI Stream I/O	File Manager	Device Driver
mt_fread	***read***	***read***
mt_fgetc	***read***	***read***
mt_fgets	***readln***	***read*** (***raw_read***)
mt_fwrite	***write***	***write***
mt_fputc	***write***	***write***
mt_fputs	***writeln***	***write*** (***raw_write***)

Control (Stream)

ANSI Stream I/O	File Manager	Device Driver
	fm_ioctl	***ioctl*** (***diskchange***) (***timestamp***) (***format***)

Control (Outside Connection)

ANSI Stream I/O	File Manager	Device Driver
mt_remove	***_delete***	
mt_rmdir	***fm_ioctl***	
mt_mkdir	***makdir***	

Adding a New File Manager

The MultiTask! stream I/O features are implemented by three layers of functions. The upper layer is composed of the ANSI C functions ***fopen(), fread(), fwrite()***, etc**.** The middle layer is known as the *file manager* and the lower layer as the *device driver*. The standard C level functions such as ***fgetc(),*** for example, call the file manager routines associated with the stream, which in turn call the driver routines. The driver also has interrupt service routines (ISRs) associated with it for interrupt-driven devices.

The maximum number of paths (i.e., ports, or devices) that can be open at any time is set by the `NUMSTREAMS` parameter in **depends**.**h**. Each stream has a structure of type `MTFILE` (alias `FILE`) associated with it, which contains all the control information for the stream. The `MTFILE` structure is defined in **mtio**.**h**. This structure contains a great deal of information, including pointers to other structures necessary for controlling the device. Some of these include buffer pointers, buffer sizes, and counts of characters in the buffers. Some of this information could have been omitted, but it was placed here in this implementation to minimize the code overhead in the ISRs for the device.

Some of the control structures are contained in unions to allow for expansion to new types of devices. The **mtio**.**h** file contains comments indicating where new structures would be added to control new port types or classes of devices.

The file manager, device driver, and ISRs all access the `MTFILE` structure for the stream they are currently operating on. It may be expedient for you to directly access some of the `MTFILE` structure entries also. You might, for instance, wish to test the input buffer

character count of a serial port to test if there are any characters in the input buffer.

Example:

```
if ( ((FILE *)fp->device->devparm.pcs->inbuf_cnt) )
```

We will refer to the type `MTFILE` structure for a stream as its *file descriptor*.

The file descriptor for each stream contains a pointer to the file manager and device driver that are associated with that stream. The file manager is a structure of type `FILEMAN` (defined in **mtio.h**). This structure, sometimes called a 'vector table,' consists of a list of function pointers to the functions that constitute the file manager. The driver is a similar structure of type `DRIVER`, which contains function pointers to the functions that constitute the device driver. There are two `DRIVER` variants. One is for character devices, and the other is for record devices. If your system has several ports of the same characteristics (same type `UART` chip, etc.) they would most likely be using the same driver and file manager. If you have two different types of `UARTs`, they will most likely share the same file manager but have different drivers.

The *`sfm`* file manager routines provided should be usable for any type of serial character I/O device (i.e., `UART`). These routines are all written in C. If another type of device is to be supported, such as a floppy disk with a file system, then new file manager routines would be developed that perform the same functions as the *`sfm`* routines but for the new class of device.

The following lists show the functions in the order in which they are usually used.

The defined file manager functions for any file manager are:

open()
read()
readln()
write()
writeln()
close()
seek()
makdir()
_delete()
fmioctl()

The specific routines that constitute the *sfm* file manager are:

sfm_open()
sfm_read()
sfm_readln()
sfm_write()
sfm_writeln()
sfm_clos()
sfm_seek()
sfm_makdir()
sfm_delete()
sfm_fmioctl()

The ***sfm_seek(), sfm_makdir()*** and ***sfm_delete()*** are dummy routines; they perform no function, since these operations have no meaning for serial ports.

If you are adding only another serial I/O device, you will not need to make a new file manager or modify the *sfm*. In this case you can skip ahead to the discussion of Device Drivers. The earlier lists in *Functions for Customizing Stream I/O* show the relationship between the stream I/O, file manager, and driver functions.

The following function descriptions are in alphabetical order.

File Manager _delete() function

This function is unused unless USFiles is included.

```
int _delete(MTFILE *fp)
```

The file manager ***_delete()*** function removes the file pointed to by *fp* from the file system. In the *sfm* file manager, this is a dummy routine.

File Manager close() function

```
int close(MTFILE *fp)
```

The file manager ***close()*** function calls the driver ***term()*** function to deinitialize the stream. This function returns zero if successful or EOF if an error is detected (i.e., the calling task does not own the device).

File Manager fmioctl() function

```
int fmioctl(MTFILE *fp, int function, va_list ap)
```

The file manager ***fmioctl()*** function provides multiple miscellaneous control functions for the device. The *fp* identifies the path to operate on and the *function* defines the function to be performed. Any additional parameters required are contained in the variable argument list *ap*. The function `IO_FLUSH` is defined for all *sfm* devices to flush the output buffer for the device. Other function codes that are applicable to specific devices are defined in **mtstdio.h**.

File Manager makdir() function

This function is unused unless USFiles is included.

```
int makdir(MTFILE *fp)
```

The file manager ***makdir()*** function converts the open path *fp* into a new directory. This is a dummy routine in the *sfm* file manager.

File Manager open() function

```
MTFILE * open(MTFILE *fp, const char *filename)
```

The ***open()*** function of a file manager is passed the file descriptor pointer *fp* and the filename. In the *sfm* file manager, the *filename* is not used and is ignored. The ***open()*** function fills in most of the ile descriptor tructure and associated substructures with initial values, and calls the driver_init routine to initialize the device. The ***open()*** function returns a pointer to the file descriptor structure if it is successful, otherwise it returns a NULL pointer.

File Manager read() function

```
int read(MTFILE *fp, char *buf, int bytes)
```

The file manager ***read()*** function reads the number of bytes specified by *bytes* from the stream specified by *fp* into the buffer pointed to by *buf*. ***Sfm_read*** calls the driver read routine for each byte to be transferred from thISR input buffer to the user's buffer. The control structure wake_cnt and wake_mode parameters are set as necessary to cause the task to sleep until the necessary data bytes are in the input buffer if they are not immediately available. The ***read()*** function returns the number of bytes actually read. If this is fewer than the *bytes* requested, then some sort of error occurred.

File Manager readln() function

```
int readln(MTFILE *fp, char *buf, int bytes)
```

The file manager ***readln()*** (read line) routine reads at most the number of bytes specified by *bytes* from the stream specified by *fp* into the buffer pointed to by *buf*. The read will terminate early if the EOL_CHAR is read. n all other respects, this call is the same as the ***read()*** function.

File Manager seek() function

This function is unused unless USFiles is included.

```
int seek(MTFILE *fp, uint32 s)
```

The file manager ***seek()*** function repositions the file pointer for *fp* so that the next access will be at *pos* bytes from the beginning of the file. This is a dummy function that returns an unimplemented error for the *sfm* file manager.

File Manager write() function

```
int write(MTFILE *fp, char *buf, int bytes)
```

The file manager ***write()*** function writes the number of bytes specified by *bytes* taken from the memory buffer pointed to by *buf* and writes these to the stream specified by *fp.* The ***write()*** routine accomplishes this by sendi each byte to the driver ***write()*** routine for the device. If the device is interrupt driven, the driver ***write()*** routine will place the characters into the ISR output buffer and the ISR will do the actual transmitting of the data. The file manager ***write()*** routine will set up the necessary parameters so that the task will sleep if the output buffer is full and there are more data to be written. The actual number of bytes written is returned by this function. This will be zero if an error occurs.

File Manager writeln() function

```
int writeln(MTFILE *fp, char *buf, int bytes)
```

The file manager ***writeln()*** function is identical to the ***write()*** function except that the write will terminate before bytes have been transmitted if an `EOL_CHAR` is encountered in the output stream. (The ***writeln()*** termines after the transmission of the `EOL_CHAR`.)

Adding a New Device Driver

The group of functions necessary to implement a *device driver* depends upon what *file manager* will be controlling it. All device drivers controlled by the *sfm* file manager consist of the following routines:

init()
read()
write()
ioctl()
term()

For a specific instance of a driver, these routines will be given the above-mentioned names with a unique prefix prepended to them to designate the driver; e.g., ***drv0_init().*** The ***drv0_read()*** function is in **sfm.c**, and the other functions will be in the **driver0.c** (or similarly named) file.

The exact function performed by these routines depends somewhat upon what file manager will be calling them. The division of responsibilities between the file manager and the device driver may be altered if a new file manager is developed. Since we expect all serial-type devices to use the *sfm* file manager, we will explain the exact function of the driver as expected by the *sfm* file manager.

Device Driver init() function

```
int init(FILE *fp)
```

The file descriptor pointed to by *fp* will be initialized with all available information before the driver ***init()*** function is called. This function can be written in assembly language or C for the particular device. This function will perform all hardware initialization necessary to use the device, including:

- Initializing device registers (baud rate, interrupt controller setup if necessary, etc.)
- Installing the device interrupt vector(s), unless this is already in ROM or in some other way already initialized
- Enabling the device receiver interrupt (unless the device will be polled, i.e., not interrupt-driven)
- Copying *fp->mode* to *fp->init_flag* when the initialization is complete

NOTE: It may be necessary to physically mask interrupts for some of the above operations.

The return value is not used by *sfm* and therefore does not need to be implemented in an assembly *init* routine.

Device Driver ioctl() function

```
int ioctl(FILE *fp, int function, va_list ap)
```

The driver ***ioctl()*** function is a catch-all for miscellaneous control functions for the device. Function code numbers less than 100 are reserved for definition by US Software. User-added function codes should begin at 100. The ***ioctl()*** function returns 0 if successful or EOF (-1) as an error indication.

Function code: ***IO_FLUSH***

The ***ioctl() IO_FLUSH*** function flushes the output buffer for the stream, i.e., the function waits until all characters in the output buffer have been transmitted by the transmit ISR.

Function code: ***IO_BAUD***

```
int ioctl(FILE *fp, IO_BAUD, baud_code)
```

The ***ioctl()*** change baud function uses the device-specific *baud_code* to reprogram the device baud rate for the stream. For some drivers the *baud_code* is the specific value to stuff into the device registers to set the new baud rate. On most of the newer ports, it is the desired BAUD divided by 100.

Other function codes are listed in **mtstdio.h**. Examine supplied drivers for examples of their use.

Device Driver read() function

```
byte read(FILE *fp)
```

The driver ***read()*** routine returns the next byte available from the stream specified by *fp*. The routine ***drv0_read()*** can be used as the driver ***read()*** routine for all interrupt-driven *sfm* serial devices. In this case, the device ISR does the actual reading in response to interrupts, and places the input characters into the stream input buffer. The ***drv0_read()*** routine returns the next available character from the input buffer. If the input buffer is empty when ***drv0_read()*** is called, it will cause the calling task to wait in the I/O queue until characters are available. The ***read()*** ISR will wake up the task when the input buffer contains the appropriate number of characters that cause ***drv0_read()*** to resume its transfer of data. Each serial device has a parameter structure that is accessible through a pointer in the file descriptor pointed to by *fp*. The parameter structure contains pointers to the read circular buffers for the port and other needed information such as the task requesting I/O and the conditions to wake the task for.

If a device is to be polled rather than interrupt-driven, the driver ***read()*** function for that device can be a simple polling routine that returns the next byte of data, rather than the ***drv0_read()*** routine. In this case, there would be no ISR for the device, and a task will never sleep waiting for input from the device but rather waste time in the polling loop until the input is available.

Device Driver term() function

```
void term(FILE *fp)
```

The driver ***term()*** function flushes the device output buffer and then deinitializes the device, disabling any interrupt generation by the device. When this is complete, it writes zero to *fp>init_flag* and *fp>owner_slot*.

Device Driver write() function

```
void write(FILE *fp, byte c)
```

The driver ***write()*** function for a polled device would simply transmit the character *c* to the device indicated by stream *fp*.

For an interrupt-driven device, the ***write()*** function will place character *c* into the stream output buffer and enable the device transmitter interrupt. The transmit ISR will remove the character from the output buffer and actually transmit it.

JumpTable

The above device driver functions are gathered in a jump table used by the appropriate file manager. This table can be found in **userio.h** for older ports, and at the end of the particular **driver?.c** file for newer ports.

Device Driver Interrupt Service Routines

Interrupt-driven serial devices that perform both read and write logically have both a read and a write ISR. In many cases, only one interrupt vector services both interrupts. If this is the case, the beginning of the ISR will determine if the interrupt was caused by received data and will vector to the read interrupt service, or, if the interrupt was due to the transmit buffer becoming empty, it will vector to the transmit interrupt code.

On some processors, several devices will be vectored to the same interrupt service location. In this case, a poll of each device needs to be done to determine the source of the interrupt, and then a jump made to the appropriate service routine.

If the version of MT! you are using was supplied with an interrupt-driven device driver, you will be able to reuse parts of the ISR code in the ISRs to support a new device.

The first step on entering any ISR will be to save any registers that will be used in the ISR code. Each device ISR will then need to locate the address of the file descriptor, i.e., the (`MTFILE *`) value for that device. Each device has a `DEVICE` structure entry in device table (`device_tab`). The *file descriptor pointer* (`MTFILE *`) for the device can be obtained from here at `device_tab[n].fp`, where *n* is the index into the device table for the port causing the interrupt. This value is initialized by ***sfm_open()*** when the port is opened.

The ISR will need to initialize a pointer to the file descriptor for the port to locate information such as buffer locations needed for servicing the interrupt. Once the proper file descriptor has been located, code common to all ports of the type can be used for the remaining processing, since all details specific to the port are contained in the file descriptor.

If the device supports both read and write and will generate a separate interrupt for each, the (`MTFILE *`) value will be the same for both the read and write. If there is only one interrupt vector for both the read and write, now is the time to determine if the interrupt is a read or write interrupt (after locating the (`MTFILE *`) value) and then branch to either the read or write service path.

Device Read ISR

The read interrupt for the device will indicate that a character has been received and is ready to be read from the device hardware.

The read interrupt service routine for a driver will save any registers that will be used by the routine and then locate the address of the file descriptor for the device being serviced. This file descriptor address would be a type (*MTFILE* *) in C. This value should be saved into a local variable. We will refer to this value as *fp* and use the C notation for accessing elements in this structure (e.g., *fp->init_flag)* in order to make clear what operations we require the ISR to perform. The ISR can be written in assembly for maximum efficiency, or in C.

The device table entry can be located at *fp->device*, and from this a pointer to the device parameter table that contains most of the information needed for port control can be obtained as:

```
parmp = fp->device->devparm.pcs
```

In most processors supported, interrupts will be disabled when the ISR is entered. If this is not the case, then it will be necessary to disable them before any parameters in the file descriptor are updated.

The input byte *c* is read from the device and placed in the input buffer at (**parmp->inbuf_addp++ = c*). If after incrementing the input buffer add pointer is past the end of the buffer, then it is wrapped back to the beginning.

```
if(parmp->inbuf_addp == parmp->inbuf_end)
      parmp->inbuf_addp = parmp->inbuf_beg;
```

If the input buffer is overrun, then set bit 0 of *fp->error_code*. Bits 1..7 of *fp->error_code* can be used to log any other I/O errors for the device.

Increment *parmp->inbuf_cnt*, which is the count of the number of characters currently in the input buffer. If the WAKE_READ bit is set in *parmp->wake_mode* and the number of characters currently in the input buffer is greater than or equal to *parmp->wake_cnt*, then wake up the task at the slot specified by *parmp->read_owner* by

using the ***MTqcmd_c()*** call to queue a ***wketsk()*** command and clear the `WAKE_READ` and `WAKE_LINE` bits in *`parmp->wake_mode`*.

If the `WAKE_LINE` bit is set in *`parmp->wake_mode`* and the last character *`c`* received was equal to *`fp->eol_char`*, then wake up the task at the slot specified by *`parmp->read_owner`* by using the ***MTqcmd_c()*** call to queue a ***wketsk()*** command and clear the `WAKE_READ` and `WAKE_LINE` bits in *`parmp->wake_mode`*.

If the ISR sends a ***wketsk()*** command via the ***MTqcmd_c()*** call, it will then restore any saved registers and jump to ***MTsched()*** (or call ***MTsched_c()***) rather than performing a return from interrupt.

If the ISR did not send a ***wketsk()*** command, then it will restore any saved registers and do a normal return from interrupt.

Device Write ISR

The write interrupt for the device will indicate that the device is ready to accept another character for transmission.

The write ISR for a device will save any registers that the routine will use, and then locate the file descriptor address value *`fp`* for the device, and the parameter structure *`parmp`* as in the read ISR.

If *`parmp->outbuf_cnt`* (the number of characters in the output buffer waiting to be transmitted) equals zero, then further transmit interrupts from the device are disabled and the ISR is exited.

If *`parmp->outbuf_cnt`* <> 0, then the character pointed to by *`parmp->outbuf_remp`* is transmitted, *`fp->outbuf_remp`* is incremented and wrapped if necessary:

```
parmp->outbuf_remp++;
if( parmp->outbuf_remp == parmp->outbuf_end )
      parmp->outbuf_remp = parmp->outbuf_beg;
```

The count if characters in the output buffer is decremented: (`parmp->outbuf_cnt–;`).

If the resulting value of *parmp->outbuf_cnt* is greater than *parmp->outbuf_min*, the ISR is exited (registers restored, and return from interrupt executed).

If the new value of *parmp->outbuf_cnt* is less than or equal to *parmp->outbuf_min* and the bit WAKE_OBE is set in *parmp->wake_mode*, then a ***wketsk()*** command is sent to the task at slot *parmp->write_owner* as described for the read ISR. (The ***wketsk()*** command is queued with the ***MTqcmd_c()*** call, registers are restored, and the ISR exits by jumping to ***MTsched*** or calling ***MTsched_c()***).

Supplied Serial Drivers (driver0.c)

Each platform contains a driver either for an internal CPU serial port, if applicable, or a commonly used UART. You can study this source code that follows the form just described. This can usually serve as code for any other serial UART device, with minor modifications.

We will provide drivers for the three major UARTs (8250, Zilog 85xx, and 26xx) if requested. These have been done for various ports, but will still require tweaking to work with your board.

Changing the I/O Device Table

The *device table* is an array of structures of type DEVICE (defined in **mtio.h**). The table is named device_tab and is defined in the file **userio.h**. There must be a device table entry for each device that can be opened.

The ***mt_fopen()*** function will search the name entries in the device table to find a name that matches the filename given as the first argument of the ***mt_fopen()*** call. If the name is not found, the device cannot be opened.

The DEVICE structures contain essential information about the device, such as port address, interrupt number, and initialization

parameters in addition to the device name. These other items of information are initialized data in the device table.

All of this essential initialized data to define a device is in **userio.h**. To add another device (port, etc.), an entry must be added to the device table.

The device table (in **userio.h**) is delivered preconfigured for supported devices. All deliveries will have support for at least a pipe named "`PIPE`," and most will have a serial port named "`COM1.`" Some platforms also have entries for `ramdisk, floppies, hard disk,` CD-ROM, and `console/keyboard`. These supplied entries are conditionally included in the table when the following labels are defined during compilation.

Table 5-1: Device Table Codes

Code	Label	Device Included in Device Table
`s`	SIO	Serial port devices
`i`	PIPE	Pipes
`cd`	CDIO	CD-ROM (with USFiles only)
`c`	CIO	Console/keyboard (PC only)
`p`	PCIO	Hard disk/floppy (with USFiles only)
`r`	RIO	RAM disk (with USFiles only)

6. Stream I/O Library

Chapter Contents

I/O Functions by Category

ANSI Stream I/O Functions

mt_fclose	***mt_fflush***	***mt_fgetc***	***mt_fgets***
mt_fopen	***mt_fprintf***	***mt_fputc***	***mt_fputs***
mt_fread	***mt_fwrite***	***mt_printf***	***mt_sprintf***
mt_vsprintf	***sscanf***		

ANSI Stream I/O Functions in USFiles

mt_clearerr	***mt_feof***	***mt_fseek***	***mt_fgetpos***
mt_fsetpos	***mt_ftell***	***mt_rename***	***mt_rewind***

Additional I/O Functions

find_pipe	***mt_ioctl***	***mt_remove***	***mt_rmdir***

I/O Function Descriptions

find_pipe

Finds handle for a named pipe.

```
MTFILE *find_pipe(char *filename);
```

filename pathname to the named pipe

This function attempts to return the file-descriptor pointer to the named pipe specified by *filename*. The pipe must be open, or it will not be found.

Return Value

`MTFILE *` pointer to the pipe

`NULL` pipe not found (not open)

Example

```
MTFILE *pfp;

  pfp = find_pipe("PIPE:pipe1"); /* get pipe handle */
  if( !pfp)
        error("pipe1 not opened");
```

mt_clearerr

Clears the end of file and error indicators.

```
void mt_clearerr(MTFILE *stream);
```

stream pointer to the stream file descriptor

The end of file and error indicators associated with *stream* are cleared.

See also: ***rewind, feof, ferror***

Example

```
FILE *fp;
  /* open for read/write */
  fp = mt_fopen(DEVICE_0, “r+b”);
  mt_clearerr(fp);
```

mt_fclose

Closes an open path to a stream.

```
int mt_fclose(MTFILE *stream);
```

stream	pointer to the stream file descriptor object

The ***mt_fclose()*** function returns zero if the *stream* was successfully closed, or EOF if any errors were detected. For ***mt_fclose()*** to successfully complete, the stream must be open and accessible by the task making the ***mt_fclose()*** call. The stream output buffer will be flushed before the device is closed. The device interrupts are disabled when the device is closed, and any tasks waiting for the device I/O to complete will be reactivated.

Return Value

0	file successfully closed
EOF	error (file not open, or not in possession of task making the call)

Example

```
MTFILE *fp;

  fp = mt_fopen("COM1", "r+b"); /* open for read/write */
  {     /* processing */ }

  if( mt_fclose(fp) )
      { /* file close error */ }
```

NOTE: If the task in possession of an open stream or streams dies, or is killed (by ***klltsk()***), all streams in the possession of that task are closed.

mt_feof

Tests for end of file condition.

```
int mt_feof(MTFILE *stream);
```

stream pointer to the stream file descriptor object

The ***mt_feof()*** function returns non-zero if the *stream* is at end of file. Once the EOF flag is set, it will only be cleared by a ***rewind()*** or ***clearerr()*** function, or by closing the stream.

NOTE: This function is implemented as a macro in **mtstdio.h.**

Return Value

0	file is not at end
!0	file is positioned at end

Example

```
MTFILE *fp;
int i;

  /* open for read/write */
  fp = mt_fopen("a:file1", "r+b");
  while( !mt_feof(fp) ){
      i = mt_fgetc(fp);
      /* etc. */
```

mt_ferror

Returns the file error indicator.

```
int mt_ferror(MTFILE *stream);
```

stream pointer to the stream file descriptor object

The ***mt_ferror()*** function returns non-zero if the error indicator is set for the stream. The error indicator will be cleared by a ***rewind()*** or ***clearerr()*** function, or by closing the stream.

NOTE: ANSI C does not specify under what conditions the error indicator for the stream is set. In the current implementation, only the driver level ever sets the error indicator. You should generally rely on the return status of each function to determine errors, and the value of *errno*. Note also that *errno* is not cleared by any ANSI C function. Once it is set non-zero, it is up to you to clear it.

Return Value

0 no error

!0 some kind of error

Example

```
MTFILE *fp;
int i;

  /* open for read/write */
  fp = mt_fopen(DEVICE_0, “r+b”);
  while( !mt_feof(fp) ){
      i = mt_fgetc(fp);
      if( mt_ferror(fp) )
          report_error(“File error occurred”);
      /* etc. */
```

mt_fflush

Flushes the output buffer of a stream.

```
int mt_fflush(MTFILE *stream);
```

stream	pointer to the stream file descriptor object

The ***mt_fflush()*** function will cause the calling task to wait until any remaining data in the *stream* output buffer has been transmitted to the port or file. If the argument is NULL, then all open streams are flushed.

Return Value

0	file successfully flushed
EOF	error (file not open, or not accessible from the task making the call)

Example

```
MTFILE *fp;

  if( mt_fflush(fp) )
      { /* file flush error */ }
```

mt_fgetc

Gets a character from a stream.

```
int mt_fgetc(MTFILE *stream);
```

stream pointer to the stream file descriptor

The ***mt_fgetc()*** function obtains the next character as an unsigned `char` converted to an `int`, from the input stream pointed to by *stream*.

Return Value

`character` the next character from the stream

`EOF` error (stream not opened or not in possession of calling task)

Example

```
MTFILE *fp;               /* open stream pointer */
int     c;

  c = mt_fgetc(fp);       /* get character */
  if( c == EOF )
      { /* stream error */ }
```

mt_fgetpos

Gets stream's current position.

```
int mt_fgetpos(MTFILE *stream, fpos_t *position);
```

stream pointer to I/O stream

position position returned by function

The ***mt_fgetpos()*** function fills *position* with a value representing the current position of the file pointer for *stream*. This is usually the byte number from the beginning of the file. In the case of a file open in text mode, this may not be the same as the actual number of bytes you have read from the file. The *position* returned by ***mt_fgetpos()*** should be used as an argument to ***mt_fsetpos()*** to reposition a file to a former location.

See also: ***mt_fsetpos***

Return Value

0	success
<>0	*errno* set to EBADFP or EUNSUP

Example

```
MTFILE *fp;          /* open stream pointer */
fpos_t offset;       /* place to remember position */
int status;          /* for error value */

status = mt_fgetpos(fp, &offset);
if( status ){ /* error code here */ }
        /* read/write work with the file */
status = mt_fsetpos(fp,&offset);
if( status ){ /* error code here */ }
        /* we are now back at the same position */
```

mt_fgets

Gets a string from a stream.

```
char *mt_fgets (char *s, int n, MTFILE *stream);
```

s	pointer to character array of at least size *n*
n	maximum number of characters to read plus one (for the null string terminator)
stream	pointer to the stream file descriptor

The ***mt_fgets()*** function reads at most one less than the number of characters specified by *n* from the stream pointed to by *stream* into the array pointed to by *s*. No additional characters are read after the new-line character (which is retained). A null character is written immediately after the last character read into the array.

NOTE: The new-line character is defined in **userio.h** as EOL_CHAR and is not necessarily the same as "\n" produced by your C compiler. The default value of EOL_CHAR is the ASCII carriage return ("\r") for sfm and pipe devices, and new line ("\n") for pcfm devices.

See also: ***mt_fputs***

Return Value

s	operation successful
NULL	error (stream not open, or not in our possession)

Example

```
MTFILE *fp;                /* open stream pointer */
char buf[80];
  if( mt_fgets(buf, 80, fp) ){
     /* we have string */
  }else{
     /* error processing */
  }
```

mt_fopen

Opens a path to a stream.

```
MTFILE *mt_fopen(const char *name, const char *mode);
```

name	pathname to `device:[file]`
mode	type of access permitted

The ***mt_fopen()*** function opens a path to *name* and returns a pointer to the `MTFILE` structure controlling the stream. The device component part of *name* must appear in the device table (*device_tab*). The additional *name* components, if any, must conform to the rules for the type of device opened. If the operation fails, a null pointer is returned. The *mode* string specifies the type of access requested as follows:

"r"	open text mode for reading
"w"	create text mode for writing
"a"	append (open/create for write at EOF)
"rb"	open binary mode for reading
"wb"	create or truncate for binary write
"ab"	append binary (open/create for write at EOF)
"r+"	open for update (read and write)
"w+"	truncate or create for update
"a+"	append (update at EOF)
"r+b"	open binary mode for update
"w+b"	truncate or create for binary update
"a+b"	append; open/create for binary update at EOF
"d"	open directory (USFiles only)

NOTE: Not all modes defined are meaningful for every device; i.e., for `sfm`-controlled serial streams, truncating or appending to a file is the same as just opening it for write. Text mode translation is implemented for `pcfm` (USFiles) streams and `sfm` serial streams, but not for pipes.

See also: ***mt_fclose***

Return Value

`MTFILE *`	File descriptor pointer to be used as a "handle" argument for all subsequent I/O calls for the device.
`NULL`	Unable to open the device, possibly because the device name was invalid, or the device is already in the possession of another task. The value of the global *`errno`* may contain additional error status. See **mtio.h** for the error codes returned in *`errno`*.

Example

```
MTFILE *fp;

   fp = mt_fopen("COM1", "r+b"); /* open for r/w binary mode */
   if( !fp )
       { /* error opening device */ }
```

mt_fprintf

Sends formatted output to a stream.

```
int mt_fprintf(MTFILE *stream, const char *format, ...);
```

stream	output stream file descriptor pointer
format	format specification string
...	arguments to be formatted for output

The ***mt_fprintf()*** function writes output to the stream pointed to by *stream*, under control of the string pointed to by *format* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the *format*, the behavior is undefined. If the *format* is exhausted while arguments remain, the excess arguments are ignored. The ***mt_fprintf*** function returns when the end of the *format* string is encountered.

The *format* must be a multibyte character sequence composed of zero or more directives. A directive is one or more white-space characters, ordinary characters (not %) that are copied unchanged to the output stream, or a conversion specification. A conversion specification is introduced by the character %, and has the following format:

```
%[flags][width][precision][mod]type
```

flags	-	left-justify result
	+	always prefix with + or -
	space	prefix with a blank if non-negative
	#	alternate form conversion
width	*n*	prints at least *n* characters, pad with spaces
	0n	prints at least *n* characters, pad with zeros
	*	next argument that must be type int; is consumed from the *args* list and used as the width specifier

precision		
	(default)	=1 for `d,i,o,u,x,X` =6 for `e,E,f`
	`.0`	no decimal point for `e,E,f`
	`.n`	*n* decimal places or characters are printed
mod	`h`	short `int` for types: `d,i,o,u,x`
	`l`	long `int` for types: `d,i,o,u,x` double for types: `e,f,g`
	`L`	same as `l`
type	`c`	`int` converted to unsigned `char` printed
	`d`	signed decimal `int`
	`e`	signed exponential
	`f`	signed floating point
	`g`	same as `e` or `f` based on value and precision
	`i`	signed decimal `int`
	`n`	argument is a pointer to `int` into which is written number of `char`s written to stream so far
	`o`	octal unsigned `int`
	`p`	pointer
	`s`	string
	`u`	decimal unsigned `int`
	`x`	hexidecimal unsigned `int` (a..f)
	`X`	hexidecimal unsigned `int` (A..F)

Return Value

`+`*n*	number of characters written
`EOF`	output error (stream not open or not accessible)

Example

```
MTFILE *fp;             /* open stream pointer */
int count,i,j;
double x,y;
count = mt_fprintf (fp,"i = %d, (%04X hex), x=%e\r\n",i,i,x);
```

mt_fputc

Writes a character to a stream.

```
int mt_fputc(int c, MTFILE *stream);
```

c	character to be output
stream	pointer to the stream file descriptor object

The ***mt_fputc()*** function writes the character specified by *c* (converted to an unsigned `char`) to the output stream pointed to by *stream*. The ***mt_fputc()*** function returns the `char`acter written unless an error occurs, in which case it returns `EOF`.

See also: ***mt_fgetc***

Return Value

`character`	next character from the stream
`EOF`	error (stream not opened or not in possession of calling task)

Example

```
MTFILE *fp;               /* open stream pointer */
int     c;

  if( mt_fputc(c,fp) == EOF )
      { /* stream error */ }
```

mt_fputs

Writes a string to a stream.

```
int mt_fputs(const char *s, MTFILE *stream);
```

s	pointer to the string to write
stream	pointer to the stream file descriptor

The ***mt_fputs()*** function writes the string pointed to by *s* to the stream pointed to by *stream*. The terminating null of *s* is not written. The number of characters written is returned unless a write error occurs, in which case EOF is returned. (NOTE: The ANSI C standard specifies only that a non-negative value is returned in the normal case.)

See also: ***mt_fgets***

Return Value

count	number of characters written
EOF	error (stream not opened for write or not in possession of calling task)

Example

```
MTFILE *fp;                /* open stream pointer */

  if( mt_fputs(“Hello there”,fp) == EOF )
      { /* write error handling */ }
```

mt_fread

Reads bytes from a stream.

```
int mt_fread(void *ptr, int size, int nmemb, MTFILE
             *stream);
```

ptr	pointer to the buffer to receive data
size	size in bytes of each element
nmemb	number of elements
stream	stream object pointer

The ***mt_fread()*** function attempts to read *nmemb* elements of *size* bytes into the array pointed to by *ptr*, from *stream*. The actual number of elements read is returned. Note that the number of elements returned will be equal to *nmemb* unless the EOF is reached or some error occurs.

See also: ***mt_fwrite***

Return Value

+*n*	number of elements actually read
EOF	end of file reached or some other error

Example

```
MTFILE *fp;               /* open stream pointer */
char buf[80];

  if( mt_fread(buf, 1, 80, fp) != 80 )
      { /* incomplete read */ }
```

mt_fseek

Repositions file pointer.

```
int mt_fseek(MTFILE *stream, long offset, int location);
```

stream	pointer to I/O stream
offset	number of bytes to offset from location to determine new file pointer position
location	file position from which to add offset SEEK_SET (0) - beginning of file SEEK_CUR (1) - current file pointer position SEEK_END (2) - end of file

The ***mt_fseek*** function repositions the file pointer for *stream* by *offset* bytes from *location*. If the *stream* is text mode, offset should be 0 or the value returned by ***mt_ftell().*** The value in *location* should be SEEK_SET for beginning of file, SEEK_CUR for current file pointer position, or SEEK_END for end of file.

See also: ***mt_ftell***

Return Value

0	file pointer successfully repositioned
<>0	reposition error (stream not open or not the owner)

Example

```
MTFILE *fp;                /* open stream pointer */
int status;
/* Note: second arg below is 30"ell" */
   status = mt_fseek(fp,30l,SEEK_SET);
```

mt_fsetpos

Sets stream's current position (byte offset from beginning of file).

```
long int mt_fsetpos(MTFILE *stream, const fpos_t *pos);
```

stream	pointer to I/O stream
pos	new position to set

The ***mt_fsetpos()*** function sets the file pointer associate with *stream* to the new position *pos*. The new position is the value obtained by a previous call to ***mt_fgetpos()*** on that stream. The reason for the existence of ***fgetpos()*** and ***fsetpos()*** (in addition to ***fseek***) is that if you want to position to a file in text mode, you cannot necessarily find a position by counting the characters you have written out because text mode translation may change that number, in which case you can only use ***fgetpos()*** to find a current position and then return there later with ***fsetpos()***.

See also: ***mt_fgetpos***

Return Value

0	success
non-zero	failure, with the global variable *errno* set to a non-zero error code

Example

```
MTFILE *fp;          /* open stream pointer */
fpos_t offset;       /* place to remember position */
int status;          /* for error value */

status = mt_fgetpos(fp, &offset);
if( status ){ /* error code here */ }
/* read/write work with the file */
status = mt_fsetpos(fp,&offset);
if( status ){ /* error code here */ }
/* we are now back at the same position */
```

mt_ftell

Gets current file position.

```
long int mt_ftell(MTFILE *stream)
```

stream pointer to I/O stream

The ***mt_ftell()*** function returns the value of the file pointer for *stream*. The file pointer contains a value that specifies the current position of the file as the byte offset from the beginning of the file.

See also: ***mt_fseek***

Return Value

offset	value of file pointer on success
-1	*errno* set positive on failure

Example

```
MTFILE *fp;                /* open stream pointer */
long offset;
int i;
double x;

  offset = mt_ftell(fp);
  status = mt_fseek(fp,offset,SEEK_SET);
```

mt_fwrite

Writes to a stream.

```
int mt_fwrite(const void *ptr, int size, int nmemb,
              MTFILE *stream);
```

ptr	pointer to the data to write
size	size of each data item
nmemb	number of data items
stream	pointer to the stream file descriptor

The ***mt_fwrite()*** function writes, from the array pointed to by *ptr*, up to *nmemb* elements of *size* bytes each, to *stream*. The number of elements actually written is returned, which will be less than *nmemb* only if an error occurred. If the stream is not open or not accessible to the calling task, EOF will be returned.

See also: ***mt_fread***

Return Value

count	number of items written
EOF	error (stream not opened for write or otherwise not accessible by the calling task)

Example

```
MTFILE *fp;               /* open stream pointer */
int count;
int data[10];

  count = mt_fwrite(data, sizeof(int), 10, fp);
  if( count < 10 )
      { /* write error occurred */ }
```

mt_mkdir

Creates a new directory.

```
int mt_mkdir(const char *path)
```

path complete pathname of the directory to create

The ***mt_mkdir()*** function creates a new directory from the given pathname *path*.

Return Value

0	success
EOF	error, and global variable *errno* set to a non-zero error code (*errno* codes are defined in **mtio.h**)

Example

```
int status

  status = mt_mkdir("a:\\thisdir/thatdir/newdir");
  if( status )
      { /* mkdir error occurred */ }
```

NOTE: US Files accepts either '\' or '/' characters as name separators interchangeably.

mt_printf

Formats output to `stdout` stream.

```
int mt_printf(const char *format, ...);
```

format	format specification string
...	arguments to be formatted for output

The ***mt_printf()*** function writes output to the `stdout` stream, under control of the string pointed to by *format* that specifies how subsequent arguments are converted for output. The ***mt_printf()*** call behaves exactly like an ***mt_fprintf()*** call with `stdout` specified as the stream, and indeed it is implemented as this. See ***mt_fprintf()*** for further information on the *format* specification. The definition of `stdout` is in the file **mtstdio.h** and may be modified by the user to be any device.

NOTE: `stdout` is not automatically opened.

Return Value

+*n*	number of characters written
EOF	output error (stream not open or not the owner)

Example

```
FILE *fp;               /* open stream pointer */
int count;
int i,j;
double x,y;

   count = mt_printf("i = %d, (%04X hex), x=%e\r\n",i,i,x);
```

mt_remove

Deletes a file.

```
int mt_remove(const char *pathname);
```

pathname complete pathname to the file

The ***mt_remove()*** function deletes a file specified by *pathname*. A complete pathname, including the device name, must be specified.

Return Value

0	success
EOF	error condition, with the global variable *errno* set to the specific error code

Example

```
if( mt_remove("a:\\subdir\\thisfile.txt") )
    printf("errno = %d\n", errno);
```

mt_rename

Renames (or moves) a file or subdirectory.

```
int mt_rename(const char *oldname, const char *newname);
```

oldname	pathname to an existing file
newname	new pathname to give file

The ***mt_rename()*** function changes the name of the file *oldname* to *newname*. A complete pathname must be given for both, which must be on the same device (drive). Subdirectories can be renamed. The *newname* does not need to be in the same directory as *oldname*. The effect in this case is that of moving the file to the new directory (and possibly renaming it in the process).

Return Value

0	success
EOF	error condition, with the global variable *errno* set to the specific error code

Example

```
if( mt_rename("a:\\file1.txt","a:\\file2.txt") )
    printf("errno = %d\n", errno);
```

mt_rmdir

Removes a subdirectory.

```
int mt_rmdir(const char *pathname);
```

pathname the complete pathname of the directory to delete

The ***mt_rmdir()*** function removes the directory specified by *pathname* from the file system. The directory must be empty or an error is returned. An attempt to remove the root directory returns an error.

See also: ***mkdir()***

Return Value

0	success
EOF	error, and global variable *errno* set to a non-zero error code

Example

```
int status

  status = mt_rmdir("a:\\thisdir/thatdir/newdir");
  if( status )
      { /* rmdir error occurred */ }
```

NOTE: US Files accepts either '\' or '/' characters as name separators interchangeably.

mt_sprintf

Formats output to a string.

```
int mt_sprintf(char *s, const char *format, ...);
```

s	pointer to string to receive output
format	format specification string
...	arguments to be formatted for output

The ***mt_sprintf()*** function writes output to the string pointed to by *s*, under control of the string pointed to by *format* that specifies how subsequent arguments are converted for output. The ***mt_sprintf()*** call behaves exactly like ***mt_fprintf()*** except that the output is written to the string *s* rather than a stream.

See also: ***mt_fprintf()*** for further information on the *format* specification

Return Value

+*n*	number of characters written
EOF	output error (stream not open or not accessible)

Example

```
MTFILE *fp;               /* open stream pointer */
int count;
int i,j;
double x,y;
char line[100];

count = mt_sprintf
   (line,"i = %d,  (%04X hex), x=%e\r\n",i,i,x);
```

mt_sscanf

Formats conversion from a string.

```
int mt_sscanf(const char *s, const char *format, ...);
```

s	pointer to string containing input characters
format	format specification string
...	pointers to objects to receive input items

The ***mt_sscanf()*** function reads input from the string pointed to by *s*, under control of the string pointed to by *format* that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the *format*, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are ignored.

The format shall be a multibyte character sequence composed of zero or more directives. A directive is: One or more white-space characters, an ordinary character (neither a % nor a white-space), or a conversion specification. A conversion specification is introduced by the character % and has the following format:

```
%[flags][width][mod]type
```

flags	*	suppresses assignment of next field
width	n	maximum number of characters that will be read
mod	h	short `int` for types: `d,i,o,u,x`
	l	long `int` for types: `d,i,o,u,x` double for types: `e,f,g`
	L	same as `l`

type	c	single character
	d	signed decimal `int`
	e	signed exponential
	f	signed floating point
	g	same as `e` or `f` based on value and precision
	i	signed decimal, octal, or hex `int` (e.g. 123, 0123, 0x123)
	[abc]	matches characters in set or...
	[^ab]	matches characters NOT in set
	n	`int` to receive count of `char`s consumed so far
	p	pointer
	s	string

Return Value

+*n*	number of input items assigned {0..*n*}
EOF	failure

Example

```
MTFILE *fp;                    /* open stream pointer */
char buf[80];
int count, arg[4];

      mt_fgets(buf, 80, fp);  /* read string */
      count = mt_sscanf
      (buf,"%d %d %d %d",&arg[0], &arg[1], &arg[2], &arg[3]);
```

mt_vsprintf

Formats output to string.

```
#include <stdarg.h>

int mt_vsprintf(char *s, const char *format, va_list arg);
```

s	pointer to string to receive output
format	format specification string
arg	list of arguments to be formatted for output

The ***mt_vsprintf()*** function is equivalent to ***mt_sprintf(),*** with the variable argument list replaced by *arg*, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The ***mt_vsprintf()*** function returns the number of characters written in the array, not counting the terminating null character.

Return Value

+*n*	number of characters written
EOF	output error (stream not open or not accessible)

Example

```
MTFILE *fp;                /* open stream pointer */
int count;
int i;
double x;
void *args[3];

  args[0] = &i;
  args[1] = &i;
  args[2] = &x;

count = mt_vsprintf
  (line,"i = %d,  (%04X hex), x=%e\r\n",&args[0]);
```

timed_getc

Gets a character from a stream with timeout.

```
#include "timedin.h"

int timed_getc(MTFILE *fp, uint timeout);
```

fp	pointer to the stream file descriptor object
timeout	maximum clock ticks to wait for char

The ***timed_getc()*** function obtains the next character as an unsigned char converted to an int from the input stream pointed to by *fp*. If *timeout* clock ticks elapse with no input available, a value of E_TIMED_OUT is returned instead of a character from the stream.

This input function with timeout works with any MultiTask! interrupt-driven serial (sfm) stream driver. To use this function, #include **timedin.h** in your source file for the function prototype. (This function does not work with pcfm or pipefm paths.)

See also: ***timed_read, timed_readln***

Return Value

character	next character from the stream
EOF	error (stream not opened or not accessible by calling task)
E_TIMED_OUT	*timeout* expired before character received

Example

```
MTFILE *fp;               /* open stream pointer */
int     c;

  c = timed_getc(fp, 5);  /* get character */
  if( c == E_TIMED_OUT)
      { /* timeout occurred */ }
```

timed_read

Reads a block with timeout.

```
#include "timedin.h"

int timed_read(MTFILE *fp, char *buf, int bytes, int
               timeout);
```

fp	pointer to the stream file descriptor
buf	pointer to `char` buffer
bytes	number of bytes to transfer
timeout	maximum clock ticks to wait for `char`

The ***timed_read()*** function reads the number of bytes specified by *bytes* from the stream specified by *fp* into the buffer pointed to by *buf*. If the input data is not already available in the interrupt input buffer, then the task will sleep until the specified number of bytes is received or *timeout* clock ticks elapse with no new input. The ***timed_read()*** function returns the number of bytes actually read. If this is less than the *bytes* requested, then some sort of error occurred. A return value of zero could result from the stream not being open, or the calling task not having access permission, or a *timeout* with no input available. If a *timeout* occurs, then the return value could be any value less than *bytes*.

This input function with *timeout* works with any MultiTask! interrupt-driven serial (`sfm`) stream driver. It will not work with and should not be used with `pcfm` (disk) or `pipefm` (pipe) paths. To use this function, be sure to `#include` **timedin.h** in your source file for the function prototype.

See also: ***timed_getc, timed_readln***

Return Value

n	number of bytes actually returned

Example

```
MTFILE *fp;               /* open stream pointer */
char buf[40];
int count;

  count = timed_read(fp,buf,40,10);
  if( count < 40 )
      /* timeout */
```

timed_readln

Reads a line with timeout.

```
#include "timedin.h"

int timed_readln(FILE *fp, char *buf, int bytes, int
                 timeout);
```

fp	pointer to the stream file descriptor
buf	pointer to `char` buffer
bytes	number of bytes to transfer
timeout	maximum clock ticks to wait for `char`

The ***timed_readln()*** function reads at most the number of bytes specified by *bytes* from the stream specified by *fp* into the buffer pointed to by *buf*. The ***timed_readln()*** function will terminate early if the `EOL_CHAR` is read. In all other respects, this call is the same as the ***timed_read()*** function.

This input function with timeout works with any MultiTask! interrupt-driven serial (`sfm`) stream driver. It will not work with and should not be used with `pcfm` (disk) or `pipefm` (pipe) paths. To use this function, be sure to `#include` **timedin.h** in your source file for the function prototype.

See also: ***timed_getc, timed_read***

Return Value

n	number of bytes actually returned

Example

```
MTFILE *fp;               /* open stream pointer */
char buf[81];
int count;

  count = timed_readln(fp,buf,80,10);
  buf[count] = 0;         /* add terminating null */
  if( count < 80 )
      /* timeout */
```

A. Platform-Specific Information

Chapter Contents

ARM/StrongARM Platform

Evaluation Platforms

ARM7 PIE

Processor: ARM7*
Compiler: ARM SDT*
Debugger: EmbeddedICE*

EBSA-285

Processor: StrongARM*
Compiler: ARM SDT
Debugger: Angel Debug Monitor*

The Makefile

Be sure to set *PTH* (and *IPTH*, if necessary) before compiling. `TRG_ID = 0` selects ARM7 PIE, and `TRG_ID = 3` selects EBSA-285.

Support for StrongARM EBSA-285 Evaluation Board

The support for running on the StrongARM EBSA-285 Evaluation Board is selected by setting the macro `TRG_ID = 3` in the **makefile** before compiling.

Test Setup:

The EBSA-285 board is plugged into the System PCI slot of the Digital Semiconductor PCI Development Backplane.

A PC serial port is connected to the serial port on the EBSA-285 with a null modem cable. The port is set up for 9600 baud, 8 data-bits, no parity, and 1 stop-bit.

SuperTask! has been configured to run under the control of the Angel debug monitor, which interacts with a symbolic debugger running on a host PC running Windows*. The debug monitor uses the FIQ interrupt but leaves the IRQ free for the application to use. SuperTask! uses the IRQ for its timer interrupt.

SuperTask! uses SWI interrupts, but we need to retain the SWI interrupt handling provided by the debug monitor so that SuperTask! can chain the debug monitor SWI interrupt handler, enabling it to execute if SuperTask! does not recognize the SWI.

SuperTask! uses the debug monitor character I/O capability for output of diagnostic messages, because the EBSA-285 has only one serial port. Since the debug monitor uses FIQ for all its interrupt processing, including serial I/O, a large amount of character output can have an effect on overall system performance and IRQ interrupt response.

The debugger can be invoked from the DOS command line like this:

```
armsd -ADP -Port h=0,s=2 -LINEspeed 9600 my_program.aif
```

Support for ARM7 PIE Board

The support for running on the ARM7 PIE* board is selected by setting the macro `TRG_ID = 0` in the **makefile** before compiling.

Test setup:

EmbeddedICE connected to ARM7 PIE board `JTAG` port.

PC (or SUN) serial port connected to EmbeddedICE.

ARM7 PIE card serial port connected to 9600 baud terminal.

The `scc2691` interrupt on the PIE card must be vectored to the IRQ rather than the FIQ for all MT! code to run. This must be done by moving LK5 on the board from the B setting to the A setting. (Unfortunately this jumper is soldered to the board so you will need a soldering pencil to change it. There appear to be through-holes in the board connected to each leg of the jumper settings, so alternately you could use 30-gauge wire-wrap wire and solder a switch to these. This would allow quick switching between the two settings if you still need to run code requiring the B setting also.)

(Note: The monitor EPROM u16 must be removed from the PIE card for running with EmbeddedICE attached. This requires the use of a plcc extraction tool.)

It is essential that *`$semihosting_enabled`* variable of `armsd` be set to 0 (off) to run all test programs provided. If this is not done, the EmbeddedICE will intercept the SWI interrupt.

We have found the following command sequence to work:

```
C:\ajunk>armsd -serial -li

A.R.M. Source-level Debugger vsn 4.45b (ARM Toolkit v2.0) [Oct 4 1995]

EmbeddedICE v1.03, 512kB RAM, ROM CRC OK, Little Endian

ARMSD: 0x00000018 = 0xE1A00000

ARMSD: 0x0000001C = 0xE14FD000

ARMSD: 0x00000020 = 0xE38DD0C0

ARMSD: 0x00000024 = 0xE169F00D

ARMSD: 0x00000028 = 0xE25EF004

ARMSD: $semihosting_enabled=0

ARMSD: load coretest.aif

ARMSD: go
```

NOTE: The first five lines starting with `ARMSD:` are from the **armsd.ini** file, which must be in the current directory to be found. The last three lines are typed in by you. All of the test programs (**coretest**, **mtbench**, **pipetest**, **siotest**, **tintest**) begin by writing a line of text out the serial port, so after you enter **go** you should see something on the attached terminal.

The serial port is programmed to 9600 baud, 8 data-bits, no parity, 1 stop-bit.

Special Issues

ARM Operating Modes

All tasks are run in USER32 mode. The MT! kernel itself is an extension of the task making the service call, and is therefor also in USER32 mode. We have provided an SWI service call that can be used for masking interrupts or changing into supervisor (SVC32) mode temporarily when necessary.

Supervisor mode might be necessary for accessing regions of memory that are accessible only in this mode. For instance, on the ARM7 PIE board, the serial port is mapped into supervisor memory space, so it can only be accessed when the CPU is in supervisor mode (any mode except USER32).

The scheduler entry (***MTsched***) can be entered from IRQ32 mode, i.e.; for an IRQ handler, but not from FIQ32 mode. The fast interrupt mode (FIQ32) is reserved for use by interrupt handlers that will not be entering the scheduler. This is with keeping in the overall intent of the FIQ32 interrupt mode, that it should provide the quickest possible interrupt response.

The MT! OS never masks the FIQ32 interrupt.

The IRQ32 interrupt is masked very briefly in some instances.

Interrupt Considerations

IRQ Interrupt Handling

In order to provide the best possible performance when task switching from the IRQ32 interrupt, the `IRQ_Handler` saves all registers immediately on the task's user-mode stack. The handler then vectors to the ***usrclk()*** or another individual service routine.

The specific service routine can either return from the interrupt with a ***mov pc,lr*** instruction, or branch to ***Mtsched***, which is the scheduler entry point.

The ***usrclk()*** handler branches to ***MTtick***, which is optimized code to queue the `TIKTOK` command (clock processing) and then continues to ***MTsched,*** which processes the command queue, thus acting on the queued command immediately. This allows the easiest recoding of the ***usrclk()*** function itself to support a different interrupt source as the system clock. An alternate ***usrclk()*** function need only acknowledge the interrupt if necessary and then branch to ***MTtick***.

FIQ Handling

The FIQ interrupt is not used in any of our test programs. The user can devise their own handler for this interrupt with the one condition that the FIQ interrupt handler must not branch to ***MTsched***.

SWI Handling

A software interrupt service routine is provided which duplicates the "demon" ***SWI_WriteC()***, ***SWI_ReadC()***, and ***SWI_Exit*** functions. The exit function only hangs in a loop where you can break execution if using the EmbeddedICE.

Another function, ***change_cpsr,*** is also provided. It is used to implement several macros essential to the MT! implementation. These are:

MASK_INTS()	disables the IRQ interrupt.
UNMASK_INTS()	enables the IRQ interrupt.
SAVE_AND_MASK_INTS()	disables the IRQ interrupt saving previous state.
RESTORE_INT_MASK()	restores IRQ interrupt mask to the state when the SAVE_AND_MASK_INTS() macro was called.
ENTER_SUPERVISOR_MODE()	switches to SVC32 mode.
LEAVE_SUPERVISOR_MODE()	switches to USER32 mode.

M*Core

Evaluation Platforms

PowerStrike MMC2001

Processor:	MMC2001
Compiler:	Diab Data 4.2b
Debugger:	Single Step* 7.41

Red Cap 56651

Processor:	56651
Compiler:	Diab Data 4.2b
Debugger:	Single Step 7.3

The Makefile

Modify *PTH* to point to your tool chain, and *CVER* to specify the tool chain version number.

Setting `TRG_ID = 1` selects the PowerStrike MMC2001, and `TRG_ID = 2` uses the Red Cap 56651.

Special Issues

CPU Notes

Red Cap Issues

Red Cap has an early rev part with some problems. The `divide` instruction is not implemented for M*Core rev. 1.0. The breakpoint is shaky, which SDS had to work around. We ran into this issue in a couple of places. In particular, Console I/O to SDS/SS is very shaky — use the external UART support in **getput20**.

Also, the Red Cap chip is designed with the JTAG sharing pins with the internal UART. Since SDS/SS can only work via the JTAG, we could not try the UART. Thus both the Stream-I/O driver and the internal UART option in **getput20** have not been tested.

Stack Size

There is a brief (1 sentence) note in the C calling convention guide that the stack should be modified by multiples of 8 bytes. The current interrupt and task-switch code does not follow that convention. At some point, Motorola may finalize the M*Core Reference Manual and provide some Assembly Programming Guides.

Software Breakpoints in Single Step

When running under an HP-Probe and Single Step, the system occasionally gets a ‘software breakpoint’ and loses it. This shows up as `pc: ???unknown`. SDS is aware of the problem.

Interrupt Considerations

Interrupt Vectoring

The M*Core 2001 does not vector interrupts. So we have a vectoring facility in **intrpt1.c** that allows the drivers to 'register' some non-interrupt subroutines. The vectoring code handles the jump to the scheduler too. Thus we only support 'handler' style interrupts.

Interrupts and the Diab Data Compiler

The early Diab Data compiler (4.1a) available for this port does not handle the M*Core interrupts properly. The compiler assumes all exceptions are 'fast' style, which is invalid for traps. The next version (4.2b) supports command-line selection between 'normal' and 'fast' styles. This does not help much since the **intrpt.c** module has both styles in a single module.

Diab Data 4.2b is out with some fixes for the interrupt problems. The Red Cap support has been partially upgraded to support both interrupt levels, but some work is still needed.

The interrupt interfacing has been localized in the **intrpt.c** module. The other modules (ticker and UART) register their interrupts via a `setvect` type call. ***MTtick:*** has not been provided since the ticker is in C. ***MTsched:*** needs modification to comply with the Diab Data limitations.

If you use assembly language interrupts and do not want to do a lot of work, change ***MTsched:*** in **MTsched.s** to use `rfi` as the exit instruction.

See also: For further information, see the files **cpunotes.txt** and **compnote.txt** in the SuperTask! installation directory.

MIPS Platform

The Makefile

You might need to set the *PTH* macro to point to your tool chain directory. The *TRG_ID* macro will specify the proper board as follows:

`TRG_ID = 1`	is for 79R385
`TRG_ID = 2`	is for 79S461/P400i
`TRG_ID = 3`	selects IDT v7.13 for 79S465
`TRG_ID = 4`	is for NEC 4373

Interrupt Considerations

R3000 Support

Interrupt Handling

The interrupt handling code used by MultiTask! resides in the file **usrclk.s** (or **usrclk.S** for GNU tools).

With the EPI tools, the debugger (ROMS) have already implemented handling of the general exception vector, so hardware interrupt handlers have to change back to the debugger's handlers for other exceptions. The ***usrclk_init()*** function attaches to the general exception vector and sets up the structures used by EPI so it can chain to the previous handler for exceptions we don't want to handle.

The version of **usrclk.s** provided for the GNU tools simply takes over the general exception vector. (You must add code to handle the exceptions other than hardware interrupts.)

In both cases, the general exception will enter the ***int_dispatch()*** function also in **usrclk.s.**

The ***int_dispatch*** function will create a stack frame and save most CPU registers, then read the `C0_CAUSE` register for the source of the interrupt. If the `EXC CODE = 0`, then the IP field is used to vector to one of 8 interrupt sources. The vectors for these are placed in the vector table `xint_dispatch_table` beforehand. The small lookup table `offtab` is used to prioritize the interrupts so than IP7 is the highest and IP0 is the lowest priority. If the vector table entry for the particular interrupt being asserted is zero, then ***int_dispatch*** chains to the next exception handler. If it is not zero, a call to the function at the vector address is made.

The interrupt handler called can be in C or assembly. If the handler is in assembly, you must take care to preserve the registers that are normally preserved by C, if these are to be used. See the following list showing the stack frame and registers saved by ***xint_dispatch***.

If a jump to ***MTsched*** is made from the interrupt handler instead of returning, then the stack must be as set up by ***int_dispatch*** before this jump is made.

NOTE: The ***usrclk*** function provided essentially does this through the helper function ***Mttick,*** which queues the clock tick command.

MultiTask! Version 6.xx MIPS (R3000)

The following list shows the register stack frame saved by ***int_dispatch()*** before calling the interrupt handler (— indicates nothing saved at this location).

Location: $29+offset	**Register Saved**
sp+31*4+16	$31 (ra)
sp+30*4+16	—
sp+29*4+16	—
sp+28*4+16	$28 (gp)
sp+27*4+16	C0_SR

Location: $29+offset	Register Saved
sp+26*4+16	C0_EPC
sp+25*4+16	$25 (t9)
sp+24*4+16	$24 (t8)
sp+23*4+16	—
sp+22*4+16	—
sp+21*4+16	—
sp+20*4+16	—
sp+19*4+16	—
sp+18*4+16	—
sp+17*4+16	—
sp+16*4+16	—
sp+15*4+16	$15 (t7)
sp+14*4+16	$14 (t6)
sp+13*4+16	$13 (t5)
sp+12*4+16	$12 (t4)
sp+11*4+16	$11 (t3)
sp+10*4+16	$10 (t2)
sp+9*4+16	$9 (t1)
sp+8*4+16	$8 (t0)
sp+7*4+16	$7 (a3)
sp+6*4+16	$6 (a2)
sp+5*4+16	$5 (a1)
sp+4*4+16	$4 (a0)

Location: $29+offset	Register Saved
sp+3*4+16	$3 (v1)
sp+2*4+16	$2 (v0)
sp+1*4+16	$1 (at)
sp+0*4+16	— (used by isr0() to save ra)
sp+12	reserved for called function use
sp+8	reserved for called function use
sp+4	reserved for called function use
sp+0	reserved for called function use

Registers `s0..s8` (`r16..r23,r30`) are not saved. These will be automatically saved by any C-level interrupt handler the user has installed which is called by ***int_dispatch().*** If ***int_dispatch()*** is to call an assembly-level handler, you must take care to preserve these registers. The four words at `sp+0..sp+12` allow you to call a C handler function taking up to four word-sized arguments. The C compiler generates code to store the arguments in these locations.

R4650 Support

For the IDT79R4650, support files are provided only for the GNU tool set (UNIX workstation). The release diskette is in PC (DOS) format and contains an ***install*** shell script which will copy the files to the workstation using the ***dos2unix*** command. Please examine the script comments for the usage syntax.

If you use the script to install these files, you will get support file versions for both the R3000 and R4650. The **makefile** is for the R3000, and **makefile.4** is for the R4650. If you are using the R4650, you might want to rename **makefile.4** to **makefile**, and discard the R3000 version.

Interrupt Handling

The interrupt handling for the 4650 is similar to what was described above for the R3000, with the following differences:

- The alternate vector for the external interrupts is used. This is done by setting the IV bit in the `Cause` register in the ***usrclk_init*** function.
- The system clock interrupt is implemented with the internal `COUNT` and `COMPARE` registers (which use `xint5`).
- The ***int_dispatch*** routine saves 64-bit registers rather than 32-bit. (See the stack frame diagram below.)

MultiTask! Version 6.xx MIPS (R4650)

The following list shows the register stack frame saved by ***int_dispatch()*** before calling interrupt handler (— indicates nothing saved at this location).

Location: $29+offset	**Register Saved**
sp+31*8+32	$31 (ra)
sp+30*8+32	—
sp+29*8+32	—
sp+28*8+32	$28 (gp)
sp+27*8+32	C0_SR
sp+26*8+32	C0_EPC
sp+25*8+32	$25 (t9)
sp+24*8+32	$24 (t8)
sp+23*8+32	—
sp+22*8+32	—

Location: $29+offset	Register Saved
sp+21*8+32	—
sp+20*8+32	—
sp+19*8+32	—
sp+18*8+32	—
sp+17*8+32	—
sp+16*8+32	—
sp+15*8+32	$15 (t7)
sp+14*8+32	$14 (t6)
sp+13*8+32	$13 (t5)
sp+12*8+32	$12 (t4)
sp+11*8+32	$11 (t3)
sp+10*8+32	$10 (t2)
sp+9*8+32	$9 (t1)
sp+8*8+32	$8 (t0)
sp+7*8+32	$7 (a3)
sp+6*8+32	$6 (a2)
sp+5*8+32	$5 (a1)
sp+4*8+32	$4 (a0)
sp+3*8+32	$3 (v1)
sp+2*8+32	$2 (v0)
sp+1*8+32	$1 (at)
sp+0*8+32	— (used by isr0() to save ra)
sp+24	reserved for called function use

Location: $29+offset	**Register Saved**
sp+16	reserved for called function use
sp+8	reserved for called function use
sp+0	reserved for called function use

Registers `s0..s8` (`r16..r23,r30`) are not saved. These will be automatically saved by any C-level interrupt handler the user has installed which is called by ***int_dispatch()***. If ***int_dispatch()*** is to call an assembly-level handler, you must take care to preserve these registers. The four double words at `sp+0..sp+24` allow you to call a C handler function taking up to four double-word-sized (64-bit) arguments. The C compiler generates code to store the arguments in these locations.

NEC 4373

The chip has 32 integer/address registers with 64 bits each. There is also a set of 32 floating point registers with 64 bits each. The context switch saves and restores most of the integer registers. If you use floats, then you must add similar code to save and restore the float registers.

Interrupts

The EPI compiler provides the hooks to the interrupt vector. For simplicity, we hook to EPI's chain and then implement a table of six entries for each of the size interrupt lines. Again, for simplicity, we only support high level 'handler' type routines.

Ticker

We use the processor's Count/Compare registers. It uses the standard interrupt and is installed as a handler.

Console I/O and Stream I/O Driver

The onboard UART is an old, slow part, and the onboard wait-state generator either cannot support it, or else EPI used the incorrect settings. The chip unfortunately links the two channels' interrupts together. Because EPI uses the interrupt, the Stream I/O driver cannot use it.

PowerPC Platform

Evaluation Platforms

Version 6.28 of SuperTask! provides support for the MPC8xx PowerPC* parts. Support for the IBM 403 has been suspended. Later releases will add support for other PowerPC family members, and/or additional peripheral device support.

Motorola MPC821

Processor:	MPC821/860	
Compilers:	Diab Data 4.2b	Green Hills 1.8.9
Debuggers:	Single Step	MULTI*

The Makefile

The *PTH* macro in the **makefile** will need to be edited to equal the pathname where your compiler is installed, and *CVER* should specify the tool chain version. The *TRG_ID* macro in the **makefile** selects the target board (processor) as follows:

`TRG_ID = 8` selects the MPC821/860

`TRG_ID = 1` selects the PPC403GA (support discontinued)

The *DBG_ID* macro must also be set to specify the proper debugger:

`DBG_ID = 0` selects MPC8Bug

`DBG_ID = 1` selects SDS/Single Step (when using Diab Data)

`DBG_ID = 2` selects Green Hills/MULTI (not using Diab Data)

No other change should be required to build the test programs and library for either of these two environments.

MPC8xx Test Environment

We have been using the 821-ADS board and the 860-FAD board for testing. For these boards, we use the MPC8bug via an ADI interface card in an IBM PC clone. For console I/O, we use a terminal program on an IBM PC.

We have also done testing with SDS Single Step* using the Background Debug Mode and with Green Hills MULTI* via OCD.

Building and Running Test Programs

Build the test programs by invoking Opus ***make***:

```
omake
```

This will build all of the OS modules, put them in a library, and finally build the test program modules.

We have supplied our initialization file **init.cfg** for use with the MPC8Bug board interface program. It will configure the debug registers to allow external interrupts to be handled.

All test programs use the PIT periodic interval timer in the MPC8xx for the system timer interrupt.

The **siotest** and **tintest** programs (serial port driver) use the SMC1 UART interrupts.

To run a program:

1. Start MPC8BUG.
   (Note: The **MPC821.CFG** and **MPC860.CFG** files need to be present so they will be automatically loaded by MPC8BUG, otherwise the board will not be configured properly.)
2. To load our initialization file, which enables the program to get control of external interrupts, type:

   ```
   ex init.cfg
   ```

3. To load the program code and symbols, type:

```
load coretest.elf
```

4. To execute the program, type:

```
go
```

Console I/O routines are provided for a UART, SDS/SS, and MULTI.

Special Issues

Cache

SuperTask!'s interrupt code saves and restores the cache state around an interrupt and disables it for interrupt processing. I/O is mapped into cached memory space and the chip select logic does not override it. You will want to enable cache at the start of each task.

Interrupt Considerations

MPC8xx Interrupt Handling

The files **Intrpt8a.s** and **Intrpt8.c** provide new code to support the SIU and the CPM. Decrementer support is not provided. Jumping to low-level assembly routines is also not supported. All ISRs are treated as handlers and are called as plain C routines.

IBM PPC403GA Test environment

NOTE: Support for this environment is currently suspended.

For the 403GA (IBM Oak board), target-specific code is:

File	Contents
usrclk1.s	timer `int` handler and `init`
getput1.c	serial port polled routines, for both internal and external ports; you set which one with an `#if` in the file
ireq403.s	external interrupt decode and vectoring
sup403.s	SPFR access routines
driver1.c	interrupt driver for internal serial port
driver2.c	interrupt driver for Oak board 16550 UART
uart403.s	ISR stubs for above 2 drivers

The appropriate routines above will be selected in the **makefile** simply by setting the *`TRG`* macro in the **makefile**. You might want to make some alteration to the interrupt code in respect to how vectors are initialized for your final environment. We have set things up assuming you are downloading code to RAM (hopefully with a debugger) in order to test. Our `usrclk` and serial driver `init` routines initialize the vector to the interrupt handler when they are called. We implement vector tables for the individual interrupt sources in **ireqhand.s** or **ireq403.s**. Our entry point for the external interrupt in these files will save all appropriate registers on the stack (156 bytes of stack space is used for this). This is more than needs to be saved in some cases, but all of this will need to be saved if you want to call MT! (using ***MTqcmd_c***) and task switch from the interrupt (by exiting to ***MTsched***). Saving the registers immediately on entry saves you doing it later, and optimizes the process for task switching from an interrupt.

On the 403GA board, we are running the SDS Singlestep debugger using a target monitor through the serial port. The original monitor from SDS uses the internal serial port (S1), at 19200 baud. This has the drawback that you can't run/debug any code making use of this port (which you will probably want to do). We have retargeted the monitor to use the other (16550) port at 115200 baud, and can supply you with an image of either. (It's up to you to get it into flash on the board.)

NOTE: When you use the internal serial port, the CTS line on this port must be tied high (or connect to an active signal on the other end). This is pin 6 on the S1 connector (our documentation page 3-8 for Oak board incorrectly shows this as pin 4; the schematics are correct). There is no way (that I could find) to do 3-wire-only communication with this port without tying CTS/DSR active.

SH Platform

Evaluation Platforms

Version 6.28 of SuperTask! for the SH-series processors contains specific board support for the SH-1 LCEVB from Hitachi with either a SH7032 or SH7034, the SH7604 (SH-2 board), and the SH7708 (SH-3 board).

SH7032

Processor:	SH1	
Compilers:	Hitachi	GNU
Debuggers:	None	None

SH7604

Processor:	SH2
Compilers:	Hitachi
Debuggers:	None

SH7708

Processor:	SH3
Compilers:	Hitachi
Debuggers:	None

The Makefile

To compile SuperTask! for any of these three platforms with the Hitachi compiler, we provide both a UNIX makefile (called **makefile**) and a DOS makefile (called **makefile.dos**). In both cases, you will need to set *PTH* to point to the directory containing the tool chain.

The *TARGET* macro will need to be specified as follows:
`TARGET = 1` is for the SH7032 board
`TARGET = 2` is for the SH7604 board
`TARGET = 3` is for the SH7708 board.

Notes on SH1 Support

The macro `TARGET = 1` in the **makefile** will select the target-specific support files for the SH1 test environment. Other settings will be used for the SH-2 and SH-3 targets. Examine the **makefile** for specific details. The `$(TARGET)` becomes a filename suffix to select the appropriate support files.

The ***usrclk_init*** function in **usrclk1.s** reprograms the wait states for areas 1, 2, and 6 to one wait state (i.e., 2 clock cycle access). Change this if it is not appropriate for your target design.

The ***MTtick*** function expects `r0` through `r3` to be saved on the stack when it is entered.

For efficiency, there are several entry points for the ***MTsched*** function (in **mtsched.s**):

MTsched expects the stack to be empty except for the `'sr'` and `'pc'` register values that were saved by the exception.

MTsched2 is a shortcut that you can jump to with `r0` through `r3` saved on the stack.

MTsched_full expects all registers saved on the stack.

Using these can eliminate redundant saves and restores from interrupt processing.

There is also a "return from ISR" function, ***_isr_exit,*** that can be jumped to for restoring all registers. Examine the **mtsched.s** file for the stacking order if you intend to use these. Also examine **isr0a.s,** which is the **driver0.c** interrupt serial driver entry point for the serial ports. This makes use of ***MTsched_full***, and this stacking order. You can copy this when implementing other ISRs that will interact with the OS.

The ***MTqcmd_c()*** function has been written in assembly for this implementation. This provides greater efficiency. This function now resides in **mtsched.s,** so the **mtqcmd_c.c** file will not be used for this implementation even though it is included in the release.

Notes on SH2 Support

The SH2 support files are selected by setting the *TARGET* macro in the **makefile** to 2. Most of the SH2 support files are the same as for the SH1. Differences are in the serial port initialization and timer interrupt initialization. The test programs for this environment are configured to be downloaded into RAM on a DENSAN SH7604 VME board. The difference between this and running from ROM is that we assume the interrupt vector table to be in RAM, and during program initialization dynamically alter interrupt vectors for the devices we are using. When moving to ROM, these vectors should probably reside in ROM. In this case you will modify the vector initialization code in the functions ***usrclk_init()*** and ***drv0_init()***.

Notes on SH3 Support

The SH3 test environment consisted of a DENSAN SH7700 board connected to a UNIX (Solaris 2.4) workstation via serial and ethernet connections. All test code is configured for this environment to be downloaded to RAM using the on-board monitor. For testing code in ROM (or ROM image with an emulator), you should make minor changes to how interrupt vectors are initialized. Our existing support code dealing with interrupts is in the files **ireqhand.s**, **usrclk3.c**, and **driver3.c**. Our test environment assumes that there are existing exception handlers in place in RAM where they can be overwritten by the program.

During program startup, code in the **usrclk_init()** function will replace the interrupt handler (at address `vbr+0x600`) with our new handler. This is accomplished by copying the code from ***new_intsrv***

in **ireqhand.s** to the handler location in RAM. The ***new_intsrv*** handler uses a new vector table, `new_vectortab`, which is defined in **usrclk3.c** to vector to the appropriate code for handling the individual interrupts.

SH3 new_intsrv interrupt handler

The ***new_intsrv*** handler function that is supplied has been optimized to allow the user's interrupt handling functions to efficiently use MultiTask! service calls and initiate task switching. The user's interrupt handlers for individual interrupt sources are written in C (or assembler if preferred) as ordinary functions. Do not use the `#pragma interrupt` for these functions. The code in ***new_intsrv*** will save all CPU registers (except bank1 registers which may be modified by the interrupt). When the user's handler function returns to the calling point in ***new_intsrv***, these registers are restored and execution returns to the context before the interrupt happened.

The `new_vectortab` (defined in **usrclk3.c**) is an array of the following type:

```
typedef struct inttab_entry {
        void (*isr)(void);      /* handler address */
        uint32  newsr;          /* new sr value */
} INTTAB_ENTRY;
```

Each "vector" entry is composed of two words, the `isr` element is the handler function address, and the `newsr` element is the value that will be placed into the `SR` (status register) before the call is made to the handler function.

Since MultiTask! is currently designed to run only in the "privileged mode", the `MD` bit must be set in the `newsr` value. In addition, the `BL` and `RB` bits will normally be cleared, and the `I3..I0` bits will be set to the interrupt level of the associated interrupt. (It is possible to set the `I3..I0` bits to any level). A sample vector entry in `new_vectortab` taken from **usrclk3.c** is shown below:

```
  usrclk,SRMD|(T0level<<4),/* Vector 32 (TICPI2)*/
```

SRMD is defined as the value of the status register word with only MD bit position set. T0level, in this case, is the priority level of this interrupt source, and ***usrclk*** is the function name of the interrupt handler function, which is an ordinary function (no #pragma interrupt). The T bit position (bit 0) of the newsr word should be zero. When the T bit of the newsr is 1, then the interrupt is handled in a different manner. All interrupt handler functions that have the newsr T bit (bit 0) equal to zero should be functions returning an int value, rather than void. A non-zero handler return value designates that the return path is to be through the MultiTask! scheduler entry ***MTsched***. A zero value returned by the handler instructs that the ***new_intsrv*** code is to return directly to the previous context without going through ***MTsched***.

In addition to this, the ***new_intsrv*** code takes care of incrementing the *mt_busy* variable before the user's handler function is called, and also decrementing it, when appropriate, upon return. The user should therefore not increment *mt_busy* in their interrupt handler functions that are entered through the new_vectortab by the ***new_intsrv*** code. This applies only to the SH3. The interrupt handlers for the SH1 and SH2 should follow the normal rules for this described earlier in Chapter 2 of this manual. In all cases it is best to study the ***usrclk*** and serial driver handlers (**driver?.c**) for examples of the proper handling.

Raw-mode interrupt handling

When the newsr value for the handler in new_vectortab has the T bit (bit-0) set, then the vectoring is done in a different manner (which we will call raw-mode). In this case, only the SSR and SPC registers are saved on the stack before vectoring to the user's interrupt handler. This mode should not be used by an interrupt handler that needs to queue MultiTask! system commands with the ***MTqcmd*** or ***MTqcmd_c*** functions, or that needs to exit through the scheduler (***MTsched***). This is suitable for handlers that need to use only a few registers, and which do not need to use MultiTask! services. These interrupt sources should also have a high enough priority to ensure they will not be interrupted by any normal mode handlers which might use MultiTask! services. By setting the RB bit in the newsr

value for a raw handler, it can use the bank-1 registers, which do not need to be saved, if you ensure that the handler cannot be interrupted by another interrupt source. Otherwise, a raw-mode handler should save any registers used on the stack, and restore them before exiting. The exit should be done with this sequence:

```
ldc.l @r15+,spc
ldc.l @r15+,ssr
rte
nop
```

386 Protected Mode

Evaluation Platforms

i386EX

Processor: 386
Compilers: Microsoft, Borland, Watcom, MetaWare
Debugger: CSi-Mon™

NS486SXF

Processor: 486
Compilers: Microsoft, Borland, Watcom, MetaWare
Debugger: CSi-Mon

DOS PC

Processor: 386, 486, Pentium
Compilers: Microsoft, Borland, Watcom, MetaWare
Debugger: Soft-Scope®

NOTE: This uses U S Software's Hexloader to load the SuperTask! application and run.

The Makefile

The provided **makefile** will build the MultiTask! test programs for any of several environments. These are selected by setting macros near the beginning of the **makefile**. You will need to specify *PTH* as your compiler directory, *APTH* (or possibly *MPTH*) for your assembler directory, and *CVER* for the version of your compiler.

TARGET Specifies the intended target environment. (This name will eventually change to *TRG_ID.*)

TARGET = 1 For all compilers, specifies the 386EX (eval board) as the target. (This is without DOS.) This has two variants: Running with the Soft-Scope debugger (*CSIMON* = 1) and running without (*CSIMON* = 0). When running without the debugger, support code is provided that will handle all initialization from power-on reset, and call the program main function. This version is programmed into the flash memory using a flash program. When running with the debugger, the debugger performs some of the initialization, so the startup code is a little different. This version is downloaded by the CSI monitor programmed into ROM and can be executed or debugged using CSI Soft-Scope.

TARGET = 2 For the Microsoft compiler only, specifies building a Windows95* executable that currently only runs until the multitasker needs a clock interrupt to continue (no longer supported).

TARGET = 3 For the Watcom compiler only, specifies building a DOS Extender application that runs with Pharlap's RUN386* (no longer supported).

TARGET = 4 For all compilers, specifies the NS486SXF* (eval board) as the target. (This is without DOS.) The programs created are downloaded to the board either in ROM or RAM using a flash loader program.

`TARGET = 5` For the Watcom compiler only, specifies building a DOS Extender application that runs with the Rational Systems DOS-extender* (no longer supported).

`TARGET = 6` Boot MT! application from DOS with U S Software **hexload** program.

The kernel code is the same for all targets. The appropriate start-up code and the support code for initializing interrupt vectors, hardware port addresses, interrupt numbers, etc., is specifically selected during a build, based upon the target selected. The correct code will be included in the Multitask! library that is created when you make one of the test programs, and the correct linking commands will be selected to build either an absolute file for debugging, or a hex file for programming in flash.

We suggest all customers build the program **coretest,** which will create a library with the appropriate support code. After you have the **coretest** program running in your environment, you can be sure the library contains the correct support code. You might need to modify some of the support code (**usrclk.asm** contains hardware initialization code) to account for differences in your target hardware.

See the **makefile**, and **compnote.txt** for more details.

Hardware-Dependent Configuring

Many 386 linkers (e.g. USLINK®) and Pharlap's LinkLoc*) will create the structures necessary to run in protected mode. However, they will not handle the transition from real to protected mode. The startup file **ustart1.asm** contains the code to handle this transition for `TARGET = 1`, `CSIMON = 0.`

For the NS486SXF board, `TARGET = 4, CSIMON = 0`, the processor starts out in protected mode, and it uses the different startup code contained in **usstart4.asm**.

If `TARGET = 6`, then the application starts in protected mode as well. `TARGET = 6` specifies U S Software's Hexload support. The file

usstart6.asm contains the code to accept execution from hexload and perform the necessary operations to switch to the application's setup. This includes installing the linker's GDT and IDT, saving the size of memory to be passed to ***meminit()***, and resetting the programmable interrupt controller (PIC) to use interrupt vectors 20h to 27h for the master and 28h to 2Fh for the slave. CSi-Mon is supported in addition to stand-alone.

This code and the initialization code in **usrclk.asm** is specific to the environments supported, and would need possibly extensive modification to support another environment (for example, if you wanted to use it in the process of loading a program under DOS and then taking over the machine and switching to protected mode).

68xxx Platform

Special Issues

Creating Supervisor State Tasks

All tasks that are run under MT! are by default run in the 68000 user state. In order to run a task in the supervisor state, the initial machine status register value for the task is shifted left 16 bits and `OR`ed to the priority value given to ***runtsk()*** when the task is started.

MT! will then use the upper 16 bits of the priority value passed to ***runtsk()*** as the status register value for the task. The label `SUPERMASK` is defined in **depends.h** as the value to `OR` with the task priority to create a supervisor-mode task. Note that it is possible to set an initial interrupt mask level for the task (which may not necessarily be a good idea). You might want to run a task in supervisor state in order to access privileged resources (including instructions). An example is some peripheral registers in the MC68332 that exist only in the supervisor data space. These are accessible only when the processor is in supervisor state. It might be necessary to initialize some of these registers during startup. One way would be to have the startup code in assembler do this before it calls the program ***main()*** function. An alternative is to create a supervisor-state task to perform the initialization once during program startup.

If a supervisor-state task is run for initialization, it could conceivably be useful to set the interrupt mask level of this task higher than the clock tick interrupt level to block rescheduling of tasks until that task is complete. The same thing would be effectively accomplished by giving the ***init*** task a higher priority than all of the other tasks.

Since each task is allocated both supervisor and user stack space, you can save RAM space by making tasks run in supervisor mode. If a

task runs in supervisor mode, it will never use its user-mode stack space, so this can be eliminated.

Example of starting a supervisor state task:

```
runtsk(SUPERMASK | 100, init_task, 2000)
```

Task Stack Space Allocation

This example starts the task ***init_task*** with a status register value of `0x2000`, which puts it in supervisor state. The task priority is 100, and 2,000 bytes of stack space are allocated. The stack argument in ***runtsk()*** specifies the total stack space allocation, which is a combination of the supervisor and user stacks for that task. The stack space allocated is divided into a supervisor stack space of size *`SSTKSIZE`* (defined in **depends.h**), the upper portion of the 2,000-byte allocation, and a user mode stack occupying the remaining space. For a task running in supervisor mode, the allocation is entirely supervisor stack space since the user stack will not be used by that task.

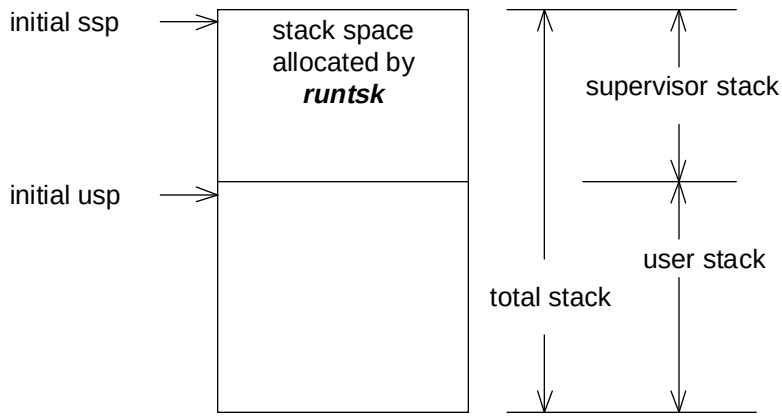

Figure A-1: Task stack space allocation

Stacks in the 68020/68030/68040

When running MultiTask! on 68020 or above processors that have separate supervisor `master` and `interrupt` stack pointers, the `master` stack pointer should not be used. The `master` stack pointer will not be used unless you specifically set the `m` status register bit while in supervisor mode. Check the startup code for your C compiler to ensure that this is not done. As long as the `master` stack pointer is not used, MultiTask! will run properly on any 68000-family processor.

Use of Math Coprocessors

If the 68881 or 68882 math coprocessor is used, the macro *`fp`* in the **makefile** should be set to a non-zero value. This will enable the appropriate code to save the coprocessor status during a task switch. If a coprocessor is used, the supervisor stack allocation for each task should allow about 320 bytes extra for saving the coprocessor state. This is the worst-case amount saved for the coprocessor, and in most instances only a far smaller amount will be saved.

MT_TRAP Vector Initialization

MultiTask! uses one trap vector to switch into supervisor state in various places in **mtsched.s**. The trap number used can be configured in the file **mtcfg2.s**. The vector is initialized by the ***MTstart()*** call. If the vectors in your system reside in ROM, you can omit the initialization code in ***MTstart()***.

Stack Allocation

The supervisor stack requirements for each task must be sufficient to hold the task's context (processor register set), the coprocessor context if applicable, plus the requirements by the deepest level of interrupt nesting anticipated. The supervisor stack space requirement will be essentially the same for all tasks. Set the label `SSTKSIZE` in **mtcfg.s** (or with **stconfig.exe**) to the desired task supervisor stack allocation size (in bytes). The user stack requirement will be determined primarily by the number and size of auto variables used

by the task and all function calls nested within the task, plus some temporary storage space used by the C compiler for storing registers. The user stack requirement can vary considerably from task to task. Each task may have a different stack size specified by the ***runtsk()*** function call, if desired.

The stack size specified to ***runtsk()*** will be the sum of the desired supervisor and user stack allocations for the task. This number must always be larger than the value of `SSTKSIZE`. `SSTKSIZE` bytes of the stack allocation made by ***runtsk()*** will be used for the task's supervisor stack, and the remaining amount becomes the task's user stack allocation. If the task runs in supervisor mode, then the amount of stack space specified by ***runtsk()*** will essentially be all supervisor stack space for the task. For supervisor-mode tasks, the C storage requirements for auto variables, etc. that usually reside on the user stack will instead be on the supervisor stack for that task.

The label `stack_checking` (defined in **mtcfg.s**) when set to 1 causes stack overflow checking code to be included in the scheduler in **mtsched.s**. Leaving this code in will save you a lot of grief. If either the task's supervisor or user stacks are overflowed during a context switch, an illegal instruction will be executed. If your system ever stops at the labels `stack_err` or `sstack_err`, you will know that either the user stack or supervisor stack, respectively, has overflowed for that task and the stack allocation should be increased.

There is also a check for command queue overflow. The command queue is where commands are sent to be executed by the ***MTqcmd*** and ***MTqcmd_c()*** functions. These functions are always called from interrupt routines. If overflow occurs, an illegal instruction at `cmdque_err` is executed. If you have a legitimate need for a deeper command queue, you can define a larger queue by changing the value of `MAX_CMD_CNT` in **depends.h**. The illegal instructions can be replaced by routines of your choice if you want some action performed when these situations arise. A possibility is to use the overflow conditions as a kind of watchdog timer and reset the system. Supervisor stack overflow can be caused by too heavy an interrupt load. A scheduling clock tick interval much shorter than 1 millisecond can overload a 16Mhz 68000.

Configuration

It is essential for proper operation that you correctly set several parameters in the file **depends.h**. These will be set by the supplied **stconfig.exe** program if you run it. The most important of these is PROC_TYPE, which should be set to the appropriate processor value for your target system. This is set to zero for 68000 and 68008 and any variants that have a 6-byte exception frame, and non-zero for all those with an 8-byte exception frame (CPU32, 68010, 68020, 68030, 68040). If this parameter is set wrong, you won't run very long!

Interrupt Considerations

ireqbuf Function Call

The function ***ireqbuf*** is an assembly language function that can be called from an Interrupt Service Routine to request a memory buffer from a pool. This function must **only** be called by ISR routines and not by ordinary "task" code. Furthermore, for any given memory pool, only one ISR can request memory from that pool.

The ***ireqbuf*** function is equivalent to the ***ireqbuf_c()*** function except that it is called from the assembly level rather than C-level code. Both of these functions perform the equivalent operation to the ***reqbuf()*** function, except that to maximize speed no error checking is done. Error recovery in an ISR is generally not possible anyway.

Calling Convention

```
entry:  d0.l = poolid

exit:          a0.l= returned buffer pointer
  (NULL if no buffer available)

altered regs:  d0.l, a0.l, SR
```

Example

```
movem.l d0-d1/a0,-(sp)     save regs
moveq.l #2,d0              poolid = 2
jsr     ireqbuf            get buffer ptr
```

Example: Interrupt Exiting through MT! Scheduler

```
intrtn:

movem.l   d0-d2/a0,-(sp)   save registers used here
          ..  interrupt code ...
moveq.l   #setevt,d0       set event command code
moveq.l   #2,d1            set event 2
jsr       MTqcmd           issue the command
movem.l   (sp)+,d0-d2/a0   restore registers we used

addq.b    #1,_mt_busy
jmp       MTsched          exit through MT!
```

NOTE: This routine exits with a jump to ***MTsched*** rather than an RTE instruction. This provides faster processing of the command issued with the ***MTqcmd_c*** call. If the routine ends with an RTE instead, the command might be delayed for up to one clock tick before it is processed.

Example: User Clock Tick Interrupt

```
intrtn
...  interrupt code ...(preserve any registers used)
jmp     MTtick         enter MT! clock process
```

ISR Coding

On the 68xxx platform, the ***MTsched_c()*** function is not valid. To enter the scheduler from an ISR, you must use a jump at the assembly level. This requirement is imposed because the ***MTsched*** entry point checks the exception frame on the supervisor stack. If the interrupt mask level in the exception frame is non-zero, it assumes the interrupt is nested and does an RTE instead of scheduling. The exception frame must be at `0,sp` when ***MTsched*** is entered; this is impossible to do if entry is from C. If your ISR needs to exit through the scheduler; you can make an entry stub in assembly, call a C-level handler, and on return from the C-level handler, jump to ***MTsched*** after restoring all registers. See **isr0a.s** and **driver0.c** for an example of this.

This scheme will prevent task switches from occurring within a nested interrupt, but requires that you operate normally at interrupt mask level 0. This only poses a problem if you have a spurious interrupt source you are trying to mask by raising the interrupt mask level. You must instead shut off unwanted interrupts at the source.

Interrupt Priorities

The scheduling clock normally should have a low priority such as 1. For fastest interrupt response, interrupt routines that do not call any MT! function can have a priority value higher than `INTOFF` (defined in **mtcfg2.s**).

80960 (i960) Platform

The Makefile

The following macros in the **makefile** should be configured to your particular situation:

COMP macro	You must set the *COMP* macro for the compiler you will be using (Archelon, GNU, or iC960).
PTH macro	Under the conditional section for the compiler you are using (`!if $(COMP) == x`), you must set the *PTH* macro to be the root path where your compiler resides, and the *PTH2* macro to the root path where the Pharlap DOS extender used by the compiler resides. (*PTH2* is not used for Archelon C.)
BOARD macro	Selects one of several evaluation boards we support: The QT960*, EP80960CX,* TomCAt*, and Cyclone* boards. If you are using one of these, setting the *BOARD* macro to the appropriate board number will also select the use of the correct support files for serial I/O and clock interrupt.
PROC macro	Selects the processor you are compiling for. If you are not testing on one of the BOARDs we have routines for, you will need to develop the ***usrclk*** and character I/O routines to match your environment. These are contained in the files **io$(BOARD).ss** and **ioc$(BOARD).c** where the $(*BOARD*) is the numeric value of the *BOARD* macro. The ***usrclk_init*** function also performs some hardware initialization such as writing the IMAP and ICON registers. These should be modified to suit your environment.

TIM macro — Has meaning only when the Cyclone board is selected. `TIM=0`, selects the internal timer 0 of the Jx part to be the source of the clock interrupt (***usrclk()*** and ***usrclk_init()*** functions). `TIM=1` selects the Z8536 counter/timer 1 on the Cyclone base board to be the clock interrupt source.

FP macro — Controls conditional code to save the state of the floating point hardware during a task switch. This should be set to 1 only if using the 960 parts with floating point hardware (e.g. KB, CB).

ALLC macro — Selects between versions of the ***usrclk()*** function written completely in assembly (when `ALLC=0`) or in C (when `ALLC=1`). The ***usrclk()*** function is the Interrupt Service Routine for the clock interrupt.

IRSTACK macro — Controls whether or not a separate interrupt stack will be used for each task. When `IRSTACK=0`, the processor is run always in the interrupt state, so that the hardware stack pointer in use is always the interrupt stack pointer. Each task has its own stack space, which will be the machine interrupt stack while that task is running. This mode of operation is the fastest, and is therefore preferred for all `PROC` types.

When *IRSTACK* is set to 1, tasks normally run in supervisor mode with their own stack, and the upper 512 bytes of their stack space is reserved to be the interrupt stack for the task. When a task switch occurs from an interrupt, the operating system must copy the stack contents from the task interrupt stack to its normal (supervisor mode) stack. This makes the task switch considerably slower.

INEST macro

The operating system must prevent a task switch from occurring during a nested interrupt for reasons explained elsewhere in the manual. There are three different methods of doing this selected by the *INEST* macro. Methods 0 and 1 are available on all processors. Method 2 (which we feel is the best) is available only on the Cx and Jx processor families.

Methods 0 and 1 work by detecting that ***MTsched*** is being entered from a nested interrupt call and then preventing the call to ***MTqproc*** (the command queue processing and scheduling routine). When `INEST=0`, the `pfp` is checked to see if the previous frame type was type `interrupt`; if so a nested interrupt is assumed. When `INEST=1`, priority level in the saved PC is checked and a non-zero value infers a nested interrupt. (The normal operating priority should always be zero when this method is used.)

When `INEST=2`, the `ICON` register is programmed to clear the `IMSK` register when any interrupt is acknowledged (thus masking further interrupts). The `IMSK` value prior to clearing is stored in register `r3` (by the hardware). The ISR can increment *mt_busy*, and then copy `r3` to `IMSK`, reenabling interrupts. As long as all ISRs use this procedure before reenabling interrupts, and then exit to ***MTsched***, nested interrupts (of any priority) will be allowed and no task switch will be performed from inside a nested interrupt. You may of course also leave `IMSK` clear until just before returning from the interrupt to prevent any interrupt nesting.

If you have set the *COMP*, *BOARD*, and *PTH* macros for your environment (and you are using one of the evaluation boards we have support routines for) you should be able to "**make coretest**" and get a working test program in the file called **coretest.dwn** which you can download to the test board and run.

If you are using a different test platform than one of the boards we are supporting with the **makefile**, then you may also need to change the link address. This is controlled with the macros *ROMPAR* and *RAMPAR* in the **makefile**.

The user-buildable targets in the **makefile** are **coretest**, **mtbench**, **siotest**, **tintest**, and **pipetest**. There are a number of other targets used for in-house testing at U S Software only.

Downloading Code to Target Board

A simple program **upload** is provided in C source form and compiled to **.exe**, which will upload code via a serial port to a MON960* or NINDY* monitor and then emulate a terminal. The option *-Pn* selects the port (where n=1 for COM1, etc.). You can also use any other terminal emulator program that can download via XMODEM protocol to these monitors.

Special Issues

Assembly File Preprocessing

Since the assemblers with GNU and the other compilers do not support conditional assembly, we have provided a simple preprocessor which implements this. All assembly source files are supplied with the extension **.ss.** The **makefile** will run the preprocessor step **pp.exe** on these to convert them to **.s** files before assembly.

Interrupt Considerations

Calling C Functions from an Assembly ISR

An 80960 interrupt handler can call C subroutines under the following conditions:

- It must save registers `g0-g14` at the start, and restore these before returning.
- It must make sure `g14` is zero before calling any C code.

Example: ISR Calling C Code

```
intrtn:  mov    sp,r4          get a base pointer
         lda    64(sp),sp      reserve space
         stq    g0,(r4)        save g0-g14
         stq    g4,16(r4)
         stq    g8,32(r4)
         stt    g12,48(r4)
         mov    0,g14          clear g14
         call   _Croutine      call subroutine
         ldq    (r4),g0        restore g0-g14
         ldq    16(r4),g4
         ldq    32(r4),g8
         ldt    48(r4),g12
         ret    returnc
```

The 80960 processors can’t be masked against level 31 interrupts. Therefore, you should use level 31 interrupts for emergency purposes only.

The 80960 tasks need a large stack, especially if they call library functions. About 2000 bytes should suffice in most cases.

80x86 Platform

Evaluation Platforms

The 80x86 version of MultiTask! is applicable to all 8086* family processors and compatible NEC V-series processors running in real address mode. There is also a 386 version for 386 and above processors running in protected mode to provide flat memory address support.

AMD Net 186

Processor:	186
Compilers:	Microsoft, Borland
Debuggers:	Soft-Scope and E86Mon™

DOS PC

Processor:	x86
Compilers:	Microsoft, Borland
Debuggers:	Soft-Scope

The Makefile

For the 80x86, all code modules must be compiled with the same compilation model (e.g., `small`, `medium`, `compact`, `large`, etc.). The **makefile** provided sets the variable *`MODEL`* to the compilation model for all modules and automatically uses this for all compilation and assembly. If you change models, you must be sure to delete all ***.obj** files, which can be accomplished by using **omake clean**.

NOTE: IMPORTANT FOR 80x86 TARGETS
In order to utilize more than 64K of memory with the memory management functions on an 80x86 (real mode) target, the `#define HUGE_MEMORY` in **depends.h** must be set to 1. This can be done with the **stconfig** program or by editing **depends.h** directly, after which the operating system library must be rebuilt.

The `small` and `medium` models give the best performance because data pointers are all `near` pointers. Microsoft C allows the use of "based pointers" that will be controlled by the **makefile** variable *`_BP`*. This allows the `large` and `compact` models to be nearly as efficient as the `small` models.

NOTE: All function name arguments to ***runtsk()*** must be declared as `far` functions in the `Tiny`, `Small`, and `Compact` code models. The `far` is implicit in the other code models.

Special Issues

Using usrclk

The ***usrclk*** routine provided is for running your application on a DOS machine. The ***usrclk_init()*** function chains the original DOS clock interrupt to another interrupt number and, after calling this on a clock tick, jumps into ***MTtick***. The ***usrclk_init*** function will reprogram the timer to any value specified by the *`CLOCKHZ`* label defined in **depends.h**. When *`CLOCKHZ`* is 18, the default PC setting is used, which is actually 18.2 hertz. If you change this by a multiple of 5 (to 91, for example) the actual timer interrupt rate will match the *`CLOCKHZ`* parameter and the ***usrclk*** routine we provide will only chain to the DOS handler at the 18.2 hertz rate; i.e., every fifth

interrupt at 91 hertz. When ***usrclk_term*** is called, it resets the timer to the 18.2 hertz rate and restores the original interrupt vector. If you exit the program prematurely (without executing ***MTterminate()***), which might occur when running under a debugger, then the timer rate does not get restored. Worse yet, the interrupt vector does not get restored, which generally means your computer will crash as soon as something is loaded over the memory location containing the ***usrclk*** routine. Two short programs are provided that might be able to fix this after an abnormal program exit, provided they don't themselves write over a critical area when the clock ticks. The program **clockfix.exe** should be run after an abnormal exit from a program using the 18.2 hertz clock rate, or **clk91fix.exe** after a program using the 91 hertz (or other than 18.2 hertz) rate.

Running an Application Under MS-DOS

A multitasking application can be written and run under DOS using MultiTask! 80x86 with Borland C or Microsoft C. If no DOS calls are made by any task, nothing special needs to be done. If more than one task will make calls to DOS functions (e.g., C-library calls that ultimately make DOS or BIOS calls), then special precautions must be taken.

The problem arises because DOS and many BIOS calls are not reentrant. When a task is inside a DOS call, you must prevent a task switch from occurring to another task that will make a DOS call. A simple way to accomplish this is to treat DOS as a "resource" and surround each function call that will call DOS with a ***getres()*** and ***relres()*** MultiTask! call.

Example

```
#define DOS_RES   0

{
      getres(DOS_RES,0); /* wait for exclusive use of DOS */
      printf(...
      fopen(...
      relres(DOS_RES,0);      /* allow others to use DOS */
```

This technique will allow you to run multiple tasks, each using DOS services. A related problem you will encounter is really a library problem. Most of the C libraries supplied with DOS-resident compilers contain many functions that are not reentrant. Some ANSI functions are also by definition non-reentrant (e.g., ***strtok***). Use of ANSI-defined global variables such as `errno` will make a function non-reentrant, although in the case of `errno`, if you are not using the value in `errno`, this can be ignored.

In the case of non-reentrant library functions, your choices are:

- Don't use them.
- Use them in only one task.
- Protect them with a resource.
- Get a replacement library that is reentrant.
- Write your own reentrant functions.

If you are protecting the library call with a resource, it can be the same `DOS_RES` as used for DOS library calls, or if you know that the function being called does not enter DOS, it can be a separate resource. An example might be floating point math. The compiler-supplied libraries are not reentrant, but the functions do not call any DOS `int` 21H function (I think). In this case you could use a separate resource from the `DOS_RES` to protect these. A better solution might be to call us and inquire about our replacement math libraries that are both reentrant and faster (and in some cases more accurate).

DOS Control-C Handler

In order to make a graceful exit via a user-installed control-C handler, the control-C handler function (which is an interrupt function) should use the ***MTqcmd_c()*** function to issue a ***RUNTSK*** deferred call to run a task that will terminate MultiTask!.

Example

```
void (interrupt far abort_func(void))
{
   MTqcmd_c(RUNTSK, 255, terminate, TASK_STKSIZ);
}

void far terminate(void)
{
   MTterminate();
}
```

When ***abort_func*** is entered after <Ctrl-C> is pressed, it sets up the ***terminate*** task to run at a high priority. This should be the highest priority task if you want immediate action. You could also accomplish the same result by having ***terminate*** run during startup, and then wait for an event that will be set by ***abort_func***.

Running an Application as a DOS TSR

The MultiTask! 80x86 delivery includes an example TSR (Terminate and Stay Resident) application in the source file **tsr.c.** This example demonstrates how a MultiTask! application can be written as a TSR. Once the application is started, DOS becomes the lowest priority task of the application, and the user is returned to the command processor. The DOS prompt reappears on the console, and you are able to run other DOS applications on the system, while the TSR is able to perform activities such as logging data to a disk file.

The main routine for this program calls ***inittsr(0x200)*** before calling ***MTstart().*** The ***inittsr*** function determines the size of the program and saves this value in a global variable `progsize` for the DOS task to use when it makes the DOS TSR call. In our example, we do this by subtracting the application's PSP (Program Segment Prefix) segment value from the stack segment. The stack segment is the last segment of the program, and the PSP immediately precedes the program and is the process ID used by DOS to keep its I/O straight, among other things. The ***fopen*** library call allocates a file buffer in the heap space that must be retained. In ***tsr.c,*** we added 0x200 paragraphs to `progsize` (the argument to ***inittsr()***) to allow for heap

space used by the program. This works for both `small` and `large` program models. In practice, you would probably want to use the `tiny` or `small` program models for a TSR whenever possible. Since the `tiny` model can use no more than 64K, you would be totally safe by specifying a 64K allocation for a `tiny` model TSR, which is 0x1000 paragraphs.

In our example, the only task running besides DOS is *LogTask*. There could be any number of other tasks, but *DosTask* must be the lowest priority, and every other task must be stimulated by an interrupt in some way. Every task must be waiting (i.e., no longer in the run queue) before *DosTask* will run. *LogTask* is reawakened by the clock interrupt. This is accomplished by doing a ***dlytsk()*** for three seconds. Every three seconds, *LogTask* will write a line to a disk file. The file is either **A:\testlog.txt** or the path given by *`argv[1]`* on the command-line, if present. If *`argv[2]`* is present on the command line, it will be included as part of the message written to the output file. The output file is opened in append mode, so if the file already exists, new data is appended to its end.

LogTask writes 10 lines to the output file (one every three seconds for 30 seconds) and then displays a status message on the 25th line of the terminal. The message remains until <Escape> is pressed, at which point the original 25th line contents are repainted, and the TSR takes itself out of memory and terminates.

Application note 2 already describes the method of using a resource for DOS when you have multiple tasks within an application making DOS system calls. For a TSR application, however, something additional is required. DOS provides the necessary hooks via a flag that is set every time a DOS (`int 21h`) call is made, and another interrupt that is called while in the keyboard polling loop within DOS.

The functions ***GetDos()***, ***GetDosCon()***, and ***ReleaseDos()*** employ these hooks in conjunction with a MultiTask! resource to allow a task to safely make DOS function calls from a TSR application. Any task that is going to make a function call through DOS (this includes C library functions) must call either ***GetDos()*** or ***GetDosCon()*** before calling the DOS function(s), and then call ***ReleaseDos()*** when finished.

GetDos() will allow the task to run when it is safe to make any DOS function call except for console I/O. ***GetDosCon()*** will allow the task to run when it is safe to make any DOS call including console I/O (e.g., ***printf***). Console I/O can be performed after a ***GetDos()*** call only via BIOS console functions (e.g., ***status_msg()*** is safe to use after a ***GetDos()*** call since it is built upon BIOS calls rather than DOS `int 21h` calls).

The first time *DosTask* is run, it performs some initialization necessary for ***GetDos()*** and ***GetDosCon(),*** and then calls the DOS TSR function. At this point, the DOS command processor becomes the *DosTask.* and whatever is run from this command processor is in effect the *DosTask.* When some interrupt (clock or other) causes a higher-priority task to wake up, the *DosTask* is suspended until all higher-priority tasks are again dormant. As soon as all tasks with a priority higher than the *DosTask* are dormant (waiting), the *DosTask* will be resumed.

A MultiTask! TSR application must not call ***MTterminate()*** to terminate the way a non-TSR application would. To do so would cause a crash. A TSR application should terminate by a task calling the function ***endtsr(),*** which will perform the necessary cleanup, and then calling ***suspend(cur_task),*** which will cause the final return to the *DosTask.*

If you have added code to chain to or take over any other interrupts, these should be restored to their original state either immediately before or after the call to ***endtsr()*** before the final return to DOS. You might, for example, wish to chain to an interrupt to allow a utility command to be run in the foreground to check the status of or kill the TSR.

Microsoft Publications do not document some of the interrupts used to implement this TSR example. If you intend to write a TSR application, we suggest you obtain some other publication that does. A good example is *DOS Programmer's Reference*, 2nd edition, by Que Publishing (ISBN 0-88022-458-4). Que Publishing's order phone is (800) 428-5331, extension ORDR.

Interrupt Considerations

Example: Interrupt Routine Allowing Nested Interrupts

```
intrtn:
        inc     mt_busy       ;next nesting level
        sti                   ;interrupts enabled
        ...  interrupt code ...
        cli                   ;interrupts disabled
        jmp     MT_sched      ;schedule tasks
```

NOTE: While ***mt_busy*** is non-zero, interrupts are processed but task switching is prevented. It is preferable to keep interrupt routines as short as possible and not allow nesting as above.

Example: Interrupt Exiting through MT! Scheduler

```
intrtn: ...  interrupt processing ...
        push    ax
        push    ds              ;save regs
        mov     ax,seg DGROUP
        mov     ds,ax
        inc     mt_busy         ;next nesting level
        push    2               ;event #2
        push    seg _qp_incevt
        push    offset _qp_incevt
        call    far ptr _MTqcmd_c
        add     sp,6
        call    MTqcmd_c        ;issue the command
        pop     ds              ;restore regs
        pop     ax
        jmp     MTsched         ;exit through MT!
```

Example: User Clock Tick Interrupt

```
intrtn:
        ...  interrupt code ...
        jmp    Mttick                ;enter MT! clock
process
```

ireqbuf Function Call

The function ***ireqbuf*** is an assembly language function that can be called from an Interrupt Service Routine to request a memory buffer from a pool. This function must only be called by ISR routines and not by ordinary "task" code. Furthermore, for any given memory pool, only one ISR can request memory from that pool.

The ***ireqbuf*** function is equivalent to the ***ireqbuf_c*** function except that it is called from the assembly level rather than C-level code. Both of these functions perform the equivalent operation to the ***reqbuf()*** function, except that in order to maximize speed no error checking is done, since error recovery in an ISR is generally not possible anyway.

Calling Convention:

```
entry:   ax = poolid
exit:    ds:bx = returned buffer pointer,
          (NULL if no buffer available)
altered regs:  ax,bx,ds,flags
```

Example

```
        pushf
        push  ax
        push  bx
        push  ds
        mov    ax,2       ;poolid = 2
        call   ireqbuf
                          ; ds:bx  points to memory
```

B. PC-Compatible Console/Keyboard

Chapter Contents

Description

The console/keyboard driver described here supports interrupt-driven keyboard input and text mode screen output, from a PC-AT standard 101-key keyboard and VGA monitor screen in 80 x 25 color text mode. This driver applies only to this environment and is not applicable to any other, and is therefore contained only in the 80x86 distribution. The function of this driver does not make use of any BIOS or DOS function calls, and therefore solves many reentrancy and performance problems associated with using those calls.

The keyboard portion of the driver takes control of the keyboard interrupt when the device is initialized by ***mt_fopen()***. Keyboard input is fully driven by interrupts, requiring no polling for characters by the user. A task performing a read function will wait (relinquish the CPU) until the desired number of characters are ready. Nearly all special keys on the keyboard return simple single-character codes (above 0x80). The codes returned are determined by a table lookup that you can easily modify if you desire different return codes.

The console output portion of the driver provides for dividing the screen into any number of separate non-overlapping windows that we will refer to as *views*. Each of these views is opened by a call to ***mt_fopen()*** and has a separate distinct file handle (`MTFILE *`). This allows separate tasks to control each of these views independently without the need of locking screen control with a resource. The keyboard can be reassigned to any open view**,** allowing that task input with its own separate input buffer.

The sample program **stdemo.exe** (compiled from **stdemo.c**) demonstrates most of the features of this driver, and should be studied as example code.

Usage

To use the console/keyboard driver, you must compile and link the files **condrv.c**, **conintr.asm**, **streamio.c** and **sfm.c** with your application. These files should be in the MultiTask! library file, but can be linked separately if you wish. The device table created in **userio.h** must add entries for the `CON` and `VIEW` devices. The table is set up to do this automatically if you define the label `CIO` when you compile **dev_tab.c**. The application should also include the file **console.h** along with other MultiTask! files for function prototypes and console/keyboard-related definitions.

Before any I/O can be performed, the `CON` device must be opened. This must be done in a task, not in the program main routine. Example:

```
MTFILE *confp;

confp = mt_fopen("CON","r+");
```

Opening this device causes the keyboard interrupt to be intercepted, and the screen to be cleared and set to its initial values of 25 lines of 80 characters each. Note that the driver initialize routine does not actually change the mode registers on the VGA controller board, as this would require specific settings for all possible controller boards. For the driver to function properly, the screen must already be in 80 x 25 color text mode.

After the device is opened successfully, I/O can be performed using the other stream calls with the file handle returned by the open. The ***fopen*** mode should allow both reading and writing if this is intended. If the `CON` device is opened in binary mode, then any read call will return the key value for any key typed without the key being echoed to the screen.

If the device is opened or later placed in *text* mode, then input editing takes place. In text mode, only ASCII printable characters plus <Tab> and <Return> will be returned by the call, and each character returned is also echoed to the display. Furthermore, if a call is used

that requests more than one character, such as ***mt_fgets()***, then line editing takes place. In the ***line edit*** mode, the <Backspace> key will cause the removal of the previous character from both the display and the input buffer. When line editing, the <Escape> key will remove all characters to the beginning of the field. The input buffer size must be greater than the size of the field to be input in line edit mode, otherwise the characters in the buffer will be returned when the buffer is full, and then input will resume with the effect that the line editing cannot back up before the point where the buffer emptied out. The buffer size is currently set to 128 bytes in **userio.h.**

Two macros are provided in **console.h** to switch the path between binary and text modes after the device is open:

set_text_mode(confp);
set_binary_mode(confp);

After the `CON`sole device is opened, you may open one or more paths to the `VIEW` device. This is used to establish a separate path to an area of the screen to which a task can privately write without contending for ownership of the screen with other tasks. Each view is initially opened in *write only* mode. After it is opened, an ***attach_keyboard()*** function call can be performed, which will dynamically set up an input buffer for the view and condition it so that the keyboard can be later assigned to the view with an ***assign_keyboard()*** call. The ***attach_keyboard()*** call modifies the view path so that reading will also be allowed. After the keyboard is assigned to a view with ***assign_keyboard()***, any keystrokes entered will be placed into the input buffer for that view. The console path (`CON`) must remain open at all times because when it is closed, the keyboard interrupt vector will be restored to its original contents and no more keyboard input can be received, even if the keyboard was assigned to another view when the `CON` path was closed. The `CON` path will be automatically closed if the task that opened that path is killed, so this should be avoided also. Alteratively, a field in the `CON` path handle (e.g., `confp->owner_slot`) could be changed to the slot number of another task, which would prevent path closure when the original task is killed.

The console (`CON` device) and all view paths (`VIEW` device) occupy an area of the display described by the coordinates of the upper left and lower right character positions on screen. The upper left corner of the screen has coordinates row 0, column 0. The lower right corner of the screen is row 24, column 79. When the `CON` and each `VIEW` is initially opened, its display area encompasses the entire screen and the foreground and background color attributes are as set in **console.h**. Initially, scrolling and line wrap are also enabled within each view.

Once a view is opened, the function ***view_init()*** can be called to change its screen limits and/or color attributes and other parameters. Each view has its own foreground and background color attributes and cursor position independent of all other views and the console.

A number of utility functions are provided to facilitate writing to the screen and drawing borders around a view as well as handling keyboard assignment. These are in addition to all of the ANSI stream I/O functions (contained in **streamio.c**) that can deal with any open device for which you have a file manager and a driver.

Utility Function Summary

view_init()	Initializes view screen limits and attributes.
get_cursor_loc()	Gets current cursor coordinates.
set_cursor_loc()	Sets current cursor coordinates.
set_cursor_type()	Changes cursor shape.
set_view_attrib()	Sets the default foreground/background color attributes.
put_attribc()	Writes only the attribute character without changing the displayed character.
write_attribc()	Writes a field of attribute characters at specified coordinates.
clear_screen()	Blanks the view screen to background color.
display_box()	Draws a box on the screen at specified coordinates.
box_view()	Draws a box around a view.
attach_keyboard()	Conditions a view to accept keyboard input.
detach_keyboard()	Inverse of ***attach_keyboard***.
assign_keyboard()	Directs future keyboard input to the assigned view.
get_keyboard_assignment()	Returns the file pointer of the view to which the keyboard is currently assigned.
link_view()	Allows <Ctl/Tab> or <Atl/Tab> to move keyboard assignment to the view.
unlink_view()	Inverse of ***link_view***.
set_text_filemode()	Changes an open path's access mode to text.

set_binary_filemode()	Changes an open path's access mode to binary.
freeze_view_attrib()	Causes all attribute characters in a view to remain constant.
thaw_view_attrib()	Causes future output to the view to use the attribute character set by ***set_view_attrib***.

Utility Function Descriptions

assign_keyboard

Assigns keyboard input to a new path.

```
#include "console.h"

int assign_keyboard(MTFILE *fp);
```

fp pointer to the VIEW path with keyboard attached

Assigns all subsequent keyboard input to be directed to the VIEW or CON path *fp*. Any previous path that the keyboard was attached to will receive no further input until the keyboard is again assigned there. Assigning the keyboard to a path cancels any previous assignment so that the keyboard can only be assigned to one path at a time.

NOTE: Typing the <Ctl/Tab> or <Alt/Tab> key combinations will cause the keyboard driver (**condrv.c**) to assign the keyboard to the next or previous linked view path. See ***link_view()***.

Return Value

SUCCESS	function succeeded
EBADAFP	bad MTFILE* argument
EBADASS	bad assignment, cannot assign to path until ***attach_keyboard*** performed

Example

```
#include “console.h”
MTFILE *viewfp;
int status;

status = assign_keyboard(viewfp);
if( status != SUCCESS )
  error_routine();
```

attach_keyboard

Makes keyboard input possible from view.

```
#include "console.h"

int attach_keyboard(MTFILE *fp);
```

fp pointer to the VIEW path

Modifies the path to the view specified by *fp* so that keyboard input may later be requested through that path. This is accomplished by allocating an input buffer along with some new internal structures to modify the path. This allows keyboard input from this path to be isolated from all others. The file access mode is modified to permit reading during this process. No actual input can arrive at the new buffer until the ***assign_keyboard()*** function is called to assign input to this path. The CON path must remain open for the keyboard to be active.

Return Value

SUCCESS	function succeeded
EBADFP	bad MTFILE* argument
EISATT	keyboard already attached
ENOMEM	no *COLOR0* memory available

Example

```
#include “console.h”
MTFILE *confp,*viewfp;
VIEW_INIT_DATA vdata = {BLACK,WHITE,10,40,20,79,1,1};

if(!(confp = mt_fopen(“CON”,”r+”)));   /* open console */
  error_routine(1);
if(!(viewfp = mt_fopen(“VIEW”,”w”)));  /* open view */
  error_routine(2);

view_init(viewfp, &vdata);  /* initialize view limits */

if( !(attach_keyboard(viewfp) )/* condition for input */
  error_routine(3);
```

box_view

Draws a rectangular box around an open view.

```
#include "console.h"

void box_view(MTFILE *fp, int background_color,
              int foreground_color, int style);
```

fp	pointer to the open CON or VIEW path
background_color	background attribute
foreground_color	foreground attribute
style	box line style as defined in **console.h**

Draws a rectangular box on the screen (using PC text mode line draw character set) around the view path specified by *fp*. The box occupies the character positions just outside the view limits of *fp*. In other words, the *ulrow* and *ulcol* positions of the box are one less than the *ulrow* and *ulcol* positions of the view limit, and the *lrrow* and *lrcol* positions of the box are one greater than the *lrrow* and *lrcol* positions of the view limit. The box position must lie within the screen or the result is undefined. The style value is specified as for the ***display_box()*** function and the *background_color* and *foreground_color* parameters specify the attribute character with which the box will be drawn.

Example

```
#include "console.h"
MTFILE *fp;

box_view(fp, BLACK, YELLOW, SINGLE_SIDES);
```

chatout

Writes character and attribute to screen.

```
#include “console.h”

void chatout(MTFILE *fp, int row, int col, uint16 chat);
```

fp	pointer to the open CON or VIEW path
row	the row position to access
col	the column position to access
chat	the combined character and attribute to write

Writes a word containing both the screen attribute byte and the character to display to the absolute screen location given by *row* and *col*. The coordinates *row* and *col* are absolute from the upper left corner of the screen regardless of the current view limits set for *fp*. The character and attribute word *chat* is constructed as: *chat* = ((*background_color* << 12) + (*foreground_color* << 8) + *character*).

Example

```
MTFILE *fp;
uint16 att = (GRAY << 12) + (RED << 8);
char *msgp = “WARNING!”;
byte row=2;
byte col=60;

if( !(fp = mt_fopen(“CON”,”r+”)) )
  return -1;        /* return error */
while( *msgp )
  chatout(fp, row, col++, *msgp++);
```

clear_screen

Blanks the view.

```
#include “console.h”

void clear_screen(MTFILE *fp);
```

fp pointer to the open `CON` or `VIEW` path

Clears the screen within the limits of the current view path pointed to by *fp*, and sets the current cursor position to 0,0 relative to the current view limits. The screen is cleared by writing a space character with the current default attributes to every position within the view limits.

Example

```
#include “console.h”
MTFILE *fp;

clear_screen(fp);    /* blank the screen */
```

detach_keyboard

Makes further keyboard input from the view impossible.

```
#include “console.h”

int detach_keyboard(MTFILE *fp);
```

fp pointer to the VIEW path with keyboard attached

If the keyboard was attached to the view specified by *fp*, it no longer is after this call. The buffers and structures allocated by ***attach_keyboard()*** are deallocated. The read mode permission is removed from the path so that any future read function calls made to this path will return an error status. If the keyboard was also assigned to this path, it will be reassigned to the CON path.

Return Value

SUCCESS	function succeeded
EBADFP	bad MTFILE* argument
ENOTATT	keyboard not attached

Example

```
#include “console.h”
MTFILE *viewfp;
int status;
status = detach_keyboard(viewfp);
if( status != SUCCESS )
  error_routine();
```

display_box

Draws a rectangular box on the screen.

```
#include "console.h"

void display_box(MTFILE *fp, int ulrow, int ulcol,
                 int lrrow, int lrcol, int style,
                 int background_color,
                 int foreground_color);
```

fp	pointer to the open CON or VIEW path
ulrow	absolute row of upper left point
ulcol	absolute column of upper left point
lrrow	absolute row of lower right point
lrcol	absolute column of lower right point
style	box line style as defined in **console.h**
background_color	background attribute
foreground_color	foreground attribute

Draws a rectangular box on the screen using PC text mode line draw character set. The box's position is defined by the coordinates of its upper left and lower right character positions as absolute row and column offsets from the upper left corner of the screen (i.e., coordinates are not relative to the view limits of the path pointed to by *fp*). The upper left position row and column coordinates are given by *ulrow* and *ulcol* respectively, and the lower right row and column by *lrrow* and *lrcol*. The style is specified as SINGLE_ALL for single lines on all sides, DOUBLE_ALL for double lines on all sides, SINGLE_SIDES for single lines on the sides with double lines at the top and bottom, or DOUBLE_SIDES for double lines on the sides and single lines on the top and bottom. A style value of ERASE_BOX can be used to remove the box by blanking the area to the *background_color* specified.

Example

```
#include “console.h”
MTFILE *fp;

display_box(fp,0,0,10,79,DOUBLE_ALL,LIGHT_BLUE,BLACK);
```

freeze_view_attrib, thaw_view_attrib

Macros to control attribute character change behavior.

```
#include “console.h”

void freeze_view_attrib(MTFILE *fp);

void thaw_view_attrib(MTFILE *fp)
```

After the ***freeze_view_attrib()*** call is made, the default attribute character for the view is no longer used when data is written to the view. The attribute previously existing in each character location is left unchanged instead. The ***thaw_view_attrib()*** call restores the default state where the current view attribute byte, as set by ***set_view_attrib()***, is written with each data byte.

Example

```
MTFILE *vfp;                    /* path to open view */
set_cursor_loc(vfp, 5, 10);     /* move cursor */
freeze_view_attrib(vfp);    /* use existing attributes /

mf_fprintf(vfp,”This string uses colors in place”);
thaw_view_attrib(vfp);          /* use default attrib */
```

get_cursor_loc

Gets current cursor location of view.

```
#include "console.h"

uint16 get_cursor_loc(MTFILE *fp);
```

fp pointer to the open CON or VIEW path

Returns the current cursor location of the open path to a CON or VIEW device *fp* in a form suitable for later restoring with the function ***restore_cursor_loc().***

Return Value

uint16 the current cursor position of the path

Example

```
MTFILE *fp;
uint16 cursor_save;

cursor_save = get_cursor_loc(fp);
```

get_keyboard_assignment

Returns the pointer to the path where the keyboard is assigned.

```
#include “console.h”

MTFILE * get_keyboard_assignment(void);
```

Returns the `MTFILE *` to the path where the keyboard is currently assigned.

Return Value

`NULL`	`CON` path not yet opened
<nonzero>	`MTFILE *` to path where keyboard is assigned

Example

```
#include “console.h”
MTFILE *viewfp;

viewfp = get_keyboard_assignment();
```

link_view

Links view with attached keyboard to others.

```
#include “console.h”

int link_view(MTFILE *oldfp, MTFILE *newfp);
```

oldfp MTFILE * to an existing path in the linked list

newfp MTFILE * to the path to add to the list

The path with keyboard attached, *newfp*, is linked to other CON/VIEW paths so that the keyboard driver (in **condrv.c**) can reassign the keyboard to *newfp* when the <Ctl/Tab> or <Alt/Tab> keys are typed. When the CON path is opened, it is the only one in the linked view list. When a VIEW path is later opened, it can be added by first performing the ***attach_keyboard()*** call and the ***link_view()*** call with *oldfp* = CON path, and *newfp* = the new VIEW path.

Return Value

SUCCESS	function succeeded
EBADAFP	bad MTFILE* argument

Example

```
#include “console.h”
MTFILE *confp,*viewfp;
VIEW_INIT_DATA vdata = {BLACK,WHITE,10,40,20,79,1,1};

if(!(confp = mt_fopen(“CON”,”r+”)));   /* open console */
  error_routine(1);
if(!(viewfp = mt_fopen(“VIEW”,”w”)));  /* open view */
  error_routine(2);
view_init(viewfp, &vdata);   /* initialize view limits */
if( !(attach_keyboard(viewfp) )/* condition for input */
  error_routine(3);
if( !(link_view(confp, viewfp) )      /* link view */
  error_routine(4);
```

put_attribc

Writes attribute character only.

```
#include “console.h”

void put_attribc(MTFILE *fp, int attribc);
```

fp pointer to the open CON or VIEW path

attribc the attribute byte to write

Writes the attribute byte *attribc* only at the current cursor position in the CON or VIEW path specified by *fp*. This is used to change the attribute of a character without modifying the character or the current default attribute. The cursor position is not changed. The attribute character is defined as:

```
attribc = (background_color << 4) + foreground_color;
```

Example

```
#include “console.h”
MTFILE *fp;

put_attribc(fp, (BLACK << 4) + YELLOW);
```

restore_cursor_loc

Restores cursor location of view.

```
#include "console.h"

void restore_cursor_loc(MTFILE *fp, uint16 loc);
```

fp pointer to the open CON or VIEW path

loc the cursor location to restore

Restores the cursor location of the open path to a CON or VIEW device *fp* to the value *loc*. The *loc* value should have previously been returned by a call to ***get_cursor_loc().***

Example

```
MTFILE *fp;
uint16 cursor_save;

cursor_save = get_cursor_loc(fp);
{ /* other screen stuff here */ }

restore_cursor_loc(fp, cursor_save);
```

set_cursor_loc

Sets current cursor location within a view.

```
#include "console.h"

void set_cursor_loc(MTFILE *fp, int row, int col);
```

fp	pointer to the open CON or VIEW path
row	row position relative to the path *ulrow*
col	column position relative to the path *ulcol*

Sets the cursor location of the open path to a CON or VIEW device *fp*, to the coordinates given by *row* and *col*. The *row* and *col* coordinates are relative to the view boundaries and cannot be moved outside of them with this call.

Example

```
MTFILE *fp;

set_cursor_position(fp, 2, 10);
              /* Set cursor at row 2 column 10 */
```

set_cursor_type

Sets the shape of the cursor within a view.

```
#include "console.h"

void set_cursor_type(MTFILE *fp, uint16 curs_start,
                     uint16 curs_end);
```

fp	pointer to the open CON or VIEW path
curs_start	starting scan line (0..13)
curs_end	ending scan line (0..13) >= *curs_start*

Sets the starting and ending scan lines of the cursor. The values range from 0..13. Scan line 0 is the top of the cell, and scan line 13 is the bottom. The default values are *curs_start* = 12, *curs_end* = 13, which produces an underline cursor. If the value *CURSOR_INVISIBLE* is ORed with the *curs_start* value, then the cursor will not be displayed within the view.

Example

```
MTFILE *fp;

set_cursor_type(fp, 7, 13);    /* Make cursor thicker */
```

set_text_filemode, set_binary_filemode

Macros to change file access mode.

```
#include “console.h”

void set_text_filemode(MTFILE *fp);

void set_binary_filemode(MTFILE *fp);
```

These macros change the file access mode of any path opened with ***mt_fopen()*** to either text or binary as indicated.

Example

```
#include “console.h”
MTFILE *viewfp;
char linein[80];

set_text_filemode(viewfp);
mt_fgets(linein, 80, viewfp);    /* input with editing */
set_binary_filemode(viewfp);
mt_fgets(linein, 4, viewfp);     /* raw input, no echo */
```

set_view_attrib

Sets the default attribute character for the view.

```
#include “console.h”

void set_view_attrib(MTFILE *fp, int background_color,
                     int foreground_color);
```

fp	pointer to the open CON or VIEW path
background_color	value from **console.h**
foreground_color	value from **console.h**

Sets the default color attribute value for the CON or VIEW path specified by *fp*. Any subsequent characters written to the path with ***mt_fputc(), mt_fputs(), mt_fwrite(),*** or ***mt_fprintf()*** will use this default attribute value.

Example

```
#include “console.h”

MTFILE *fp;

set_view_attrib(fp, BLUE, BRIGHT_WHITE);
```

unlink_view

Unlinks the view path from the list of views.

```
#include “console.h”

int unlink_view(MTFILE *fp);
```

fp MTFILE * to the path to unlink

The path *fp* that was previously linked with other views by ***link_view()*** is unlinked. After unlinking, the <Ctl/Tab> or <Atl/Tab> keys will no longer be capable of assigning the keyboard to this view; however, the keyboard may still be assigned here with a call to ***assign_keyboard().*** If the keyboard was assigned here when the ***unlink_view()*** function was called, it will be reassigned to the previous view in the linked list.

Return Value

SUCCESS	function succeeded
EBADAFP	bad MTFILE* argument

Example

```
#include “console.h”
MTFILE *viewfp;
if( !unlink_view(viewfp) )
   { /* error handling */ }
```

view_init

Initializes view or console limits and attributes.

```
#include “console.h”

int view_init(MTFILE *fp, VIEW_INIT_DATA *p);
```

fp pointer to the open CON or VIEW path

p pointer to initialization data structure

Initializes the view or console window size, attributes, line wrap, and scroll flags. This call can be used repeatedly on an open console view to change these settings.

```
typedef struct view_init_data{
  byte  background_color,
        foreground_color,
        ulrow,            /* view limits
        ulcol,
        lrrow,
        lrcol,
        linewrap,         /* auto linewrap flag
        scroll;           /* scroll flag
  } VIEW_INIT_DATA;
```

Return Value

SUCCESS function succeeded

EBADFP NULL *fp*, or device not open

Example

```
MTFILE *fp;          /* open stream pointer */
VIEW_INIT_DATA vdata = {BLUE,WHITE,5,10,15,70,1,1};

if( !(fp = mt_fopen(“CON”,”r+”)) )
  return -1;  /* return error*/
view_init(fp, &vdata);
```

write_attribc

Writes a field's attribute characters only.

```
#include "console.h"

void write_attribc(MTFILE *fp, int row, int col, int
                   attribc, int cnt)
```

fp	pointer to the open CON or VIEW path
attribc	the attribute byte to write

Writes *cnt* attribute bytes *attribc* starting at the absolute screen position *row,col* in the CON or VIEW path specified by *fp*. This is used to change the attribute of a character without modifying the character or the current default attribute. The cursor position is not changed. The attribute character is defined as:

```
attribc = (background_color << 4) + foreground_color;
```

Example

```
#include "console.h"
MTFILE *fp;

/* Change attribute of 20 characters starting at 1,15 */

write_attribc(fp, 1, 15, (BLACK << 4) + YELLOW, 20);
```

C. Global Variables

Chapter Contents

C

Global Variables

SuperTask! uses a variety of global variables internally, some of which may prove useful when debugging an application. This appendix lists the global variables that SuperTask! uses. Not all of them may be of use when debugging. These definitions are found in **mtio.h**, and most of the variables are associated with the **mtinit** module.

Interrupt-Related Items

`boolean clkon;`	if on, interrupts may add entries to *`cmdque`*
`CMDARG *cmdadd;`	pointer to where next *`cmdque`* entry will go
`CMDARG *cmdprc;`	pointer to next *`cmdque`* entry to process
`uint32 cmdqerrors;`	error flag for queued commands
`CMDARG cmdque[MAX_CMD_CNT];`	command queue

Kernel-Related Items

`MTflag_t mt_busy;`	signals kernel is busy
`MTflag_t mt_block;`	disables task dispatching
`boolean scdflg;`	signals context switch needed
`TASK_DEF task_tab[NUMTSK+1];`	array of task control blocks (TCBs)
`TASK_ID cur_task;`	ID of currently running task
`TASK_DEF * task_ptr;`	pointer to TCB of currently running task
`TASK_DEF * quetab[MAX_Q];`	array of pointers to various queues
`byte *usrsp;`	saves state of processor in ***MTstart***
`TASK_DEF *new_task_ptr;`	pointer to TCB of task to create (used by ***runtsk/inistk***)
`va_list new_argp;`	argument list used by ***runtsk/inistk***

Timing-Related Items

`tick_cnt_t sys_time;`	system time in ticks
`TIME_DEF *mt_timeq;`	pointer to time queue
`TIME_DEF mt_timestr[NUMTIME];`	array of time event structures
`byte tckcnt;`	number of ticks before rescheduling occurs

Facilities-Related Items

`MEM_POOL pool_table[NUMPOOLS];`
array of fixed buffer pools

`uint grp_event_tab[NUMGEVT];`
array of event flags

`byte event_tab[NUMEVT];` array of event flags

`TASK_DEF *resrc_tab[NUMRES];`
resource table

`byte resrc_count[NUMRES];` counts resource acquisition level

`MEMHEAD_DEF mem_rootptr[NUMCOLORS];`
free memory pointers for dynamic memory allocation

`MSG_DEF msgtab[NUMMSG+1];` message header table

`MBX_DEF mbx_tab[NUMMBX];` array of mailboxes

`MSG_DEF *mtbnxt;` pointer to next available spot in *`msgtab`*

Extra Items

`ENV_DEF mtenv[MTENVSIZE];` environment table

`profile_t sys_ticks;` system clock ticks

`profile_t profile_tab[NUMTSK+1];`
task profile data

Processor-Unique Items

```
byte  page_cluster_tab[END_PAGE-FIRST_PAGE];
```
page RAM control table (for Z180-MMU)

Optional Variables

These may be implemented as globals to save stack space. These are only used if the macro `STATIC_OPT1` is defined (generally in **depends.h**).

`TASK_DEF *prv_taskp;`	pointer to previous task in queue
`TASK_DEF *cur_taskp;`	pointer to current task in queue
`TASK_DEF *prvtskp;`	pointer to previous task in queue
`TASK_DEF *curtskp;`	pointer to current task in queue
`TASK_DEF *nxttskp;`	pointer to next task in queue
`byte keypri;`	priority of tasks to rotate
`byte curpri;`	priority of current task in queue

Stream I/O

`DEVICE device_tab[];`	device table
`MTFILE *mtstreams[NUMSTREAMS];`	open streams table

D. Error Codes

D

Chapter Contents

Error Codes Returned by Functions

These are defined in **mtlib.h**.

Label	Value	Meaning
SUCCESS	0	successful operation
E_NOSLOT	-10	no TCB slot available
E_NORAM	-11	no RAM available for stack or packet
E_INVTME	-12	invalid time
E_INVSLT	-13	invalid slot (TASK_ID)
E_INVDLY	-14	invalid delay type
E_INVEVT	-15	invalid event number
E_INVGRP	-16	invalid group event number
E_INVRES	-17	invalid resource number
E_INVMBX	-18	invalid mailbox
E_RELMEM	-19	invalid memory release (corrupt)
E_TIMED_OUT	-20	function timeout expired
E_PTFULL	-21	periodic event table full
E_INVPRF	-22	invalid profile_type code
E_INVPAG	-23	invalid MMU180 page number
E_NOTOPEN	-24	device not open
E_RELDEV	-25	device not open or not device owner
E_INVPID	-26	invalid pool ID
E_INVBSZ	-27	invalid block size for pool
E_INVPT	-28	invalid pool type
E_NOROOM	-29	no table space available for message

D

Error codes (mtlib.h) (continued)

Label	Value	Meaning
E_INVFDP	-30	invalid file descriptor pointer
E_NOTSUS	-31	task not suspended
E_NOTOWNER	-32	not owner of stream
E_IOERR	-33	stream access error
E_INVCOLOR	-34	color requested > NUMCOLORS
E_LATE	-35	missed system time required
E_ENVFULL	-36	mtenv table full
E_TABFULL	-37	acquire/release table full
E_TOOSMALL	-38	too small of memory release to ***MTmeminit***
E_CORRUPT	-39	memory chain corrupt
E_MBXFULL	-40	MBXLIMIT messages in mailbox
E_TOOSML	-41	too small of memory passed to ***MTmeminit***
E_MSGTYPE	-42	not message type expected (pkt vs. msg)
E_UNALLOC	-43	release of unallocated memory
E_NULLPTR	-44	NULL pointer passed to system call

E. Glossary

Address space
: A linear array of locations that a thread can access. Simple processors have only one, and these processors are referred to as ‘linear’ addressing. Some processors allow code and data to have separate spaces and are referred to as ‘split’ addressing. A few processors (IBM-360 and Intel x86) have a large number of ‘segments’, each a linear piece, and are referred to as having ‘segmented’ addressing. In addition, processors may have a special set of I/O instructions that access a distinct I/O space.

Block
: A block is a variable-size piece of memory that a task can acquire. Blocks are allocated from heaps. [Related: Buffer, heap]

Buffer
: A fixed-size piece of memory that a task can acquire. Buffers are allocated from buffer pools. [Related: Block]

Event
: A point in time where something happened, typically things like a button pressed (or released), a character arriving at a UART, an analog to digital conversion completes, a message is received, etc. Business people call them ‘transactions’. MT! provides both counting and multi-bit (group) events.

    Counting event
    : An event variable that can be incremented or decremented. Changing to/from zero can be used to trigger task wakeup.

    Group (multi-bit) event
    : Uses the fact that a typical processor accesses memory in pieces larger than a single bit. MT!

uses a ‘word’ of bits. A task may wait on any bit (or combination of bits) and in that word to be set or cleared.

Event flag	A single bit that can be set or cleared and that is visible to the RTOS’s scheduler. Since most microprocessors don’t support single bits, MT! provides both counting and multi-bit (group) events.
Device	A piece of physical hardware that interfaces a program to the real world. Typical devices are UARTs, disk drives, network chips, ADC/DAC, GPIB, video screens, etc. Today, most devices are being multiplexed (multiplexing the CPU is called multitasking), so most I/O is to files, windows, and connections. [Related: File, stream]
File	A part of a storage device that contains a set of data. It appears to be a small disk drive. Older systems treated files as a series of fixed or variable length records. Newer systems (especially UNIX) use a record size of one byte and, thus, files look like a byte stream. [Related: Stream, device]
Heap	A region of memory used for dynamic memory allocation. We refer to the variable-sized pieces of memory allocated by heaps as blocks. [Related: Block]
Location	Smallest unit of memory that has a unique address. This is typically an 8-bit byte. The implementation (e.g., chip) may physically use a larger size, and is often rated as such: 8-bit, 16-bit, 32-bit, or 64-bit processor. Such a rating does not indicate the size of program that can be run, but is only an efficiency rating. [Related: Word]
Messages	A means of communication between two tasks. As implemented in MultiTask! messages are sent by passing a pointer to data. No additional memory is allocated when a message is passed. [Related: Packets]

MMU — A "Memory Management Unit" is hardware that checks and/or translates a process' addresses into physical addresses. The simplest version simply checks the addresses and provides "memory protection" to prevent a process from damaging other processes' code/data. Other versions may modify the address by adding an offset or using a lookup table. Many systems use a form called 'bank switching'. Multiuser systems use privileged instructions with an MMU to fully isolate each process. With instruction restart and a large disk, this becomes "virtual memory".

Multitasking — Mechanisms that allow a single processor to work on multiple jobs or tasks by switching back and forth between them. If multitasking is within a single address space, it is known as multi-threading. If multiple address spaces are used (via an MMU), it is known as multiprocessing. [Related: Multithreading, multiprocessing]

Multiprogramming — Having more than one program executing at once. This requires having several execution units inside a single chip and/or several processors within a system. Note that this is true parallel execution, not rapid switching as in multitasking.

MUTEX — Mutual exclusion, a method of allowing only one task at a time access to an object. May be implemented as a bit, a count, or as a semaphore. [Related: Resource, semaphore]

Packet — A means of communication between tasks. As implemented in MultiTask!, when a packet is passed, an actual copy of the packet is transferred. Memory is allocated for the packet, and a pointer is not used. [Related: Message]

Process	A term used to label a user addressing space and the program running within it. On a multiprocessing system, there are many processes and each exists in its own address space. Some systems allow a process to be multi-threaded, others do not. [Related: Task, thread, address space, MMU]
Resource	A synchronization facility, usually attached to some hardware or software object, to allow ownership. This allows only one task access to the object at a time. Resources support nesting — that is, a task may acquire a resource several times and must release it an equal number of times before the object becomes available again. Also known as MUTEX (mutual exclusion). [Related: Semaphore, MUTEX]
Semaphore, bit	An exclusion mechanism that allows only one task to own the semaphore at a time. [Related: MUTEX, resource]
Semaphore, counting	An exclusion mechanism that allows only a limited number of tasks to ‘own’ the semaphore at a time. Typically seen as a limit (“There are 5 users out of 10 allowed on this FTP site.”). Resources are more useful in embedded applications. [Related: MUTEX, resource]
Stream	An I/O channel that transfers data in bytes instead of in packets or records. Popularized by UNIX. A UART is a byte stream device. Disk drives and network chips are not stream oriented. [Related: File, device]
Task	Something that needs to get done, usually somewhat independently of other ’tasks’. [Related: Process, thread]

Thread — A separate, distinct set of executions that the processor follows. Internally implemented as some data that include a ‘register set’, the instruction pointer, and certain flags. A multiprocessing system may restrict itself to one thread per addressing space, or it may allow multi-threading within each process. [Related: Process, task]

Word — A term used to reference a data object that is a convenient size for calculations on a given processor. Since the most calculations are performed on addresses, most processors make a word large enough to hold an address. [Related: Location]

There are many exceptions, though, either due to the architecture or just mistakes in the documentation:

- 16-bit word & address processors: 8080/Z80, 68xx, 8051
- 16-bit word >16-bit address processors: Z-180, 8088+, 68HC16, banking
- 32-bit word & address processors: 68000/ ColdFire(*), 80386(flat), M*Core
- (*) 68K is clearly a 32-bit word processor, but Motorola documents it as 16-bit
- Both the Power-PC and MIPS are oriented as 64-bit word and address, but they are 32-bit implementations.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

O

P

R

S

T

U

V

W