An introduction to Tics

Adapted from The Tics RTOS Programmer's Guide, originally by Michael D. McDonnell

Michel Wilson

February 2, 2007

Contents

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. Tasks may send messages, wait for messages, 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.

2 Tasks

Each task is composed of three basic parts:

Code The code for the task is simply a C function, specified as an argument to the function creating the task.

Stack Each (non-cooperative) task has its own stack, for storing local variables. The stack size can be specified separately for each task. The default stack size is 1kB. Actual stack allocation is done upon starting the task.

Task Control Block Information about the task is stored in a C structure called the task control block (tcb). It contains things like the priority, task number, pointer to the task, and other information. It functions as the single source for all information about the task.

2.1 Task format

Tasks are C functions with the following format:

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

Tasks have a single argument which is unused (due to how task switching is implemented on the X32), and they never return. On entry, tasks perform initializations, if any, then enter an infinite while loop.

2.2 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.

2.3 Hello world example

```
#define NUM_MSGS 100
#define NUM_TASKS 10
typeFreeMsg MsgSpace[NUM_MSGS];
typeTcb TcbSpace[NUM_TASKS];
unsigned char StackSpace[NUM_TASKS*DEF_STACK_SIZE_IN_BYTES]

void taskA(int unused) {
  while (TRUE) {
    printf("Hello from task A\n");
  }
}

void main() {
  typeTcb * tcb;
  startTics(makeTics(MsgSpace, NUM_MSGS));
  startTics(makeTics(MsgSpace, sizeof(MsgSpace), StackSpace,
        sizeof(StackSpace), TcbSpace, sizeof(TcbSpace)));
  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.

2.4 Task control block

The task control blok contains several fields, some of which can be used to modify task attributes.

`flags` contains the task attribute bits. All bits are set to zero by `makeTask`. All flags can be modified at any time (before or after the task has been started), except the `COOP` flag.

- **`TIMESLICE`** Set to enable time-slicing (see 2.6.1)
- **`COOP`** Set before calling `startTask` to start the task as cooperative task (see 6)
- **`NO_STACK_CHECKING`** Set to disable stack checking on context switching.
- **`DO_NOT_ADD_TO_READYQ`** `startTask` will not add the tcb to the ready queue if this bit is set.

`pri` holds the priority for the task. Priorities range from `LOW_USER_PRI` to `HIGH_USER_PRI`. Lower numerical values indicate higher priority. See also 2.7.

`ptr` A general-purpose pointer which can be used to point to instance data by the programmer.

`timeSlice` Holds the length of the task's time slice in milliseconds. See 2.6.1.

2.5 Task instances

As described at the beginning of this chapter, a task is comprised of code, a stack and a tcb. Because each task has its own stack, a task can be created more than once. The task number or the `ptr` pointer can be used to differ between tasks, or to define instance-based data for a task.

2.6 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 (see chapter 3), issuing a pause (see chapter 4), 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` in order for this to occur; see chapter 7 for more information about ISRs), or its time-slice is used up and the time-slice option is in effect.
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.

2.6.1 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` or `waitMsg` functions. Time-slicing forces preemption which may not be desirable. The `pause` function allows tasks to voluntarily relinquish control without preemption.
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.
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 */
```

2.6.2 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` must be issued from within the ISR, see 7).
- 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.

2.7 Priorities

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. Task priorities should always be chosen relative to `DEF_PRI`.

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

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.

3 Sending and receiving messages

3.1 Sending a message

A message can be sent to a task using the `sendMsg` function, after creating it with the `makeMsg` function.

```
#define DATA 1000

typeMsg * msg;
extern typeTcb * TcbB;

msg = makeMsg(TcbB, DATA);
sendMsg(msg);
```

3.2 Waiting for a specific 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);
```

3.3 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.

3.4 Waiting for any 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);
```

3.5 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);
```

3.6 Freeing messages

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

3.7 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 (see chapter 4 for more information about timers) 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.

3.8 Responding to messages

To respond to a message, the sender task can be accessed using the `senderTcb` field of the message. Note that the sender tcb is saved, after which the message can be freed before sending a new message.

```
#define HELLO 1000
#define HELLO_BACK 1001

typeMsg * msg;
typeTcb * senderTcb;
msg = waitMsg(HELLO);
senderTcb = msg->senderTcb;
freeMsg(msg);
sendMsg(makeMsg(senderTcb, HELLO_BACK));
```

3.9 Canceling 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 msgSeqNum;
extern typeTcb * TaskA;

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

cancelMsg(msg, msgSeqNum);
```

3.10 Message numbers

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

4 Timers

Tics supports three types of timers: pause, one-shot timers, and periodic timers. Timers may also be cancelled.

4.1 Pause

The `pause` function can be used in-line to suspend program execution for a specified number of milliseconds, for example when accessing hardware.

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

4.2 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. Because of this, the same strategy as described in section 3.9 can be used to cancel the timer.

4.3 Periodic timers

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

```
void datamonitor(int unused)
{
  typeMsg * timer;
  timer = makeTimer(100L);
  timer->flags |= PERIODIC;
  startTimer(timer);
  while (TRUE) {
    freeMsg(waitMsg(TIMEOUT));
    processdata();
  }
}
```

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 datamonitor(int unused)
{
  while (TRUE) {
    pause(100L);
    processdata();
  }
}
```

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.

4.4 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`.

5 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.

6 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.
In this chapter, a small description of cooperative tasks is given. For more information, see the Tics Handbook, chapter 9.

6.1 Task format

Cooperative tasks have the following format.

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

6.2 Cooperative task characteristics

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.
- 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`.

6.3 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);
```

7 Programming ISRs

An interrupt is an external signal, forcing the CPU to transfer control to a different code section, the interrupt service routine (ISR).

7.1 Entering and returning from an ISR

Upon entering an ISR, the `isrEnter` function should be called, to tell the kernel that an interrupt is being processed. This disables the `sendMsg` function from preempting the ISR and delivering the message.
When the ISR is finished, tasks might have to be woken up because a message has been sent to them. This is accomplished by using the `isrRet` call as the last statement of the ISR. This function checks the priority of the interrupted task against the task at the front of the ready queue and returns to the higher priority task. If this function is not called, the ISR will return control to the interrupted task

7.2 ISR example

A basic ISR is shown below.

```
#define NEW_DATA 1000
 extern typeTcb * IoTcb;
void IoIsr(void) {
   typeMsg * msg;
   char ioChar;

   isrEnter();
   ioChar = inportb(0x50);
   msg = makeMsg(IoTcb, NEW_DATA);
   msg->iData = ioChar;
   sendMsg(msg);
   isrRet();
}
```

A Porting Tics to the X32

In this section, a small description is given on how Tics was ported to the X32 platform.

A.1 Interrupt handling changes

Since the interrupt handling on the X32 differs significantly from that on the x86 platform, some changes were made. In the MS-DOS version of Tics, care must be taken when calling kernel functions in an interrupt handler, since interrupts are disabled inside the ISR. This is not the case on the X32, to preserve realtime responsiveness. The `diOpt` flag of the kernel functions should thus not be set to `FALSE` when calling from an ISR. In fact, the whole `diOpt` flag should be removed, and all kernel functions should be rewritten to use the `lock`/`unlock` functions provided by the X32 library instead of disabling/enabling interrupts around critical sections. This however, presents a substantial change to the code for time was not available yet. A consequence of this is that the user must be aware of the fact that interrupts can sometimes be disabled inside a kernel call, for a short amount of time, which can affect the realtime responsiveness of the system.
To prevent user error, the `_sendMsg` and other functions using the `preeOpt` parameter[1] were changed such that this parameter is no longer necessary when calling them from an ISR. A global flag (`canPreempt`) was introduced for this, which is set by `isrEnter` and cleared by `isrRet`. This enables the user to use the normal forms of these functions instead (i.e. `sendMsg`).

A.2 Hardware support file

The supplied `hs_msdos.c` hardware support file for the MS-DOS operating system was used as a basis for creating a hardware support file for X32, `hs_x32.c`. The changes which were made are described below.

A.2.1 Timer ISR

The timer ISR is virtually unchanged. MS-DOS-specific code was removed, and `canPreempt` is set to `FALSE` during execution of the ISR.

[1] `_sendMail`, which is deprecated, and `_wakeup`, which is never used, actually

A.2.2 ISR return function

A function for returning from an ISR was added, `isrRet`, as used in the X32 uC/OS port as well. The stack frame is inspected to see if control is returned to another ISR or to a normal function. In the latter case, `canPreempt` is set to `TRUE`, and the Tics macro `_isrRet` is called, which checks if there is a higher-priority task in the runqueue. If a higher-priority task is found, the current task is preempted.

A.2.3 Timer initialization

The `initTimer` function is rewritten to use the (first) X32 timer. The interrupt priority of the timer interrupt is set to 1. Other interrupt handlers should use a higher priority than 1, to make sure that they cannot be interrupted by the timer interrupt.

A.2.4 Exit funtion

The `exitTics` function disables the timer interrupt and the global interrupt, and returns control to the bootloader.

A.2.5 Task switching

In the `suspend` function, task switching is done. The x86 stack save code is removed, and the X32 functions `_get_fp` and `_set_fp` are used to save and restore the task context to and from the tcb.

A.2.6 Stack initialization

To create a new task stack, the `initStack` function is used. Stack space is allocated in an array in the `tics` structure. The stack space must be allocated by the user, and is supplied to the `makeTics` function before the operating system is started. Initialization of the stack is done like in `init_stack` in the X32 library. A pointer to the top of the stack is saved in the tcb for context switching.

A.2.7 Hardware initialization

There's not much hardware to initialize on the X32—the timer frequency (in milliseconds) is determined and saved in `MsPerTimerInt`, the global stack pointer is initialized and the timer is initialized.

A.2.8 Fatal error handler

A custom error handler is included in `hs_x32.c`, which prints the error message to the standard output (which is the serial port). This function is used as a default error handler in the `makeTics` function. The user can of course also write his own error handler, which can perform some additional functions.

A.2.9 Tics initialization function

The X32 port has its own default error handler, which is added to the `tics` structure here. Also, the count for the timer chips is slightly different from the MS-DOS version.