The Tics RTOS Programmer's Guide

by Michael D. McDonnell, Tics Realtime

Copyright 1992- 2004, Michael D. McDonnell, Tics Realtime.

All rights reserved.

www.TicsRealtime.com

1 Programmer's Guide

1.1 Introduction

Tics allows C functions (tasks) to run concurrently. Tasks do not invoke one another directly like C functions; they communicate with one another using messages or mail. Tasks may send messages or mail, wait for messages or mail, cancel messages, check for messages or mail, pause, issue timers, cancel timers, suspend, and time-slice with other tasks. Tasks may or may not have a private stack.

Tasks are C functions with the following format:

1.2 Task Format

```
void task(void)
{
   /* Perform one time operations here. */
   while (TRUE) {
       /* Main code for task here. */
   }
}
```

Tasks have no arguments and never return. On entry, tasks perform initializations, if any, then enter an infinite **while** loop.

1.3 Starting Tasks

Tasks are started with the **makeTask** and **startTask** calls.

```
typeTcb * tcb;
  tcb = makeTask(taskName, taskNum);
  startTask(tcb);
```

makeTask creates a task control block for task *taskName* (C function name) with task number *taskNum*. The task control block has the following default values:

Priority equals DEF_PRI (defined in Tics header file).

Time-slicing is disabled.

Three mailboxes are allocated.

One message queue is allocated.

The cooperative bit in the flags field of the tcb is cleared.

startTask adds the task to the ready queue according to its priority and allocates a 1K stack if the cooperative bit in the flags field is clear. Task behavior can be modified prior to starting the task.

```
  tcb = makeTask(taskName, taskNum);
  /* Raise priority */
  tcb->pri = DEF_PRI - 1;
  startTask(tcb);
```

The tcb fields *pri* and *flags* are used to modify task attributes. These fields (except for the cooperative bit) can be modified at any time before or after the task has been started.

1.4 Hello World Example

```
#define NUM_MSGS 100
typeFreeMsg MsgSpace[NUM_MSGS];
void taskA(void)
{
  while (TRUE) {
    printf("Hello from task A\n");
  }
}
void main()
{
  typeTcb * tcb;
  startTics(makeTics(MsgSpace, NUM_MSGS));
  tcb = makeTask(taskA, 0);
  startTask(tcb);
  suspend();
}
```

The listing shows one task (*taskA*) and the main program required to start it. **makeTics** creates a structure of type *typeTics* and returns a pointer to it. **makeTics** defaults the task stack size (1024 bytes), and the timer isr interval (5 milliseconds). **startTics** initializes hardware registers and Tics data structures.

makeTask creates a tcb for *taskA*. **startTask** adds the tcb to the ready queue according to its priority.

suspend suspends the current task, (main in the example), and switches to the next ready task in the ready queue, which in this case is *taskA*. Once *taskA* gains control it runs forever, since it is the only user task in the system.

1.5 Task Attributes

Task attributes are modified by changing the *pri* or *flags* field in the tcb. The default priority is DEF_PRI. To raise a task's priority, choose a number *less* than DEF_PRI.

```
/* Raise priority */
tcbA-> pri = DEF_PRI - 1;
/* Lower priority */
tcbC->pri = DEF_PRI + 1;
```

Task priorities should always be chosen relative to DEF_PRI.

1.5.1 Tcb Flags

Each bit of the *flags* field represents an attribute. See the chapter *Tasks and Task Control Blocks* for a complete description of the tcb and the *flags* bits.

1.6 *Task Switching*

Tasks run until they are preempted or give up control voluntarily. A task gives up control voluntarily by yielding, waiting for a message or mail, issuing a **pause**, or by suspending itself with a **suspend**. A task is preempted when it sends a message to a higher priority task, an interrupt occurs and the isr sends a message to a higher priority task (the isr must invoke **_isrRet** or **_dosIsrRet** in order for this to occur), or its time-slice is used up and the time-slice option is in effect.

1.6.1 Yielding

The **yield** function suspends the active task so that the next ready to run task can run. The task will run again when all tasks of the same priority have run. Note that continuous yielding can starve lower priority tasks. Yielding is recommended only when all tasks run at the same priority.

1.6.1.1 Waiting for a Message

A task can wait for a message using the **waitMsg** function. **waitMsg**(*msgNum*) suspends the active task until a message with message number *msgNum* arrives.

```
#define HELLO 1000
typeMsg * msg;
msg = waitMsg(HELLO);
```

1.6.1.2 Inspecting the Message Queue

The message queue may also be inspected without suspending by using **rcvMsg** as shown below.

```
#define HELLO 1000
typeMsg * msg;
msg = rcvMsg(HELLO);
```

Unlike **waitMsg**, **rcvMsg** always returns. If the desired message is not in the queue, NULL is returned, otherwise **rcvMsg** returns a pointer to the desired message.

1.6.1.3 Waiting forAny Message

A task can wait for a the arrival of any message regardless of its message number by specifying ANY_MSG as the message number.

```
typeMsg * msg;
msg = waitMsg(ANY_MSG);
```

Similarly, the message at the front of the queue can be retrieved by using ANY_MSG with **rcvMsg**.

```
typeMsg * msg;
msg = rcvMsg(ANY_MSG);
```

1.7 *Sending Data with Messages*

Integer data, long data, or a pointer to data can be sent with a message by filling in the appropriate fields of the message as shown below.

```
#define DATA 1000

typeMsg * msg;
extern typeTcb * TcbB;
struct {int x,y,z;} dataStruct;

msg = makeMsg(TcbB, DATA);
msg->iData = 5; /* Integer data */
msg->lData = 7L; /* Long data */
msg->pData = &dataStruct; /* Ptr data */
sendMsg(msg);
```

Data is retrieved by the receiving task by extracting the data from the received message as shown below.

```
int iData;
long lData;
typeMsg * msg;
typeData * dataPtr;

msg = waitMsg(DATA);
iData = msg->iData;
lData = msg->lData;
dataPtr = msg->pData;
freeMsg(msg);
```

1.8 Freeing Messages

Note that when a message is received it must be freed after use. Freeing a message adds it back to the message free list so that it can be reused by **makeMsg**.

1.9 Waiting for Multiple Messages

It is sometimes necessary to launch numerous messages, and then wait for responses. This is illustrated by the example below.

1.9.1 Inter-task Communication Example

```
#define HELLO 1000
#define HELLO_BACK 1002

typeMsg * timer, * msg;
long timerSeqNum;
extern typeTcb * TaskA;

timer = makeTimer(200L);
timerSeqNum = timer->seqNum;
startTimer(timer);
sendMsg(makeMsg(TaskA, HELLO));
msg = waitMsg(ANY_MSG);
switch (msg->msgNum) {
  case TIMEOUT:
    /* Handle msg timeout... */
    break;
  case HELLO_BACK:
    cancelTimer(timer, timerSeqNum);
   /* Handle HELLO_BACK... */
   break;
}
```

A timer is started and its message and sequence number are saved. A message is then sent to *TaskA,* and the task waits for a response. On receiving a message the task resumes, and the first message in the message queue is retrieved. The switch statement is then used to handle the message. Notice that in the case of receiving a HELLO_BACK the TIMEOUT message is cancelled.

1.10 Timed Wait for Message

It is sometimes useful to wait only so long for a message; if the message is not received within the time period, the message is cancelled. This can be done by starting a timer as in the example above or by using **waitTimedMsg** as shown below.

```
#define HELLO 1000
typeMsg * msg;

msg = waitTimedMsg(HELLO, 2000L);

if (msg->msgNum == HELLO) {
/* Msg was received before the timeout. */
}
else {
/* Timeout occurred. */
}
freeMsg(msg);
```

The code allows 2 seconds for the message HELLO to arrive. **waitTimedMsg** returns a pointer to a message. The message will either be a TIMEOUT message or the HELLO message, whichever arrives first.

```
#define NUM_MSGS 100
#define HELLO 1000
#define HELLO_BACK 1001
typeFreeMsg MsgSpace[NUM_MSGS];
typeTcb * TcbA, * TcbB;

void taskA(void)
{
  typeMsg * msg;
  while (TRUE) {
    sendMsg(makeMsg(TcbB, HELLO));
    freeMsg(waitMsg(HELLO_BACK));
  }
}
void taskB(void)
{
  typeMsg * msg;
  typeTcb * senderTcb;
  while (TRUE) {
    msg = waitMsg(HELLO);
    senderTcb = msg->senderTcb;
    freeMsg(msg);
    sendMsg(makeMsg(senderTcb, HELLO_BACK));
  }
}
void main()
{
  startTics(makeTics(MsgSpace, NUM_MSGS));
  TcbA = startTask(makeTask(taskA, 0));
  TcbB = startTask(makeTask(taskB, 1));
  suspend();
}
```

1.11 Responding to Messages

The listing shows two tasks, *taskA* and *taskB*. *taskA* sends a message to *taskB* and waits for a response. On receiving the message, *taskB* responds. This process continues forever. *taskB* responds without prior knowledge of the sender by accessing the *senderTcb* field in the received message.

1.12 Canceling Timers and Messages

It is sometimes necessary to cancel timers and messages. In order to cancel a message or timer the sequence number of the message must be known. This is acquired from the message structure after the message is constructed as shown below.

```
#define HELLO 1000
typeMsg * timer, * msg;
long timerSeqNum, msgSeqNum;
extern typeTcb * TaskA;

timer = makeTimer(200L);
timerSeqNum = timer->seqNum;
startTimer(timer);

msg = makeMsg(TaskA, HELLO);
msgSeqNum = msg->seqNum;
sendMsg(msg);

cancelTimer(timer, timerSeqNum);
cancelMsg(msg, msgSeqNum);
```

1.13 Message Numbers

User message numbers must not be less than 0 or greater than 32767.

1.14 Timers

Tics supports three types of timers: pause, one-shot timers, and periodic timers. Timers may also be cancelled. **pause**'s are used in-line to suspend program execution for a specified number of milliseconds.

1.14.1 Pause

```
selectPort();
pause(20L);  /* Wait 20 ms before reading. */
readPort();
```

1.14.2 Starting a Timer

One-shot timers are issued as follows.

```
typeMsg * timer;
timer = makeTimer(100L);
```

```
startTimer(timer);
/* Do other things... */
waitMsg(TIMEOUT);
```

startTimer causes a message with number TIMEOUT (defined in the Tics header file) to be sent to the issuing task after the specified timer interval has elapsed.

1.14.3 Modifying Timer Attributes

Timer attributes may be modified as follows.

```
#define READ_DATA 1000
typeMsg * timer;
extern * IoPollingTcb;

timer = makeTimer(100L);
/* Make it a periodic timer */
timer->flags |= PERIODIC;
/* Change receiver of TIMEOUT msg. */
timer->destTcb = IoPollingTcb;
/* Change msg num */
timer->msgNum = READ_DATA;
startTimer(timer);
```

The timer is changed to a PERIODIC timer (default is one-shot timer), the message number has been changed to READ_DATA, (default is TIMEOUT), and the destination task has been changed to *IoPollingTcb* (the default is ActiveTcb). When the timer times out the message READ_DATA will be sent to the task *IoPollingTcb*.

1.14.4 Periodic Timers

A periodic timer sends a timeout message every *n* milliseconds.

1.14.4.1 Periodic Timer Example

```
void keyboardMonitor(void)
{
  typeMsg * timer;
  timer = makeTimer(100L);
  timer->flags |= PERIODIC;
  startTimer(timer);
  while (TRUE) {
    freeMsg(waitMsg(TIMEOUT));
    if (kbhit()) processKey();
  }
}
```

The timer is issued only once on task entry. Every 100 milliseconds thereafter the task will receive a TIMEOUT message. The example above is more easily done with a **pause** as shown below. (The example above is presented solely to illustrate the mechanics of using periodic timers.)

```
void keyboardMonitor(void)
{
  while (TRUE) {
    pause(100L);
    if (kbhit()) processKey();
  }
}
```

Periodic timers are used only when precise timing is a requirement. For example, when generating hardware control signals, periodic timers are essential to insure that signals are generated on hard boundaries. It is not enough to use a **pause** since the pause is re-issued by the application, and, because of the time it takes to re-issue the **pause** and the possibility of preemption by higher priority tasks, precise timing is not guaranteed. Periodic timers, however, are re-issued from within the timer isr, regardless of system dynamics.

1.15 Time-slicing

Time-slicing is very rarely used in real-time systems. The preferred way of sharing time between tasks is by using the **pause**, **waitMsg**, or **waitMail** functions. Time-slicing forces preemption which may not be desirable. The **pause** function allows tasks to voluntarily relinquish control without preemption. See the chapter *Time-slicing* in *The Art of Real-time Programming* for more details.

Time-slice tasks share CPU time with other time-slice tasks of the same priority. Each task is allowed to run for *n* milliseconds, where *n* is set in the tcb field *timeSlice*, which may be changed dynamically. The default time-slice is 50 milliseconds. Time-slicing behaves as follows:

A task must have time-slicing enabled for it to be time-sliced. Time-slicing is enabled by setting the TIMESLICE bit in tcb->flags. TIMESLICE is defined in the Tics header file.

A task enabled for time-slicing will not preempt a task of the same priority that is disabled for time-slicing.

Tasks will only time-slice with other tasks of the same priority that are time-slice enabled.

Multiple groups of tasks can time-slice, where all tasks within each group have the same priority. Note, however, that if the highest priority group never voluntarily suspends, time-slice tasks of lower priority will not run.

1.15.1 Changing Time-slice Attributes

To change the time-slice flag dynamically, change the *flags* field in the tcb.

```
/* Disable time-slicing */
tcb->flags &= ~TIMESLICE;
/* Enable time-slicing */
tcb->flags |= TIMESLICE;
```

To change the number of milliseconds allocated to the time-slice task change the tcb field *timeSlice.*

```
tcb->timeSlice = 200; /* Task gets 200 ms per timeslice */
```

1.16 Critical Regions

A critical region can only be entered by one task at a time and is used for resource sharing. Consider a multi-tasking system in which all tasks must share a single printer. If tasks write to the printer whenever they have need to, printout from different tasks can become interleaved. Access to the printer can be managed with a critical region manager. To create a critical region manager, start an instance of task *RegionMgr*, which is a task defined in the Tics Kernel.

```
typeTcb * Printer;
Printer = startTask(makeTask(RegionMgr, 0));
Tasks use the print manager as shown below.
enterRegion(Printer);
/* Use printer ... */
exitRegion(Printer);
```

Using the enter and exit region calls serialize access to the printer. If a task calls **enterRegion** when another task is in the region, the task will suspend until the other task performs an **exitRegion**. Any number of region managers can be created. When Tics is initialized it creates a region called *Dos* whose use is described in a later section.

1.17 Preemption

Preemption occurs when:

An interrupt occurs and the isr sends a message to a task whose priority is higher than the active task. (In order for the higher priority task to run, **_isrRet** or **_dosIsrRet** must be issued from within the isr).

The active task sends a message to a task that has a higher priority.

A timer times out for a task whose priority is higher than the active task. (This is the same case as the first item)

The active task's time-slice is up and another time-slice task of the same priority is ready to run.

1.18 Priority

User priorities range from the lowest priority of LOW_USER_PRI to the highest priority of HI_USER_PRI. Both are defined in the Tics header file. The default priority is DEF_PRI, defined in the header file. Priorities of lower numerical value have a higher priority. Most real-time systems run well with all tasks at the default priority. Only assign high priority to those tasks that have a hard time constraint. All other tasks should run at the default priority.

Priorities apply to both tasks and messages. Tasks are put into the ready queue according to their priority while messages are put into each individual task's message queue according to the message priority. See the chapter *Priority* in *The Art of Real-time Programming* for more details.

1.19 Multi-tasking with MS-DOS

MS-DOS and the associated BIOS are non-reentrant. Preempting a task that is in DOS, and running another task that enters the same region of DOS can cause errors. This is avoided by avoiding preemption. Preemption is easily avoided by running all tasks at the same priority with time-slicing disabled. Under these conditions tasks may use DOS freely without concern.

If however, preemption or time-slicing is required with DOS, the **enterRegion/exitRegion** function calls must be used.

```
enterRegion(Dos);
/* Perform DOS operations... */
exitRegion(Dos);
```

If all tasks use DOS services in this way, time-slicing and preemption are allowed. This approach can be slower since tasks must wait for DOS access, and there is overhead associated with the **enterRegion/exitRegion** calls.

1.20 Cooperative Tasks

Cooperative tasks have the following format.

```
void task(typeMsg * msg)
{
  switch (msg->msgNum) {
  /* Handle each possible msg that
  maybe received with a case statement. */
  }
}
```

1.20.1 Cooperative Tasks Share the Same Stack

The main distinction between cooperative tasks and normal tasks is that all cooperative tasks share the same stack. Normal tasks are assigned a private stack. So, for 20 tasks each with a 1K stack, 20K of RAM must be available for stack space. Conversely, 100 cooperative tasks can share a single 1K stack.

Cooperative tasks have the following characteristics.

All cooperative tasks share the same stack.

A cooperative task is a C function that is called directly when a message has been received for it.

Cooperative tasks cannot suspend, i.e., they cannot wait on a message, or pause. The cooperative task is a message processor. For each message that it receives it takes an action and then returns. In some cases it may have to save its state.

Unlike normal tasks, cooperative tasks must return after processing the message.

Cooperative tasks may only use the a subset of the Tics functions. See the chapter *Cooperative Tasks* for details.

A normal task cannot be changed to a cooperative task once the normal task has been started.

Cooperative tasks must not free the message argument; the cooperative task need only return.

When normal tasks run, the global variable *ActiveTcb* points to the tcb for the active task. This is not true for cooperative tasks. The tcb for the cooperative task is pointed to by *ActiveCoopTcb*.

1.20.2 Application of Cooperative Tasks

Cooperative tasks are essential for applications that require a large number of tasks - typically multiple instances of a few base tasks. Hundreds of tasks can be started without the normal stack overhead required for each task. Cooperative tasks are ideal for event driven applications like communications, process control and the like.

1.20.3 Starting Cooperative Tasks

Cooperative tasks are created by setting the COOP flag in the tcb.

```
tcb = makeTask(taskA, 0);
/* Select cooperative option */
tcb->flags |= COOP;
startTask(tcb);
```

For more information see the chapter *Cooperative Tasks.*

1.21 Sending and Receiving Mail

NOTE: Mail should not be used. Mailboxes are only included here for compatibility with earlier versions. Always use messages as they are much more flexible and they are the core around which Tics is built.

Sending mail differs from message passing as follows.

No allocation or deallocation of memory is required for mail.

Mail is not queued. Unread data is optionally overwritten.

Sending mail is faster than sending messages.

Mail is sent to mailboxes. Thirty two mailboxes are allowed per task instance. Each task is allocated 32 mail flags and 3 mailboxes. The number of mailboxes can be increased up to a maximum of 32 by changing the constant LEN_MAILBOXES which is defined in the Tics header file.

1.21.1 Mail Example

```
#define NUM_MSGS 100
#define HELLO 1000
typeFreeMsg MsgSpace[NUM_MSGS];
void taskA(void)
{
  typeMail * mail;
  while (TRUE) {
    mail = waitMail(HELLO);
    printf("Data is %d\n", mail->iData);
    freeMail(HELLO);
  }
}
void main()
{
  typeTcb * tcb;
  startTics(makeTics(MsgSpace, NUM_MSGS));
  tcb = makeTask(taskA, 0);
  startTask(tcb);
  mail = makeMail(tcb, HELLO);
  mail->iData = 256;
  sendMail(mail);
  suspend();
}
```

1.21.2 Multiple Task Instance Example

The same task can be started multiple times with the different task numbers.

```
#define NUM_MSGS 100
typeFreeMsg MsgSpace[NUM_MSGS];
void monitorLine(void)
{
  int ioPort[] = {0x40, 0x41, 0x42, 0x43};
  int taskNum, reading;
  taskNum = ActiveTask->taskNum;
  while (TRUE) {
    pause(100L);
    reading = readPort(ioPort[taskNum]);
    if (reading << 10 || reading > 100) {
     error(taskNum);
    }
  }
}
void main()
{
  int i;
  startTics(makeTics(MsgSpace, NUM_MSGS));
  for (i = 0; i << 4; i++) {
    startTask(makeTask(monitorLine, i));
  }
  suspend();
}
```

Four instances of the same task are started with separate task numbers. The task is generic since the same code is used for all four instances. The task reads the IO port based on the task number.

1.22 Data Structures

The major Tics data structures are **typeMsg** and **typeTcb**, both defined in the Tics header file. **typeFreeMsg** is a generic type that populates the free list.

Message queues are doubly linked lists of **typeMsg** structures. When a message is sent, a message is allocated from the free list, set with the proper values, and put into the message queue of the destination task. For complete details of the Tics data structures see the chapters *Tasks and Task Control Blocks, Inter-task Communication Using Messages*, and *Inter-task Communication Using Mail*.

1.23 Programming Isr's

A basic isr is shown below.

```
#define NEW_DATA 1000
 extern typeTcb * IoTcb;
void interrupt far IoIsr(void)
{
   typeMsg * msg;
   char ioChar;
   ioChar = inportb(0x50);
   msg = _makeMsg(IoTcb, NEW_DATA, FALSE);
   msg->iData = ioChar;
   _sendMsg(msg, FALSE, FALSE);
   _isrRet();
}
```

In this example, data is read, a message is made, initialized, and sent. The **_makeMsg, _sendMsg** combination is the only allowed way to send a message from within an isr. **_isrRet** returns from the isr. If a higher priority task is sent a message from within the isr, **_isrRet** will suspend the active task, and return to the higher priority task. When running on the PC under MS-DOS, **_dosIsrRet()** must be used instead of **_isrRet()**.

1.24 Timer List

The timer list is a doubly linked list of message structures called *timers*. When a timer is started with the **pause** or **startTimer** function, a timer is created and added to the timer list. The *startTicCount* and *ticCountDown* fields of the timer are initially loaded with the timer count. Each time the timer isr is entered, the *ticCountDown* field is decremented. When it reaches zero, a TIMEOUT message is sent to the originator of the timer, which is the *senderTcb* field of the timer. If the timer is a periodic timer, the *ticCountDown* field is reloaded with *startTicCount* each time *ticCountDown* reaches 0.

1.24.1 Differential Timers

The timer list is sorted numerically with the shortest timer at the front of the list. Furthermore, the timers are differential timers. For example, say that the list is composed of three timers of duration 10, 20, and 30 system ticks. The list would be sorted numerically with the 10 tick timer at the front of the list followed by the 20 and the 30 tick timer. However, the count loaded into *startTicCount* and -*ticCountDown* would be 10 ticks for each timer. When the first timer of 10 ticks has counted down to zero, then only 10 more ticks are required to reach the 20 count required for the second timer. This technique of sorting differential timers means that only the first timer in the list needs to be decremented.

1.25 Tics Parameter Structure

The Tics parameter structure is created by **makeTics** and is shown below.

```
typedef struct structTics {
typeFreeMsg * msgSpace;
int numMsgs;
int flags;
void (* fatalErrorHandler)(char * errMsg);
int timerChipCount;
long uSPerTimerChipCount;
} typeTics;
```

msgSpace points to the free message list. Memory blocks are removed from this list to create messages, task control blocks, and timers. *msgSpace* points to an area of memory that is used as the free memory pool for messages and task control blocks. This memory pool is used by all tasks in the system. It should be large enough the handle the worst case system loading. One hundred messages is a good number to start with. *numMsgs* tells how many messages of type *typeFreeMsg* are contained in the message space pointed to by *msgSpace*. If *numMsgs* is not large enough, an out of memory error will be generated.

The *flags* field is currently unused.

fatalErrorHandler points to an error handler function that will be called when a fatal error is detected.

timerChipCount contains the actual raw count to be loaded into the timer register. *uSPerTimerChipCount* contains the number of microseconds that will have elapsed when the timer chip count of *timerChipCount* counts down to zero.

1.26 Mailbox Structure

Each task is allocated 3 mailboxes as a default. Tasks can have up to 32 mailboxes. The mailbox structure is defined below.

```
typedef struct structMail {
int iData;
long lData;
void * pData;
struct structTcb * senderTcb;
} typeMail;
```

iData, *lData*, and *pData* can contain integer, long, or pointer data for mail. Each task's tcb contains an array of mailbox structures (tcb field *mailBoxes*). The mailbox number is used as an index into the mail table to obtain mail data. The *mailFlags* field of the tcb is an unsigned long. Each bit is a mail flag. It the bit is set, then that mailbox has mail. For example, if bit 5 is set, then *tcb->mailBoxes[5]* has mail data. Note that the mailbox table is only accessed if there is data associated with the mail. Often mail is sent without data to simply signal another task.

1.27 The Free Message List

Tics maintains a list of free memory blocks that are allocated when a message, timer, or task control block is created. The structure is shown below.

```
typedef struct structFreeMsg {
unsigned long seqNum;
struct structFreeMsg * next;
char space[ticsMax(sizeof(typeTcb), sizeof(typeMsg))];
} typeFreeMsg;
```

seqNum is a sequence number that is assigned every time a memory block is granted. It is used for timer and message cancellation. *next* points to the next free memory block in the singly linked free message list.

1.28 Notes & Caveats

Some Tics function calls are macros, therefore function calls should be enclosed in braces when used with *if* statements.

Each normal task instance requires about 1.2K bytes of overhead - 1K for the stack, and about 220 bytes for the tcb and message queue. The stack size is one of the fields in the *typeTcb* structure that is returned by **makeTask**. It is recommended that the stack size be no less than 1K when running with MS-DOS.

Each cooperative task instance requires about 100 bytes of overhead - no stack or message queue is required.

Software floating point operations are typically non-reentrant. If tasks can preempt one another, floating point operations must be protected with **enterRegion**, **exitRegion**. In general all non-reentrant library calls should be protected by a region when preemption can occur.

The task priority is meaningless for cooperative tasks. Cooperative tasks use a run to completion paradigm, and communicate via messages. Therefore, only the message priorities have significance.

Do not use **pause** within an **enterRegion(Dos), exitRegion(Dos)** region. The timer isr will not preempt while the task is in a DOS region and deadlock will occur.

Mail may only be used for communication between normal tasks; cooperative tasks may not use mail.

For efficiency, the read operation for mail is unprotected. This means that writing to a mailbox is an indivisible operation; reading is not. This is not considered a problem, since there is typically only one reader of mail. The modification to include read protection is straightforward.

This version of Tics (3.01) disables interrupts when linked lists require updating. Also, when the timer isr is entered, interrupts are disabled while in the isr. For very time critical applications this may not be acceptable and we recommend the Tics State Machine Kernel in these cases, since the State Machine Kernel never disables interrupts. The Tics 3.01 Kernel is also modifiable so that interrupts are not disabled, however some flexibility is compromised. We plan to cover this topic in an upcoming book entitled *Modifying, Porting, and Embedding the Tics Kernel.*

1.29 Installation, Compiling, and Linking

Standard Tics is comprised of the source files, *tics.c* and *target.c,* and the header files, *tics.h* and *ticsext.h*, and *ticsmain.h*. To build applications, simply include the appropriate header files, and link *tics.c* and *target.c* with your application file(s). The Extended version includes the files *extras.c*, *ticscom.c*, and *ticscom.h*. See the *readme.doc* file on the distribution disk for further details.

1.30 Embedding Tics

Tics is easily embedded because all variable initializations are done at run time; no compile-time initializations are performed. Various vendors offer products that allow for embedding C based executable files onto an 86 platform. If your target is not an 86 family processor, target specific kernel code must be modified (file *target.c*), and the C source must be cross compiled for the target processor. If you have purchased our support package we can help you with porting via email and phone support.

1.31 Sample Programs

Sample programs are provided for learning. The best way to gain an understanding of various features of the Tics Kernel is to begin working with the MS-DOS sample programs.

1.32 Errors

Return codes are documented in the system calls section. When fatal errors occur, the fatal error handler is invoked. See the *readme.doc* file for further details.

1.33 Debugging

Because Tics is written in C, it can be used with the standalone C debuggers. Note that when using Turbo C/C++ 1.01, the standalone Turbo Debugger is required. The Turbo IDE Debugger can be used with certain restrictions. See the *readme.doc* file for further details.

1.34 User Level Kernel Interface

User level kernel calls are the most commonly used kernel calls. Most of the sample programs use user level calls only. If you need more power, generality, or you need to access the kernel from within an isr, refer to the *System Level Kernel Interface* section. System level calls provide more flexibility, but may be more difficult to use.

cancelMsg, cancelTimer

1.34.1.1 Function

Cancel a message, cancel a timer.

1.34.1.2 Syntax

```
int cancelMsg(typeMsg * msg, long seqNum);
int cancelTimer(typeMsg * msg, long seqNum);
```

1.34.1.3 Remarks

These functions attempt to cancel a message pointed to by *msg*, with sequence number ***seqNum***. If the message is in the message queue, it is removed. However, if the message has already been received or cancelled, it is not removed. If a message is to be cancelled, a pointer to the message and its sequence number must be saved prior to sending it.

1.34.1.4 Return Value

Returns TRUE if successful, otherwise FALSE.

1.34.1.5 See Also

sendMsg, startTimer.

1.34.1.6 Example

```
#define READ 1000
typeMsg * msg;
extern typeTcb * TcbIO;
long seqNum;
msg = makeMsg(TcbIO, READ);
seqNum = msg->seqNum;
sendMsg(msg);
cancelMsg(msg, seqNum);
msg = makeTimer(1000L);
seqNum = msg->seqNum;
startTimer(msg);
cancelTimer(msg, seqNum);
```

enterRegion, exitRegion

1.34.1.7 Function

Gain or release exclusive access to a protected region.

1.34.1.8 Syntax

```
void enterRegion(typeTcb * regionMgrTcb);
void exitRegion(typeTcb * regionMgrTcb);
```

1.34.1.9 Remarks

Following the **enterRegion** call, a task has exclusive access to a protected region, assuming that all other tasks that require access also use the **enterRegion** call. When finished using the region, the task must relinquish ownership by calling **exitRegion**.

regionMgrTcb points to an instance of task **regionMgr**. An instance of task **regionMgr** is required for each region. **regionMgr** task instances are started like any other task instance. See the example below for details. The **regionMgr** instance named **Dos** is started by **startTics**. **Dos** is used by MS-DOS tasks that need exclusive access to MS-DOS calls.

enterRegion(Dos)/exitRegion(Dos) calls are only required when time-slicing is in effect, or tasks are being run at different priorities. If all tasks are run at the same priority, and no tasks time-slice, **enterRegion(Dos)/exitRegion(Dos)** calls are not required.

1.34.1.10 Return Value

None.

1.34.1.11 See Also

None.

1.34.1.12 Example

```
/* To protect MS-DOS system calls...*/
enterRegion(Dos);
/* DOS system calls here... */
exitRegion(Dos);


/* To setup a user declared region... */

typeTcb * UserRegion;
UserRegion=
startTask(makeTask(regionMgr,0));
/* Now use the region */
enterRegion(UserRegion);
/* Perform operations... */
exitRegion(UserRegion);
```

exitTics

1.34.1.13 Function

Exit the Tics Kernel and restore the hardware to its original state. For use on MS-DOS systems only.

1.34.1.14 Syntax

```
void exitTics(void);
```

1.34.1.15 Remarks

exitTics restores the PC hardware timer chip, the timer isr, and the keyboard isr to their original state. **exitTics** should always be called prior to exiting when using the Tics Kernel under MS-DOS.

1.34.1.16 Return Value

None.

1.34.1.17 See Also

None.

1.34.1.18 Example

```
/* Exit Tics. */
exitTics();
/* Exit to MS-DOS. */
exit(0);
```

freeMail

1.34.1.19 Function

Clear the mail available flag for the mailbox.

1.34.1.20 Syntax

```
int freeMail(typeMail * mail);
```

1.34.1.21 Remarks

Each tcb contains 32 mail flag bits; if a bit is set, mail is available for the corresponding mailbox number. Mailbox numbers range from 0 to 31. **freeMail** clears the mail available bit for the mailbox pointed to by *mail*.

1.34.1.22 Return Value

None.

1.34.1.23 See Also

makeMail, sendMail, rcvMail, waitMail

1.34.1.24 Example

```
#define IO_DATA 12
int rampUpTime;
typeMail * mail;

mail = waitMail(IO_DATA);
rampUpTime = mail->iData;
freeMail(mail);
}
```

freeMsg

1.34.1.25 Function

Free a message.

1.34.1.26 Syntax

```
void freeMsg(void * msg);
```

1.34.1.27 Remarks

freeMsg adds the message pointed to by *msg*, to the free message list.

1.34.1.28 Return Value

None.

1.34.1.29 See Also

_freeMsg.

1.34.1.30 Example

```
typeMsg * msg;
msg = rcvMsg();
if (msg != NULL) {freeMsg(msg);}
```

makeMail, sendMail

NOTE: Mail should not be used. Mailboxes are only included here for compatibility with earlier versions. Always use messages as they are much more flexible and they are the core around which Tics is built.

1.34.1.31 Function

Make mail, send mail to a task.

1.34.1.32 Syntax

```
typeMail * makeMail(typeTcb *tcb, mailBoxNum)
typeMail * sendMail(typeMail * mail)
```

1.34.1.33 Remarks

makeMail creates a mailbox structure with mailbox number *mailBoxNum* and destination *tcb*. **sendMail** sends the mail pointed to by *mail*. **sendMail** over-writes unread mail unless the OVER_WRITE_DISABLE bit in the *flags* field of the mailbox is set. OVER_WRITE_DISABLE is defined in the Tics header file.

1.34.1.34 Return Value

Returns a pointer to the mail structure.

1.34.1.35 See Also

_sendMail, freeMail, rcvMail, waitMail.

1.34.1.36 Example

```
#define IO_DATA 12
typeMail * mail;
typeTcb * TcbA;

mail = makeMail(TcbA, IO_DATA);
sendMail(mail);
```

makeMsg, sendMsg

1.34.1.37 Function

Make a message, send a message to a task.

1.34.1.38 Syntax

```
typeMsg * makeMsg(typeTcb * tcb, int msgNum);
typeMsg * sendMsg(typeMsg * msgPtr);
```

1.34.1.39 Remarks

makeMsg creates a message with number *msgNum* and destination *tcb*. **sendMsg** sends the message pointed to by *msgPtr*.

1.34.1.40 Return Value

Returns a pointer to the message.

1.34.1.41 See Also

freeMsg.

1.34.1.42 Example

```
typeMsg * msgPtr;

extern typeTcb * TcbIO, * TcbA;
msgPtr = makeMsg(TcbIO, START);
/* Raise msg priority */
msgPtr->pri = DEF_PRI - 1;
/* Change sender */
msgPtr->senderTcb = TcbA;
/* Send the message to ioTask */
sendMsg(msgPtr);
```

makeTask, startTask

1.34.1.43 Function

Make a task control block, start task execution.

1.34.1.44 Syntax

```
typeTcb * makeTask(void (* task)(void), int taskNum);
typeTcb * startTask(typeTcb * tcb);
```

1.34.1.45 Remarks

makeTask creates a task control block (tcb) for the C function *task*, and assigns it the task number *taskNum*.

makeTask defaults the tcb entries as follows.

Priority is set to DEF_PRI (defined in Tics header file).

Time-slicing is disabled.

One message queue is allocated.

Three mailboxes are allocated.

startTask allocates a 1K stack, and schedules the task to run according to its priority.

1.34.1.46 Return Value

Returns a pointer to the tcb.

1.34.1.47 See Also

None.

1.34.1.48 Example

```
typeTcb * ioTcb;
ioTcb = makeTask(IoTask, 0);
/* Raise task priority */
ioTcb->pri--;
/* Enable time-slicing */
ioTcb->flags |= TIMESLICE;
/* Start task execution */
startTask(ioTcb);
```

makeTics, startTics

1.34.1.49 Function

Initialize the Tics Kernel and hardware.

1.34.1.50 Syntax

```
typeTics * makeTics(typeFreeMsg * msgSpace, int numMsgs);
typeTics * startTics(typeTics * ticsInfo);
```

1.34.1.51 Remarks

These functions must be called before any other Tics function. **makeTics** creates a *typeTics* data structure and returns a pointer to it. *MsgSpace* points to a block of memory to be used for message space. *numMsgs* is the number of messages of type **typeFreeMsg** that is represented by the space. The elements of this structure are defined below with the default values shown in parentheses.

int timerChipCount - the number to be loaded into the timer chip countdown register. (5966).

long uSPerTimerChipCount - the number of microseconds that will have elapsed when the counter reaches 0. (5000L).

*typeFreeMsg * msgSpace* - points to space that Tics will use as its free memory pool. (User must supply this space).

int numMsgs - the number of messages of type **typeFreeMsg** that can fit into the space pointed to by *msgSpace*. (User must supply this argument).

void (fatalErrorHandler)(char * errMsg)* - a pointer to an error handler that will be called whenever a fatal error occurs. (Defaults to Tics error handler which is MS-DOS compatible).

1.34.1.52 Return Value

Returns a pointer to a structure of type *typeTics*.

1.34.1.53 See Also

None.

1.34.1.54 Example

```
#define NUM_MSGS 200
typeFreeMsg MsgSpace[NUM_MSGS];
void main()
{
  typeTics * tics;
  tics = makeTics(MsgSpace, NUM_MSGS);
  /* Use my fatal error handler */
  tics->fatalErrorHandler = myErrorHandler;
  /* Set timer isr interval to 10 ms */
  tics->timerChipCount *= 2;
  tics->uSPerTimerChipCount *= 2L;
  startTics(tics);
}
```

makeTimer, startTimer

1.34.1.55 Function

Make a timer message, start a timer.

1.34.1.56 Syntax

```
typeMsg * makeTimer(long msTimeout);
typeMsg * startTimer(typeMsg * msgPtr);
```

1.34.1.57 Remarks

makeTimer makes a timer message. **startTimer** starts the timer countdown. On timeout, the message TIMEOUT (defined in the Tics header file) is issued to the destination task after *msTimeout* milliseconds.

makeTimer defaults the message structure members as follows.

> *msgNum* is TIMEOUT.
>
> *destTcb* is ActiveTcb.
>
> Timer type is one-shot.

1.34.1.58 Return Value

Returns a pointer to the message.

1.34.1.59 See Also

_makeTimer.

1.34.1.60 Example

```
extern typeTcb * TaskA;
typeMsg * timer;
/* Make timer message */
timer = makeTimer(500L);
/* Change sender */
timer->senderTcb = TaskA;
/* Make timer periodic */
timer->flags |= PERIODIC;
/* Start timer countdown */
startTimer(timer);
```

pause

1.34.1.61 *Function*

Suspend task execution for a given number of milliseconds.

1.34.1.62 *Syntax*

```
int pause(long msPause);
```

1.34.1.63 *Remarks*

pause suspends the current task for *msPause* milliseconds.

1.34.1.64 *Return Value*

None.

1.34.1.65 *See Also*

makeTimer, startTimer.

1.34.1.66 *Example*

```
void ioTask(void)
{
  while (TRUE) {
    pause(20L);
    if (kbhit()) processKey();
  }
}
```

rcvMail

1.34.1.67 Function

Returns the mail data in the mailbox.

1.34.1.68 Syntax

```
typeMail * rcvMail(int mailBoxNum);
```

1.34.1.69 Remarks

If mail is available, **rcvMail** returns a pointer to the indicated mailbox number, otherwise NULL is returned. Mailbox numbers range from 0 to 31.

1.34.1.70 Return Value

If mail is available, **rcvMail** returns a pointer to the indicated mailbox, otherwise NULL is returned.

1.34.1.71 See Also

makeMail, sendMail, freeMail, waitMail

1.34.1.72 Example

```
#define IO_DATA 12
typeMail * mail;
mail = rcvMail(IO_DATA);
if (mail != NULL) {freeMail(mail);}
```

rcvMsg

1.34.1.73 *Function*

Retrieve a specific message from a task's message queue.

1.34.1.74 *Syntax*

```
typeMsg * rcvMsg(int msgNum);
```

1.34.1.75 *Remarks*

rcvMsg retrieves the next available message from the active task's message queue with number *msgNum*. If *msgNum* equals ANY_MSG, and the queue is not empty, the first message in the queue is returned. If the desired message is not in the queue, NULL is returned. The message must be freed after use by calling **freeMsg.**

1.34.1.76 *Return Value*

Returns a pointer to the message if the message available, otherwise, NULL is return.

1.34.1.77 *See Also*

waitMsg.

1.34.1.78 *Example*

```
#define ANALOG_DATA 1000
typeMsg * msgPtr;
int data;
msgPtr = rcvMsg(ANALOG_DATA);
if (msgPtr != NULL) {
  data = msgPtr->iData;
  freeMsg(msgPtr);
}
```

suspend

1.34.1.79 Function

Suspend the active task.

1.34.1.80 Syntax

```
void suspend(void);
```

1.34.1.81 Remarks

suspend suspends the active task and runs the task at the front of the ready queue. The suspended task will run again only if a message is sent to it, or a **wakeup** is executed.

1.34.1.82 Return Value

None.

1.34.1.83 See Also

wakeup.

1.34.1.84 Example

```
sendMsg(makeMsg(TaskA, 0));
sendMsg(makeMsg(TaskB, 1));
sendMsg(makeMsg(TaskC, 2));
startTimer(makeTimer(1000L));
suspend();
/* Execution will resume here when a
   message has arrived or another task
   issues a wakeup to this task. */
```

waitMail

1.34.1.85 *Function*

Wait for mail.

1.34.1.86 *Syntax*

typeMail * waitMail(int mailBoxNum);

1.34.1.87 *Remarks*

waitMail waits for mail to arrive in mailbox number *mailBoxNum* for the active task. If mail is available, control is returned immediately, otherwise, the active task is suspended until mail arrives. Mail must be freed after use by calling **freeMail**. Mailbox numbers range from 0 to 31.

1.34.1.88 *Return Value*

If mail is available, a pointer to the mailbox is returned, otherwise, the active task is suspended until mail for the indicated mailbox arrives.

1.34.1.89 *See Also*

sendMail, freeMail, rcvMail

1.34.1.90 *Example*

```
#define IO_DATA 12
int data;
typeMail * mail;

mail = waitMail(IO_DATA);
data = mail->iData;
freeMail(mail);
```

waitMsg

1.34.1.91 Function

Wait for a specific message.

1.34.1.92 Syntax

```
typeMsg * waitMsg(int msgNum);
```

1.34.1.93 Remarks

waitMsg waits for a message with number *msgNum* to appear in the active task's queue and returns a pointer to it. If *msgNum* equals ANY_MSG, the first message in the queue is returned. If a message is available, control is returned immediately, otherwise, the active task is suspended until a message arrives. The message must be freed after use by calling **freeMsg**.

1.34.1.94 Return Value

Returns a pointer to the message.

1.34.1.95 See Also

rcvMsg.

1.34.1.96 Example

```
typeMsg * msgPtr;
int data;
msgPtr = waitMsg(START);
data = msgPtr->iData;
freeMsg(msgPtr);
```

wakeup

1.34.1.97 Function

Wake up a suspended task.

1.34.1.98 Syntax

```
void wakeup(typeTcb * tcb);
```

1.34.1.99 Remarks

wakeup wakes up the suspended task connected to the task control block *tcb*.

1.34.1.100 Return Value

None.

1.34.1.101 See Also

suspend.

1.34.1.102 Example

```
extern typeTcb * TaskA;
wakeup(TaskA);
```

_dosIsrRet, _isrRet

1.34.1.103 Function

Return from an MS-DOS isr, return from an isr.

1.34.1.104 Syntax

```
void _isrRet();
void _dosIsrRet();
```

1.34.1.105 Remarks

_isrRet and **_dosIsrRet** both check the priority of the active task against the task at the front of the ready queue. **_isrRet** always returns to the highest priority task. **_dosIsrRet** returns to the highest priority task only when the system is not in an MS-DOS system call. Use **_dosIsrRet** when running under MS-DOS. These calls must be the last statement of the isr.

1.34.1.106 Return Value

None.

1.34.1.107 See Also

_isrPreempt.

1.34.1.108 Example

```
 extern typeTcb * IoTcb;
void interrupt far IoIsr(void)
{
  typeMsg * msg;
   char ioChar;
   ioChar = inportb(0x50);
   msg = _makeMsg(IoTcb, NEW_DATA, FALSE);
   msg->iData = ioChar;
   _sendMsg(msg, FALSE, FALSE);
  _dosIsrRet();
}
```

_freeMsg

1.34.1.109 Function

Free a message.

1.34.1.110 Syntax

void _freeMsg(void * msg, int diOpt);

1.34.1.111 Remarks

_freeMsg adds the message pointed to by *msg*, to the free message list. *diOpt* is TRUE if interrupts are to be disabled when required, otherwise interrupts are assumed to be disabled on entry.

1.34.1.112 Return Value

None.

1.34.1.113 See Also

freeMsg.

1.34.1.114 Example

```
typeMsg * msg;
msg = rcvMsg();
if (msg != NULL) {_freeMsg(msg, TRUE);}
```

_isrPreempt

1.34.1.115 Function

Preempt active task from within an isr.

1.34.1.116 Syntax

```
void _isrPreempt(int diOpt);
```

1.34.1.117 Remarks

_isrPreempt is typically used from within an isr to preempt the active task so that when a return from interrupt is performed, control is returned to a higher priority task that the isr put into the ready queue. If interrupts are explicitly enabled while in the isr, then *diOpt* should be TRUE, otherwise *diOpt* should always be FALSE when invoking **_isrPreempt** from within an isr. **_isrPreempt** preempts the active task and runs the task at the front of the ready queue even if the active task has a higher priority. **_isrPreempt** assumes that the user has made the appropriate checks and has determined that preemption is required. **_isrRet** and **_dosIsrRet** both check the priority of the active task against the task at the front of the ready queue. **_isrRet** always returns to the highest priority task. **_dosIsrRet** returns to the highest priority task only when the system is not in an MS-DOS system call. These two calls are preferred over **_isrPreempt** and should be used whenever possible. However, if other considerations beyond the priority of the ready to run task and the active task are at issue, then **_isrPreempt** should be used.

1.34.1.118 Return Value

None.

1.34.1.119 See Also

_isrRet, _dosIsrRet.

1.34.1.120 Example

```
extern typeTcb * IoTcb;
void interrupt far IoIsr(void)
{
  typeMsg * msg;
  char ioChar;
  ioChar = inportb(0x50);
  msg = _makeMsg(IoTcb, NEW_DATA, FALSE);
  msg->iData = ioChar;
  _sendMsg(msg, FALSE, FALSE);
  if (ActiveTcb->pri > IoTcb->pri &&
  notInDos()) {
    _isrPreempt(FALSE);
  }
}
```

_makeMsg, _sendMsg

1.34.1.121 Function

Make a message, send message to a task.

1.34.1.122 Syntax

```
typeMsg * _makeMsg(typeTcb * tcb, msgNum, int diOpt);
typeMsg *_sendMsg(typeMsg * msgPtr, int diOpt, int preeOpt);
```

1.34.1.123 Remarks

_makeMsg creates a message with number *msgNum* and destination *tcb*. **_sendMsg** sends the message pointed to by *msgPtr*. *diOpt* is TRUE if interrupts are to be disabled when required, otherwise interrupts are assumed to be disabled on entry. If interrupts are explicitly enabled during isr execution, then *diOpt* should be TRUE, otherwise *diOpt* should always be FALSE when invoked from within an isr. *preeOpt* is TRUE if the active task is to be preempted if necessary. *preeOpt* should always be FALSE when invoked from within an isr.

1.34.1.124 Return Value

Returns a pointer to the message.

1.34.1.125 See Also

None.

1.34.1.126 Example

```
extern typeTcb * IoTask;
extern typeTcb * TaskA;
typeMsg * msgPtr;
msgPtr = _makeMsg(IoTask, START, TRUE);
/* Raise msg priority */
msgPtr->pri = DEF_PRI - 1;
/* Change sender */
msgPtr->senderTcb = TaskA;
/* Send the message */
_sendMsg(msgPtr, TRUE, TRUE);
```

_makeTimer

1.34.1.127 Function

Make a timer message.

1.34.1.128 Syntax

typeMsg *_makeTimer(long ticCount, int diOpt);

1.34.1.129 Remarks

_makeTimer makes a timer message. ***ticCount*** is the number of system tics. A system tic occurs each time the timer isr is entered. If interrupts are explicitly enabled during isr execution, then *diOpt* should be TRUE, otherwise *diOpt* should always be FALSE when invoked from within an isr.

_makeTimer defaults message structure members as follows.

msgNum is TIMEOUT.

destTcb is ActiveTcb.

Timer type is one-shot.

1.34.1.130 Return Value

Returns a pointer to the message.

1.34.1.131 See Also

makeTimer.

1.34.1.132 Example

```
extern typeTcb * TaskA;
typeMsg * timer;
/* Make timer message */
timer = _makeTimer(500L, TRUE);
/* Change sender */
timer->senderTcb = TaskA;
/* Make timer periodic */
timer->flags |= PERIODIC;
/* Start timer countdown */
startTimer(timer);
```

_rcvMsg

1.34.1.133 Function

Retrieve a message from a task's message queue.

1.34.1.134 Syntax

```
typeMsg * _rcvMsg(typeTcb * tcb, int msgQNum, int msgNum, int diOpt, int
remOpt);
```

1.34.1.135 Remarks

_rcvMsg gets the message with number *msgNum* and message queue number *msgQNum* from *tcb*. If *msgNum* equals ANY_MSG, the first message in the queue is returned, otherwise the requested message is returned. If *remOpt* is TRUE, the message is removed from the queue, otherwise, the message is left in the queue. This option is useful for checking the queue without removing messages. In all cases NULL is returned if the desired message is not in the queue. *diOpt* is TRUE if interrupts are to be disabled when required, otherwise interrupts are assumed to be disabled on entry. If interrupts are explicitly enabled during isr execution, then *diOpt* should be TRUE, otherwise *diOpt* should always be FALSE when invoking **_rcvMsg** from within an isr. The message must be freed after use by calling **freeMsg** or **_freeMsg**.

1.34.1.136 Return Value

Returns a pointer to the message or NULL if the requested message is not available.

1.34.1.137 See Also

rcvMsg

1.34.1.138 Example

```
/* Get msg 5 from msg queue 0 for the active task. */
typeMsg * msgPtr;
msgPtr = _rcvMsg(ActiveTcb, 0, 5, TRUE, TRUE);
```

_sendMail

1.34.1.139 Function

Send mail to a task.

1.34.1.140 Syntax

```
void _sendMail(typeMail * mail, int diOpt, int preeOpt)
```

1.34.1.141 Remarks

_sendMail sends the mail pointed to by *mail*. **_sendMail** over-writes unread mail unless the OVER_WRITE_DISABLE bit in the *flags* field of the mailbox is set. OVER_WRITE_DISABLE is defined in the Tics header file. *diOpt* is TRUE if interrupts are to be disabled when required, otherwise interrupts are assumed to be disabled on entry. *preeOpt* is TRUE if the active task is to be preempted if necessary.

1.34.1.142 Return Value

None.

1.34.1.143 See Also

sendMail

1.34.1.144 Example

```
#define IO_DATA 12
typeMail * mail;
typeTcb * TcbA;

mail = makeMail(TcbA, IO_DATA);
sendMail(mail);
```

_waitMail

1.34.1.145 Function

Wait for mail.

1.34.1.146 Syntax

```
typeMail * _waitMail(int mailBoxNum, int diOpt)
```

1.34.1.147 Remarks

_waitMail waits for mail to arrive in mailbox number *mailBoxNum* for the active task. If mail is available, control is returned immediately, otherwise, the active task is suspended until mail arrives. *diOpt* is TRUE if interrupts are to be disabled when required, otherwise interrupts are assumed to be disabled on entry. Mail must be freed after use by calling **freeMail**. Mailbox numbers range from 0 to 31.

1.34.1.148 Return Value.

Returns a pointer to the mailbox.

1.34.1.149 See Also

_sendMail

1.34.1.150 Example

```
#define IO_DATA 12
int data;
typeMail * mail;

mail = _waitMail(IO_DATA, TRUE);
data = mail->iData;
freeMail(mail);
```

_waitMsg

1.34.1.151 Function

Wait for a specific message in a specific message queue.

1.34.1.152 Syntax

```
typeMsg *
 _waitMsg(int msgQNum, int msgNum, int diOpt, int remOpt);
```

1.34.1.153 Remarks

_waitMsg waits for a message with number *msgNum* to appear in the message queue *msgQNum* and returns a pointer to it. If a message is available, control is returned immediately, otherwise, the active task is suspended until a message arrives. If *msgNum* equals ANY_MSG, the first message that arrives in the queue is returned, otherwise the requested message is returned as soon as it arrives. *diOpt* must be FALSE if interrupts are disabled when the call is made, otherwise, it must always be TRUE. If *remOpt* is TRUE, the message is removed from the queue, otherwise, the message is left in the queue. This option is useful for checking the queue without removing messages. The message must be freed after use by calling **freeMsg** or **_freeMsg**.

1.34.1.154 Return Value

Returns a pointer to the message.

1.34.1.155 See Also

waitMsg.

1.34.1.156 Example

```
#define MSG1 1000
#define MSG2 1001
typeMsg * msgPtr;
/* Wait for the arrival of any msg, but
do not remove it from the queue */
_waitMsg(MSGQ, ANY_MSG, TRUE, FALSE);
/* Now use _rcvMsg to see if MSG1 or MSG2 has arrived */
if (msgPtr = _rcvMsg(ActiveTcb, MSGQ, MSG1, TRUE, TRUE) != NULL) {
   processMsg1(msgPtr);
   freeMsg(msgPtr);
}
if (msgPtr = _rcvMsg(ActiveTcb, MSGQ, MSG2, TRUE, TRUE) != NULL) {
  processMsg2(msgPtr);
  freeMsg(msgPtr);
}
```

_wakeup

1.34.1.157 Function

Wake up a suspended task.

1.34.1.158 Syntax

void _wakeup(typeTcb * tcb, int diOpt);

1.34.1.159 Remarks

_wakeup wakes up the suspended task connected to the task control block *tcb*. *diOpt* is TRUE if interrupts are to be disabled when required, otherwise interrupts are assumed to be disabled on entry.

1.34.1.160 Return Value

None.

1.34.1.161 See Also

suspend.

1.34.1.162 Example

```
extern typeTcb * TaskA;
_wakeup(TaskA, TRUE);
```

waitTimedMsg

1.34.1.163 Function

Wait for a message. If the message does not arrive within the indicated number of milliseconds, wake the waiting task with a TIMEOUT message.

1.34.1.164 Syntax

```
typeMsg * waitTimedMsg(int msgNum, long timeout);
```

1.34.1.165 Remarks

This function performs a **waitMsg** function call for message number *msgNum*. If the message does not arrive within *timeout* milliseconds, the waiting task is sent a TIMEOUT message. The function cancels the timer if the message arrives in time.

1.34.1.166 Return Value

Returns a pointer to the TIMEOUT message or the expected response, whichever arrives first.

1.34.1.167 See Also

None.

1.34.1.168 Example

```
#define DATA 1000

typeMsg * msg;

msg = waitTimedMsg(DATA, 1000L);

switch (msg->msgNum) {

case DATA:
  processData(msg);
  break;

case TIMEOUT:
  handleTimeout();
  break;

}
freeMsg(msg);
```

yield

1.34.1.169 Function

Voluntarily relinquish control so that other tasks can run.

1.34.1.170 Syntax

void yield(void);

1.34.1.171 Remarks

yield suspends the current task so that other tasks can run. **yield** is typically used to share time between two or more tasks.

1.34.1.172 Return Value

None.

1.34.1.173 See Also

None.

1.34.1.174 Example

```
void ioTask(void)
{
  while (TRUE) {
    readIOData();
    yield();
  }
}
void keyboardTask(void)
{
  while (TRUE) {
    if (kbhit()) {processKey();}
    yield();
  }
}
```

The two tasks above share time by first doing their work, and then yielding so that the other task can run.

getMem, _getMem

1.34.1.175 Function

Allocate a block of memory.

1.34.1.176 Syntax

```
void * getMem(typeMem * * memPtr);
void * _getMem(typeMem * * memPtr, int diOpt);
```

1.34.1.177 Remarks

These functions return a block of memory from the memory pool specified by *memPtr*. The memory pool must first be created by **makeMem**. *_getMem* is useful when memory must be allocated from within an isr when interrupts are disabled. A TRUE *diOpt* argument indicates that interrupts are already disabled**.**

1.34.1.178 Return Value

Returns a pointer to the requested block of memory. If no memory is available, a fatal error is generated.

1.34.1.179 See Also

makeMem, putMem.

1.34.1.180 Example

```
#define DATA 1000
extern typeMem * MemPoolA;
extern typeTcb * TcbA;
typeMsg * msg;

msg = makeMsg(TcbA, DATA);
msg->pData = getMem(&MemPoolA);
sendMsg(msg);
```

makeMem, _makeMem

1.34.1.181 Function

Create a pool of fixed size memory blocks.

1.34.1.182 Syntax

```
typeMem * makeMem(typeMem * memMgr, void * memPtr,
int poolSize, int blkSize);
typeMem * _makeMem(typeMem * memMgr, void * memPtr,
int poolSize, int blkSize, int diOpt);
```

1.34.1.183 Remarks

These functions create a pool of fixed size memory blocks. Any number of pools may be created with successive calls to either of these functions. Each memory block will be *blkSize* bytes in length. The memory blocks are created from a caller supplied buffer pointed to by *memPtr* which is *poolSize* bytes in length. A TRUE *diOpt* argument indicates that interrupts are already disabled. *memMgr* is a memory management structure that is filled in by these functions.

1.34.1.184 Return Value

Returns a pointer to a memory management structure of type *typeMem*.

1.34.1.185 See Also

getMem, putMem.

1.34.1.186 Example

```
#define NUM_MSGS 100
#define LEN_MEM_POOL 4096
typeFreeMsg MsgSpace[NUM_MSGS];
typeMem * MemPool;
char MemPoolSpace[LEN_MEM_POOL];

void main()
{
  typeTics * tics;

  tics = makeTics(MsgSpace, NUM_MSGS);
  startTics(tics);

  startTask(makeTask(taskA, 0));

  MemPool =
  makeMem(MemPoolSpace, LEN_MEM_POOL, 128);

  suspend();
}
```

putMem, _putMem

1.34.1.187 Function

Free a block of memory.

1.34.1.188 Syntax

```
void putMem(typeMem * * memPtr, void * blkPtr);
void _putMem(typeMem * * memPtr, void * blkPtr, int diOpt);
```

1.34.1.189 Remarks

These functions release the memory block pointed to by *blkPtr* back to the memory pool specified by *memPtr*. The memory pool must first be created by **makeMem**. *_putMem* is useful when memory must be freed from within an isr when interrupts are disabled. A TRUE *diOpt* argument indicates that interrupts are already disabled**.**

1.34.1.190 Return Value

None.

1.34.1.191 See Also

makeMem.

1.34.1.192 Example

```
#define DATA 1000
extern typeMem * MemPoolA;

msg = waitMsg(DATA);
processData(msg->pData);
putMem(&MemPoolA, msg->pData);
sendMsg(msg);
```

Index