SuperTask!®
Quick Start Guide

Revision 6.2
March 2000

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

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**.**

Notes: Indicate important information.

Cautions: Indicate potential damage to hardware or data.

Documentation History

Revision Number	Date
6	February 1997
6.1	July 1998
6.2	March 2000

Contents

1. SuperTask! Quick Start

1

Chapter Contents

Installation

See also: For more detail, see *Installing SuperTask!* in the *SuperTask! User's Guide.*

SuperTask! refers to our multitasking RTOS development suite, which consists of the MultiTask! kernel and the Opus* make utility.

Installing MT!

To install MT! perform the following steps:

1. Insert the MT! disk into disk drive <*d*> and type:

   <*d*>:

   where <*d*> is the drive letter.

2. Type:

   install

   You will see a list of compilers and their codes, from which you can choose the compiler you are using.

3. Then type:

   install <*path*> <*compiler*>

   where <*path*> is the path to the directory where MT! is to be installed, and <*compiler*> is a code specifying the compiler. For example, if one wanted to install MT! into directory **MT** on drive **C:**, then:

<*path*>	=**C:\MT**
<*compiler*>	See list from step 2 above

The install batch file places all source code in the directory specified. A makefile is generated for the indicated compiler. If the directory does not exist, one will be created.

Installing Opus Make

Opus make is provided with the SuperTask! software. The makefiles take advantage of the rich feature set provided with this make utility. To avoid potential conflicts with other make utilities, rename **make.exe** to **omake.exe**. This is how Opus make is used internally at U S Software.

Compiling

See also: For more detail, see the *Makefiles* and *Configuring Multi-Task!* sections in the *SuperTask! User's Guide*.

The installation process creates a makefile for the specified compiler. Initially the makefile is configured to work with U S Software's internal network; therefore, at least one change needs to be made in order for the makefile to work on the development system. Other changes beyond those listed below may be necessary depending on the make utility used by the development system.

The following symbol must be changed:

`PTH` = Path to compiler

The symbol `PTH` points to the directory where the compiler was installed. It does not need to point to the actual compiler, just the path.

At this point, most versions can compile. A few have some additional symbols, such as `CVERS`. If your compiler version is not in the makefile, some additional symbols might need work.

The following symbols are defined using the path defined above and may need to be changed:

`CC` = Command line compiler

`AS` = Assembler

`LNK` = Linker

`LIBR` = Librarian

After `PTH` has been defined, executing the following command will compile all MT! source code and create an MT! library:

```
omake
```

The library is linked with any MT! application in order to include MT! functions. The library name will be **MT**<***STCFG***>**.**<***lib***>, where **<*STCFG*>** is normally **3** and **.**<***lib***> is **.lib** or **.a**. The names of some MT! libraries will also indicate the name of the memory model that was used.

Test Programs

There are two main test programs that should be compiled and run before any attempts are made to create an application. By running both programs, MT! stability and reliability on the development system can be ensured.

You will need to adjust `TRG_ID` and `DBG_ID` to match the evaluation board you have.

If you have any problems, try a “hello, world” program using ***putchr()*** instead of ***printf()*** or ***putchar()***.

Coretest

The first test program is called **coretest.** It is designed to test all the core features of MT! by running many tasks utilizing MT! functions and comparing the internal state of the system with expectations. If **coretest** does not run, there is a severe problem.

Coretest is compiled by issuing this command while in the install directory:

```
omake coretest
```

Load and run **coretest** on your target board. Consult your board documentation for information about downloading to the board memory.

While **coretest** is running, it will display which functions it is currently testing. If an error occurs, **coretest** will stop execution and display which function caused the error along with the corresponding line number of **coretest.c**.

Coretest should be loaded and run on the target board before any development is performed to ensure all downloading procedures are correct.

Mtbench

The second program is called **mtbench**. This program is designed to test the timing of MT! functions in a controlled manner. Information from **mtbench** is useful in determining required task response time. Keep in mind that timing is heavily weighted by the specifics of the application.

Mtbench is compiled by issuing this command while in the install directory:

```
omake mtbench
```

Load and run **mtbench** on your target board. Consult your board documentation for information about downloading to the board memory.

While **mtbench** is running, it will display which functions it is currently testing. If an error occurs, **mtbench** will stop execution.

In addition to providing timing information, **mtbench** will calculate the amount of memory required for the RTOS configuration specified in **mtcfg.h**. It will also display how much space each additional RTOS facility (e.g. another mailbox or resource) will require.

Stream I/O Test Programs

Several other test programs are included with the distribution as listed below:

siotest.c	Stream I/O tests
tintest.c	Timed I/O tests
pipetest.c	Pipe I/O tests
lbiotest.c	Serial I/O tests

First Application

A first use of MT! is best created from one of the many test programs included on the distribution disks. There are four main features an application must have in order to be an MT! success: Header files, tasks, initialization, and termination.

Code for a small sample application is included in the initialization section below.

Header Files

The first feature is the MT! header files. All necessary header files can be included easily by including **rtoshdrs.h**. Header file **rtoshdrs.h** includes these seven necessary files in their proper order:

mtcfg.h	Configuration parameters
mtlib.h	Function prototypes
mtio.h	Prototypes for stream I/O (may be removed in future versions)
mtstdio.h	Prototypes for I/O functions (may be removed in future versions)
mtdata.h	External data definitions
userio.h	User device table definitions (may be removed in future versions)
usrasign.h	User-assigned events, resources, etc.

Files **mtio.h** and **userio.h** can be omitted if no I/O is used.

Tasks

At least one task must be defined, otherwise there is no point in running the system. A task is usually of the form:

```
void  taskname( void )
{
      for(;;){    /* Loop forever */
            /* Perform operations */
      }
}
```

Tasks in an embedded application usually loop over a set of processing commands. The task will only terminate if it reaches the closing brace, is killed by another task, or if MT! is terminated.

Initialization

MT! initialization has several steps:

1. Perform any application-specific initialization.
2. Initialize the MT! system by calling ***MTinitialize()*** and then ***usrclk_init()***. Newer ports may divide the ***usrclk_init()*** function into three separate pieces for finer control.
3. Set up the system memory by calling ***MTmeminit()***.
4. Define at least one task to the system by a call to ***runtsk()***.
5. Start MT! by calling ***MTstart()***.

In the following code, the section in bold type shows the order of initialization:

```
#include “rtoshdrs.h”

/* Define system memory */
char  system_mem[10000];
```

```
/* Define stack size used by task */
#define STACK_SIZE      1000

/* Define task priority */
#define PRIORITY        100

/* Task */
void taskname( void )
{
      for(;;){
            /* Process Info */
      }
}

/* Main application */
void main ( void )
{
      user_initialization();

                        /* Application setup */
      MTinitialize();   /* Initializes MT! */
      usrclk_init();    /* Starts system clock */
      MTmeminit(&system_mem[0],sizeof(system_mem));
                        /* Creates system memory */
      runtsk( PRIORITY, taskname, STACK_SIZE );
                        /* Defines task to MT! */
      MTstart();        /* Starts MT! */
      usrclk_term();    /* Shuts down system clock */
}
```

Termination

Ordinarily an embedded application will not terminate; however, MT! can be terminated by calling function ***MTterminate()*** from the main code or from a task. When all tasks are finished, execution returns to the statement following ***MTstart()*** as indicated in the above code.

Configuration

See also: For more detail, see the section on *Configuring MultiTask!* in the *SuperTask! User's Guide.*

Three files contain the configuration parameters necessary for making MT! conform to the needs of the application: **mtcfg.h**, **depends.h**, and the makefile.

Parameters in mtcfg.h

See also: For more detail, see the section on *Parameters in mtcfg.h* in the *SuperTask! User's Guide*.

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

Parameter	Description	Maximum
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 pools	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

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

The following descriptions give more detail about these parameters:

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. See the manual for more information on the functions ***setevt()***, ***clrevt()***, ***chkevt()***, ***wteset()***, and ***wteclr()***.

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 two bytes of RAM plus the size of a TCB pointer.

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 *NUMGEVT*-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 *NUMGEVT*= 0. For more information on the functions ***reqres()***, ***getres()***, ***relres()***, and ***chkres()***, refer to the Library Reference section.

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` normally must have a value between 1 and 3.

`NUMPOOLS` specifies the maximum number of fixed-size memory pools that you will be using. The pool numbers range from {0..`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 ***klltsk()*** and ***MTterminate()*** are never called and tasks do not terminate.

Parameters in depends.h

See also: For more detail, see the section on *Parameters in depends.h* in the *SuperTask! User's Guide.*

The following parameters appear in **depends.h:**

Parameter	Description	Maximum
NUMTCK	Number of clock ticks/time slice	255
CLOCKHZ	Clock interrupt frequency in hertz	max. unsigned

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" that each task runs will be 20 milliseconds.

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 not automatically reprogram the interrupt rate to match this definition. It is up to you to do so. (Note: The ***usrclk_init()*** routine provided with MT 80x86 is the exception to this.)

Parameters in the Makefile

See also: For more detail, see the section on *Configuration Parameters* in the *SuperTask! User's Guide*.

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. It is also used in the name of the library file -- **MT3.lib**. Each bit is mapped to another variable as follows:

Bit 0 = *TSL*

Bit 1 = *INC_LOCAL_MEM*

Bit 2 = *INC_PROFILING* (if set only)

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

TSL 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 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 *INC_LOCAL_MEM* non-zero, then local memory behaves as described in the manual.

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.

Development Tips

Event Processing

See also: For more detail, see the section on *How to Design Your Application* in the *SuperTask! User's Guide*.

Basic tasks should use the event increment (***incevt***) and wait until set with decrement (***wteset_dec***). Tasks that are event processors will look like this:

```
INIT
FOREVER
  IF something to do
        ...process
        ...maybe incevt(some other task)
  ELSE
        wteset_dec(my_event,1000)
  ENDIF
```

Assuming the counter in the event doesn't overflow:

```
INIT
FOREVER
  IF something to do
        ...process
        ...maybe incevt(some other task)
```

Index

N

O

P

R

S

T

U