Real Time Operating Systems in a High-level Language 'C/C++': Pitfalls and Possibilities

Tony McCall

The Open Polytechnic Working Papers are a series of peer-reviewed academic and professional papers published in order to stimulate discussion and comment. Many Papers are works in progress and feedback is therefore welcomed.

This work may be cited as: McCall, Tony. *Real Time Operating Systems in a High-level Language 'C/C++': Pitfalls and Possibiities*, The Open Polytechnic in New Zealand, Working Paper, December 2002.

Further copies of this paper may be obtained from

The Co-ordinator, Working Papers Advisory Panel
The Open Polytechnic of New Zealand
Private Bag 31 914
Lower Hutt
Email: WorkingPapers@openpolytechnic.ac.nz

This paper is also available on The Open Polytechnic of New Zealand website:
http://www.openpolytechnic.ac.nz/

Printed and published by The Open Polytechnic of New Zealand, Lower Hutt.

ISSN — 1174 – 4103

ISBN — 0 – 909009 – 54 – 6

Working Paper No: 7-02

A list of Working Papers previously published by The Open Polytechnic is included with this document.

Abstract

The process of designing a Real Time Operating Systems (RTOS) can be fraught with difficulty. This research sets out to test the validity of using a high level language such as the Borland C/C++ programming language to write the RTOS Kernel and the tasks that will run with that operating system. This inductive study examines some of the major RTOS components and seeks to demonstrate how they can be coded into the C/C++ language. During that process, the problems encountered, and possible solutions, are documented. This enquiry was based on the 80x86 range of processors due to their popularity in large-scale embedded control applications.

Contents

Acknowledgements

Thanks goes to Dr Yaa Li and Parwaiz Karamat for their comments and review of this material, and to Richard Drummond for editing it.

Real Time Operating Systems in a High-Level Language 'C/C++': Pitfalls and Possibilities

Introduction

A Real Time Operating System is an operating system that must respond to tasks and external events within a critical timeframe. Real Time Operating Systems are commonly used in the following areas:

- Process control
- Automotive
- Computer peripherals
- Office automation
- Communications
- Robots
- Aerospace
- Domestic appliances.

Single tasking operating systems are most easily developed in real time as there are no shared resources that include memory, real time systems interrupts and i/o. Only one task may be executed at a time.

Real-time multitasking operating systems may run several tasks simultaneously if they are run on a system that contains multiple processors. An example of this would be a hard disk drive, where a separate CPU is used to run the hard disk that is operating on data from a shared memory area while the system CPU is performing other tasks. Usually, however, the need for a multitasking OS is determined by the extent that these multiple tasks, all running simultaneously, need to interact or use shared resources. If no shared resources are required and no interaction is required, then depending on the cost, multiple single tasking operating systems running on completely independent hardware may be more stable.

The term *real time* relates to the need to process information and respond to interrupts within a given time. The response of the OS to external events and regularly occurring internal events is critical. External events trigger interrupts, and these interrupts must be responded to within a specified time. That time, termed *interrupt latency*, is the time it takes a CPU to respond to an interrupt. There is no absolute standard for determining this time. David E Simon's (1999) describes this interrupt latency as the total of the following tasks:

1. The longest period of time during which that interrupt is (or all interrupts are) disabled.

2. The period of time it takes to execute any interrupt routines for interrupts that are of higher priority than the current one.

3. How long it takes the microprocessor to stop what it is doing , do the necessary bookkeeping , and start executing instructions within the interrupt routine.

4. How long it takes the interrupt routine to save the context and then do enough work such that what it has accomplished counts as a response. (Simon, p104)

Interrupt latency is best tested by timing snippets of code as it runs. The alternative is to use the manufacturer's documentation. Using the CPU's response timing, clock speed and the number of cycles taken to complete each instruction in the interrupt service routine the latency time can be estimated. This does not work so well for CPUs that cache instructions before they are ready to be used. Since cached instructions do not have to be fetched from system memory, the time to execute is considerably shorter. Whether the next instructions are cached, resulting in a cache hit, or not, will make the latency time vary. The worst case situation should be assessed.

The CPU responds to an interrupt by first storing the current state of its registers on the stack. A memory address is then looked up in an interrupt table, and that address is used to run the interrupt routine. The interrupt routine contains the code required to carry out the required task. Once the interrupt routine has responded to the device requiring attention, it terminates with an interrupt return instruction. The interrupt return instruction causes the CPU to reload the registers from off the stack in reverse order and continues executing the program that was running before it was interrupted.

The 80x86 family of processors has a flag register which keeps track of the current status of the CPU. Among these flags are two flags that are used with interrupts: the trap flag (TF) and the interrupt flag (IF).

The address of the next instruction to be executed is held in the instruction pointer register (IP). This register holds the address of the next instruction within a memory segment of 64 Kbytes. Another register, called the *code segment register* (CS), holds the segment address, that is, which 64K block of memory to look in. The code segment and the instruction pointer work together to give the effective address in the form CS:IP. Now when an interrupt occurs, only these flags and registers are saved, unlike other processors that save all their registers on the stack. It is the programmer's responsibility to store these additional registers and restore them as required. When an interrupt occurs, the flags are saved , the TF and IF flags are cleared, the current CS is pushed (saved) on to the stack , the CS for the Interrupt Service Routine (ISR) is loaded from the interrupt vector table, the current IP value is then pushed onto the stack, and the new IP address is loaded from the interrupt vector table. Execution then continues at the newly loaded CS:IP location until an interrupt return instruction IRET is encountered. The IRET instruction then pops the values of IP, CS and the flags register off the stack. They are retrieved from off the stack in reverse order to the way in which they were stored on the stack.

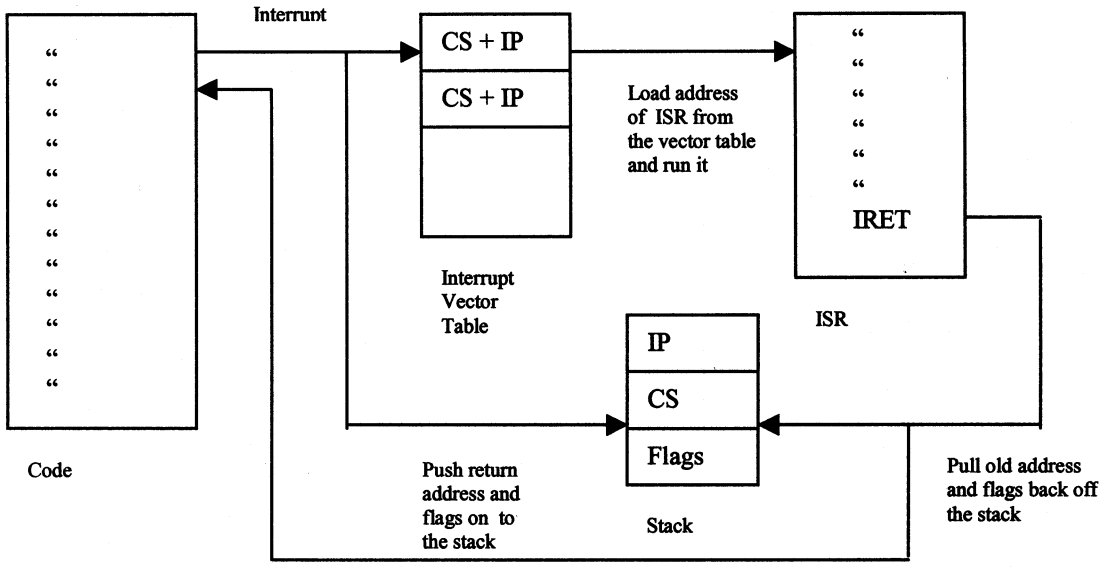

Fig.1 *Interrupt Operation*

Fortunately, the Borland C++ compiler and many other DOS based compilers provide a crucial element of support for interrupt routines, the interrupt type modifier. You are able to declare a function to be of type **interrupt**. You can then use this function as an interrupt service routine since it automatically preserves all the registers on the stack except for the Stack Pointer (SP) and Stack Segment (SS) and terminates the routine with the required IRET instruction. To use **interrupt** modifier, the exact order in which the registers are

stored must be known. Remember that the Flags CS and IP registers have already been stored on the stack before the interrupt modifier function is called. The modifier function then pushes the rest of the registers on the stack in the following order:

1. push ax
2. push bx
3. push cx
4. push dx
5. push es
6. push ds
7. push si
8. push di
9. push bp.

The stack, after entering the ISR, will be :

BP ← Stack pointer

DI

DS

ES

DX

CX

BS

AX

IP

CS

FLAGS

While mission critical code is running, that is, code that cannot be interrupted , all interrupts must be disabled and subsequently re-enabled. Fortunately, C/C++ has two functions that allow the disabling and enabling of all interrupts except non-maskable interrupts (NMI). These are disable() and enable(). The code would look something like this:

```
void interrupt handler (dosomething) /* */
{
/* disable interrupts during the handling of the interrupt */
 disable();
/* code to do something goes here */
/* re-enable interrupts at the end of the handler */
 enable();
}
```

Should another task request service during this time, it must wait for the interrupts to be re-enabled. This additional waiting time increases the interrupt latency time of the second task. While the number of pending tasks can be reduced by disabling interrupts, they can't be eliminated. On most microprocessors, there is no way to disable interrupts quickly enough while inside the ISR to prevent further interrupts from becoming pending. Pending interrupts must themselves be responded to quickly enough for the event that caused that interrupt to be serviced in time. Three observations can be made from this:

1. All interrupt routines should be short.

2. Interrupts should be disabled for the shortest possible time.

3. Alternatives to disabling interrupts should be used where possible.

Most interrupts are asynchronous. That is, they can occur indiscriminately at any time. Other tasks are synchronous, needing regular and deterministic service. Both the synchronous and asynchronous events must be managed. This is the job of the *kernel*, which is that part of the operating system responsible for managing tasks. The kernel's fundamental function is to switch between running tasks. Other functions performed by the kernel include messages, semaphores queues, mailboxes pipes and time delays. The two major software components of the kernel are:

1. The context switch, or task switch that switches between tasks, and

2. The scheduler, also called the *dispatcher*, which is responsible for determining which task will run next.

The context switch must store the current tasks context before switching to the next task. The context is the last state of the CPU as recorded by its registers. Before the task is switched, a copy of all the processor's registers are made stored and their position recorded in a memory table called a *Task Control Block* (TCB).

Often the registers will be stored on the stack in a stack frame, and the TCB will contain the stack pointer (SP) to that tasks registers. See Fig 2. The next time the processor returns to this task it will re-load the registers from the position pointed to by the task control block and then begin execution of the task exactly where it had left off before it was interrupted. The processor will continue executing the code from the instruction immediately after the last instruction that had been executed before that task was pre-empted.

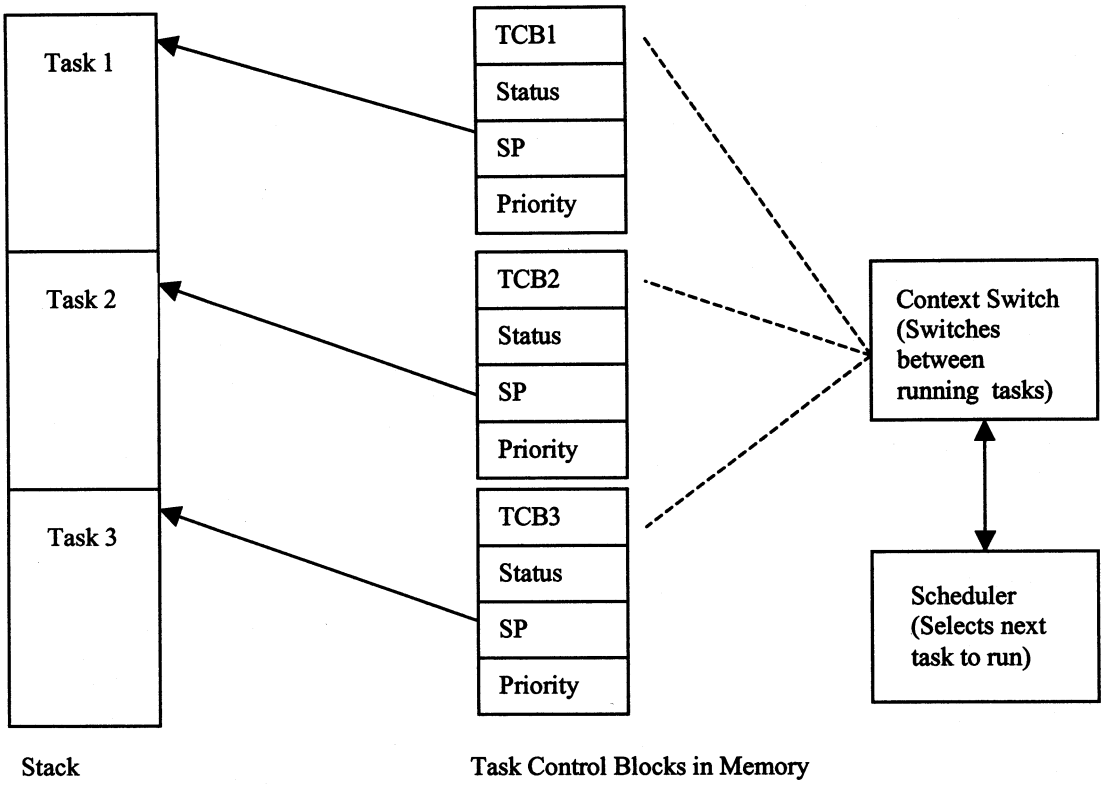

Fig. 2 *Multitasking*

The scheduler determines which task runs next, based on a priority scheme. A priority scheduler will give priority to the highest priority task that is pending service. A non-priority scheduler will execute the task that has been waiting the longest for service. The scheduler also takes into account the status of the task; Ready, Running or Blocked. Tasks running in a multitasking environment may elect to block themselves while they are themselves waiting for some event, such as a character to be received on the RS232c port.

Scheduling disciplines

While the variation in scheduling disciplines is infinite, the goals of scheduling are very much the same. What follows is some scheduling disciplines that are representative of the goals of most schedulers. When considering a scheduling discipline, real time goals must be observed, as multitasking operating systems do not necessarily respond to events in real time.

A scheduling discipline can be either *pre-emptive* or *non pre-emptive*. A scheduling discipline is pre-emptive if the CPU can be taken away from the process. It is *non pre-emptive* if it cannot be taken away from the process.

Pre-emptive scheduling allows priorities to be set so that higher priority tasks can gain the immediate attention of the CPU. This may be required in some real-time systems where the consequence of missing an interrupt could be devastating. In pre-emptive multitasking, interactive time sharing occurs between tasks. Each task is given a time slice, depending on its priority. This can be a useful method of guaranteeing acceptable response times. The scheduler must arbitrate between tasks, and the context switch must constantly switch between tasks as scheduled by the scheduler. Any time where the CPU is doing no processing, just switching between tasks results in an overhead or time cost. There are also memory costs associated with pre-emptive scheduling, since all tasks must remain in memory all the time to ensure that they are ready to run when the CPU next becomes available. A further disadvantage is that the priority scheme is arbitrarily designed and may not be effective in all situations, particularly where there are periods of overload on the services required from medium or low priority tasks. To counter all possible scenarios, the pre-emptive scheduling discipline can become overly complex, and considerable care is required to keep it simple and therefore effective.

Non-pre-emptive scheduling disciplines are inherently much simpler. The next task cannot start until the existing task has been completed or has allowed another task to start. The response times are more predictable because high priority tasks cannot displace low priority tasks. The disadvantage is that short tasks have to wait for longer tasks to complete, but overall, each task gets equal access to the CPU. Non-pre-emptive multitasking fails in real time systems when a critical task is restricted from responding to an event because the CPU is processing another task.

Two particular types of scheduling are worthy of note, first-in-first-out scheduling and round robin scheduling. Both of these scheduling disciplines use queues. Tasks requesting service are put into a queue so that their request is

not forgotten. This queue is usually a first-in-first-out-queue. (FIFO)

First-in-first-out (FIFO) scheduling is a non-pre-emptive scheduling discipline where the tasks requiring CPU time are loaded into the queue in the order they requested attention. The processor then processes each task to completion in the order they arrived. See Fig. 3 below.

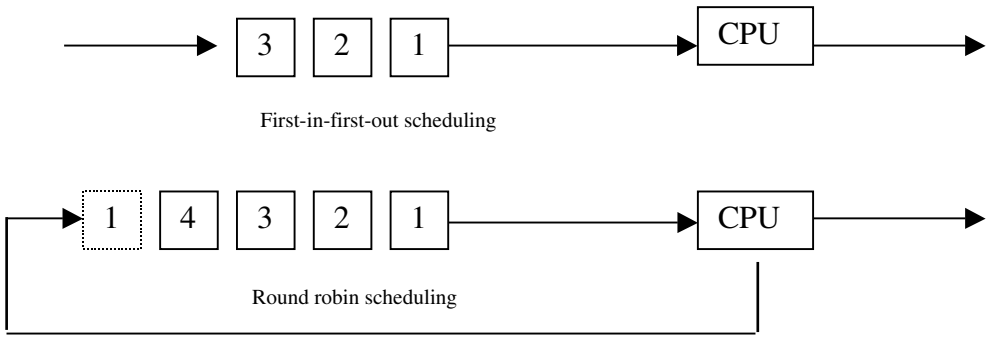

Fig. 3 *Scheduling Disciplines*

Round robin scheduling is a pre-emptive scheduling discipline where each process is given a limited amount of time called a *time-slice* or *quantum* . The tasks requiring CPU time are again stored in a FIFO queue. Each task is either run to completion or that task is suspended, not finished, and placed back at the beginning of the queue to await another time slice. The processor is made available to each task in order, for the duration of its time slice. Larger tasks need to stay in the round robin queue until they receive enough times slices to run to completion.

As the quantum given to each task becomes progressively smaller, the time taken to perform the context switching becomes a greater part of the overall time spent processing tasks. If the CPU is spending most of its time switching between tasks, there is little or no time spent processing each task. An optimum quantum must be selected somewhere on the continuum between the CPU doing very little context switching and the CPU spending most of its time switching between tasks rather than processing tasks. At both of these extremes, performance is sluggish.

Several scheduling disciplines have been developed to reduce the time that tasks remain in the queue waiting for CPU time. These revolve around the possibility of completing jobs that are either short or have little remaining time to run.

The *shortest job first* (SJF) strategy allows the shortest tasks to be processed first . This will remove these jobs from the queue, the goal being to reduce the number of tasks waiting service and thereby reduce the context switching time and scheduling time overhead on the remaining tasks. This strategy effectively reduces the average waiting time for all tasks in the queue because removal of the shortest tasks from the queue once complete, will reduce the overall context switching time. SJF is a non-pre-emptive scheduling strategy.

The pre-emptive counterpart of SJF is *shortest remaining time* (SRT). In SRT, the task with the least remaining time to completion is run next. One goal is the same, that is, to complete shorter tasks so that the average wait time is reduced overall. The other goal is to finish processing larger tasks that have little remaining time to run , which will also reduce the average overall waiting time. Since this is a pre-emptive process, time slices of each task are run until the SRT task is completed from the round robin queue. One disadvantage is that the time to run and elapsed time have to be recorded for each task, increasing the scheduling overhead. There is further overhead in determining the next task to run. A simpler scheme is to set a threshold whereby any task that is below this time-to-run threshold is completed without task switching.

The last scheduling strategy to be considered here is *highest-response-ratio-next* (HRN) scheduling.

SJF scheduling creates a bias against the running of longer tasks and a strategy is required to not only remove the shorter jobs from the queue earlier but also account for longer tasks that could be kept waiting. HRN sets the priority of each task according to the time the job has been waiting in the queue and its time to run, called the *service time*.

$$\text{Priority} = \frac{\text{time waiting} + \text{service time}}{\text{service time}}$$

This scheduling strategy dynamically alters a task's priority according to the time it has been waiting. However, once the task gets to the CPU, it is run to completion. A closer look at the formula reveals how this strategy is implemented. Because the service time appears in the denominator the priority of shorter jobs is increased, reducing the mean response time to all tasks. However, since the time waiting appears in the numerator, larger tasks will eventually be run when this value accumulates. This strategy prevents larger tasks from being constantly blocked by new and shorter tasks joining the queue (Hansen 71 in Deitel p258).

More elaborate multitasking strategies use a succession of priority-based queues. These strategies come out of the need for a scheduling mechanism to

- favour short tasks
- favour i/o bound tasks to get good i/o device utilisation
- determine the nature of a task as quickly as possible and schedule the job accordingly (Deitel p259).

Smaller and i/o bound tasks are usually completed in the first queue. However, if a task is waiting on an i/o response it is completed and must rejoin the queue when that i/o event has occurred so that the other tasks do not have to wait for this i/o transaction to be completed. If a task is not completed in the first high-priority queue, it drops down to the next lower-priority queue and is rerun in that queue. In this way, larger tasks of lower priority keep dropping down through lower-level priority queues until they receive enough time slices to complete. Typically, the lowest priority queue is a round robin queue that ensures that all remaining jobs are completed. Refer to Fig.4.

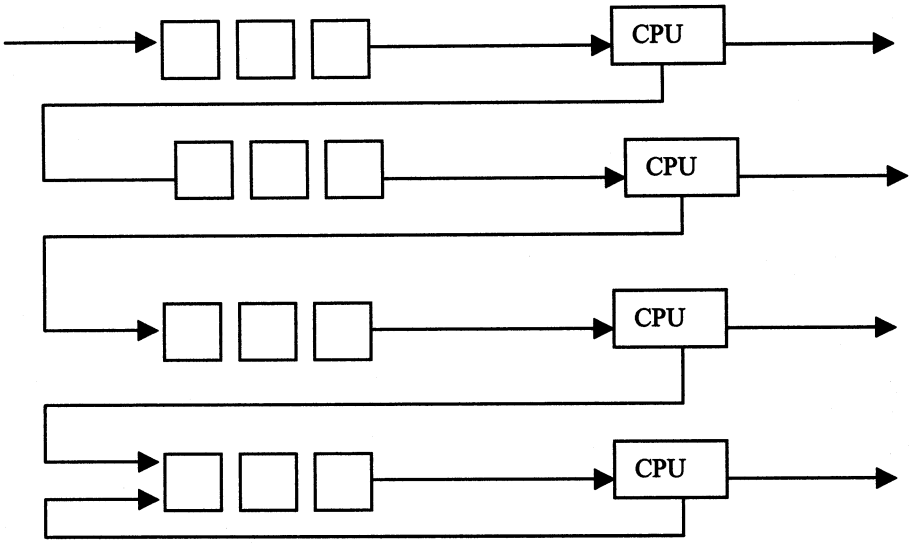

Fig. 4. *Multiple queues*

The FIFO queue can be implemented in C/C++ using an array of pointers. Using pointers in the queue rather than just the pure data allows different data types to be mixed, since a void pointer in 'C' can be cast to any required data type as need be. Rather than the data itself being listed in the queue, the pointers are placed in the queue, and these represent the data.

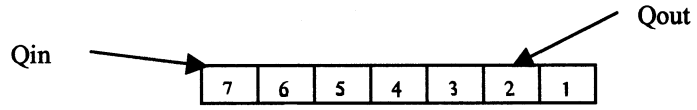

Fig. 5 *FIFO queue*

In Fig. 5, *Qin* has placed an additional task at the beginning of the queue (the head), while *Qout* has taken the first item (or task) from the tail of the queue and is now retrieving the second item deposited in the queue. This process would work well if the queue didn't continue to creep through memory as each new item was added. To prevent this, a circular buffer is used.

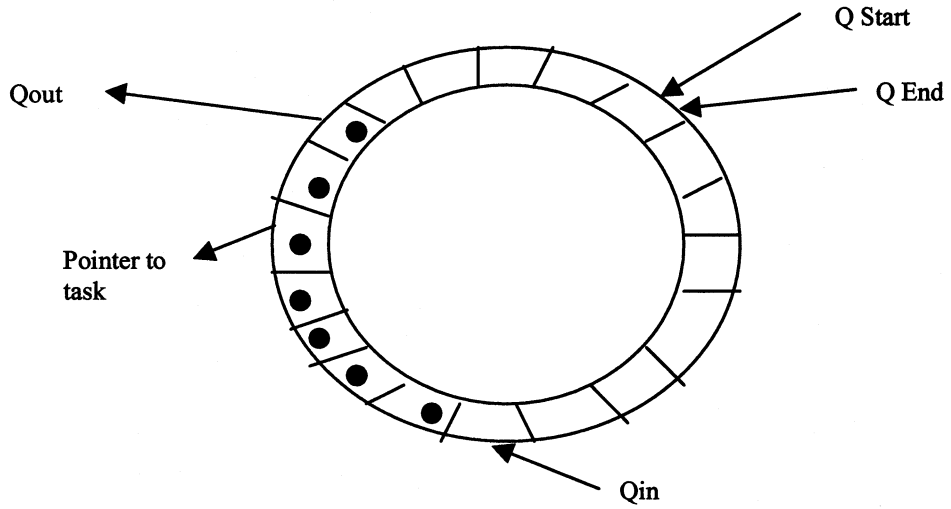

Fig. 6 *Circular buffer representing a FIFO queue*

The following definitions apply to the circular buffer shown in Fig.6.

QStart = pointer to start of queue
Qend = pointer to end of queue
Qin = pointer to the next item to be inserted
Qout = pointer to the next item to be extracted from the queue
Qsize = the size of the queue
Qentries = the number of entries in the queue.

Given the definitions of the parameters for the circular buffer given above some snippets of code show how this circular buffer may be implemented in C. Two functions may be constructed: one to put an item in the queue *(Qput),* and the other to get the item off the queue *(Qget)*. The C code would be similar to this:

```
Qput(void * data) /*Function puts data in the queue */
{
disable();  /* stop maskable interrupts */
if (q->Qentries >= q->Qsize) /* check that the queue is not full */
 {
       return Q-full;
 }
 else
 {
      *q->Qin++ = data;/* data pointer in next available location */
       q->Qentries++; /* update the number of entries */
 }
if(q->Qin++ == q->QEnd)
      {
      q->Qin== q->QStart; /* if at the end of the queue wrap around */
 }
enable() /* enable interrupts
}
```

To get data back out of the queue, code similar to the following function can be used:

```
void Qget() /* function gets data from the queue */
{
disable()
if (q->Qentries !=0)
      {
      data = q->Qout++;           /* get oldest data pointer */
      q->Qentries—;               /* delete that last entry */
      }
if (q->Qout==Qend)
      {
 q->Qout = q->Qstart; /* if at end of queue wrap around to the beginning */
       }
else
      {
      data = (void*) 0; /* queue is empty so return Null pointer */
      return (data);
       }
enable()
}
```

Circular buffers are used in his way to form the FIFO queues required to implement a scheduling strategy. They are also used for inter-task communications where one task drops a message in the mail box for another task to get. The functions in either of these cases, while more elaborate, are based on the simple circular buffer routines previously presented. The C/C++ languages' ability to use pointers makes it ideal for these applications, since the

FIFO queue need only store a pointer to the data (task) rather than the data itself. While the round robin scheduling strategy can also be implemented using a FIFO queue, it can be implemented much more easily than that, another reason for its popularity. When the embedded project is designed, the tasks can be arranged to simply call each other in sequential order (daisy chain) using pointers. See Fig. 7.

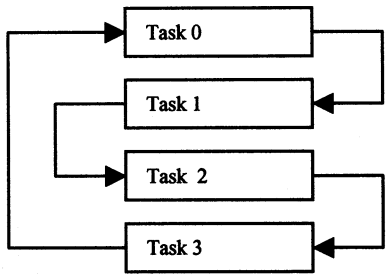

Fig.7 *Tasks formed into a simple Round Robin Queue using pointers*

To run a round robin OS, a circular queue is not required, as a circular buffer can be formed using pointers. Consider that each task has its own task control block TCB that looks after the context of that task. To set up a circular buffer to run these tasks, the following snippet of C/C++ code can be used where the pointer .next for one task points to the next task until the end is reached and the pointer points back to the beginning again.

```
TCB[0].next =&TCB[1];
TCB[1].next =&TCB[2];
TCB[2}.next =&TCB[3];
TCB[3].next =&TCB[0];
```

The tasks

So far, I have covered the scheduler from the perspective of the scheduler discipline. However, in a RTOS, the tasks themselves can be placed in any one of a number of states. In any RTOS the task must be able to be placed in a minimum of three states which are controlled to some degree by the scheduler.

- Running — currently being processed by the CPU
- Blocked — stopped from running until some other event occurs first
- Ready — the task is ready to run the next time it is given the priority.

The exceptions to this are some non-pre-emptive multitasking systems that do not allow any task to be blocked and all tasks to be run to completion. Figure 8 below shows the states that can be adopted by a task.

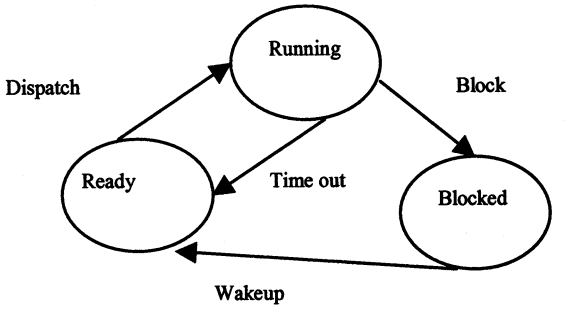

Fig. 8 *The three states a task can assume in a minimal operating system*

A task will block only because it decides to block itself, which it should do if it has nothing else it needs to do. For a task to block itself it must have first been running. When tasks voluntarily give up processing time they are engaging in co-operative multitasking, since the tasks voluntarily give up their right for further processing time. Co-operative multitasking was the method used in the Windows 3.1 kernel. This operating system was vulnerable to ill-behaved programs hogging processing time and not releasing memory resources when ending. In fact, programs could appear more responsive to the user if they didn't voluntarily give up processing time.

When a task is blocked, it never receives the CPU. Therefore some external interrupt or other event is needed to unblock the task. This external signal may come from another task that signals that the event the blocked task was waiting for has happened.

```
/* Block a task */
 void Block (int id)
 {
     if (tasks[id].status == RUNNING)
 tasks[id].status = BLOCKED;
 }
```

A task receives a wakeup when it is taken out of the blocked state and put back in the ready state.

```
/* Wakeup a blocked task that has been waiting a specified event */
 void Wakeup(int id)
{
      if (tasks[id].status == BLOCKED)
 tasks[id].status = READY;
 return;
}
```

A task has timed out if it has been pre-empted and placed back in the ready state.

```
/* Time out a task until resumed by the scheduler.*/
void Timeout(int id)
{
    if (tasks[id].status == RUNNING)
tasks[id].status = READY;
}
```

It is the task of the scheduler to move a task from its ready state to its running state. The scheduler may prioritize the tasks waiting to run using any of the methods previously described , including running the next available task in the case of a FIFO or round robin scheduler.

```
/* Dispatch a task */
void Dispatch (int id)
{
    if (tasks[id].status == READY)
tasks[id].status = RUNNING;
}
```

Partitioning

The code to be run on a multitasking RTOS must be partitioned into smaller tasks that are each time- and event-dependent. The usual goals of top-down modular design apply where each task is loosely coupled to other tasks using pass-by variable interfacing between them and good cohesion where each task performs one function completely. The major difference is the low level interfacing to the hardware. The hardware should be selected to reduce the complexity of the software.

Some major strategies reduce the programming effort required to bring an embedded project to market. The overriding strategy is to reduce both the number of lines of code and their complexity. Barry Boehm, who has developed the Constructive Cost Model (COCOMO), found that the programming effort to create a project can be found as follows:

$$\text{Programming Effort} = C*KLOC^{M}$$

Where:
KLOC means 'Thousands of Lines Of Code'
C is a factor that depends on the costs of the code
M is a factor that represents the difficulty of development.

For real-time programming, Boehm gives M a value of 1.2, which means a large exponential increase in effort for each extra line of code. Partitioning the code into smaller tasks, each having fewer lines of code, will reduce the overall programming effort. Coding complexity is also reduced when hardware is carefully selected to minimize the coding complexity represented within the exponential component M.

The C programming language allows partitioning through separate source files that are combined at link time and through the use of functions. C functions are able to receive a number of parameters but are only able to return one parameter. While the latter is certainly a restriction, correct modular design dictates that there should be only one entry point and one exit point to each module. In assembly language, the stack can be used to good effect to pass any number of arguments to or from a module. C++, being a superset of the C language, allows encapsulation (formally). Hidden detail (possibly of a hardware specific nature) can be encapsulated as objects. A simple interface is then made available to the object that allows other code to use the object without regard to the hidden detail. Other functions or tasks are prevented

from using the code within an object except through the defined interface. If more complex code is encapsulated in this way, the overall project code is simplified. Other features of C++ include inheritance and polymorphism. Through inheritance and polymorphism, lower-level objects can be used to develop higher-level objects, allowing increasing levels of abstraction. Less formally, the C language also allows similar simplification and abstraction using structures, and the keyword structure, as structures may be nested in C. The same options are also less formally available using assembler. David Simons, along with many other embedded programmers, believes C++ to be overly complex. 'However, one of the acknowledged disadvantages of C++ is that it is a complicated language, in many ways much more difficult and subtle than C.' (Simons, pXV1).

David Parnas' partitioning strategy (Parnas p125 in Laplante 92) advocates making a list of difficult design decisions and parts of the project that are likely to change. Modules are then designed to hide this complexity from the rest of the system. Only functions of these modules are then made available to the rest of the system. The goal is that, when changes are made to these modules in the future, they will not affect the rest of the system. These modules may even be coded later, when the full solution to some preliminary design decisions becomes available.

Partitioning cannot only be done in software as previously described but also in hardware using multiple processors. Larger programs can be split across a number of processors, allowing detailed hardware requirements to be encapsulated. The emergence of cheap microcontrollers and the use of the I2C serial bus makes this much more practical.

Partitioning in a real-time system also needs to take account of response times. Each module of code, which may later become a task, will have a different time/response requirement. Program code that operates a keyboard could conceivably include

- a keyboard scan task (interrupt driven from port)
- a keyboard debounce task (20 ms after keyboard scan task)
- a keyboard buffer task (called by keyboard debounce task if debounce successful)
- a keyboard look up table task (can be run asynchronously during idle processor time).

Correct partitioning will also reduce the dependency on global variables and allow the use of local variables. Local variables allow modules to be re-entrant as is discussed next.

Some partitioning rules can be concluded from the above assertions:

- Partition for good cohesion and loose coupling.
- Reduce the lines of code by breaking them down into smaller, more manageable tasks.
- Reduce the complexity of the code by selecting hardware that minimizes the complexity.
- Encapsulate complex code and provide a simple interface to it.
- Place difficult design decisions and code that is likely to change in a single module.
- Consider using multiple microcontrollers or processors to eliminate complexity in large programs of more than a thousand lines.
- Identify response and timing constraints, and partition accordingly.
- Partition to minimize global variables.

Re-entrant code

If a task is run to completion in a non-pre-emptive scheduler, the task will not be interrupted. However, if the task is to be run using a pre-emptive scheduler, it must be capable of being interrupted, and once interrupted, it must be able to be started again without losing data or function. For this to happen, the code must be *re-entrant*.

A re-entrant function is a function that

- can be used by any task without any chance of data corruption.
- uses only registers and variables that are stored on the stack.
- protects data in global variables if these have to be used.

The golden rule is to avoid using global variables and static variables. If they are used then special measures are required to ensure that only one task has access to these at a time. Use of such measures is called *serialisation*. Serialisation allows only one task or function to use the protected code at a time (serially). Protecting code or data in this way is called *encapsulation*. Simons describes three ways of protecting code and data through encapsulation (Simons 1999, p168).

1. *Disabling and enabling interrupts:** A very fast method since the processor will perform the disabling or enabling of interrupts in one instruction. The disadvantage is that all activity controlled by the RTOS is stopped between the disable() and enable() statements.

2. *Semaphores:* Affects only those tasks that take the semaphore and the data that is protected by the semaphore. This method allows other tasks to continue to run in the RTOS. This is a much slower method but well targeted because it stops access to the protected code or data only.

3. *Disabling the task switcher*:* Disabling the task switcher will only stop other tasks running that might otherwise use the shared resource. It will not stop interrupt routines from being run.

---

* Note that 1 and 3 can be the same if, as is usually the case, the task switcher is controlled by a maskable timer interrupt.

Semaphores can be used to carry out the serialization of the code. The term *semaphores* is historical. You can think of them as flags. For example, a train may only leave the station if the conductor flags that everyone is now on board. Alternatively, the train can only enter the next section of track when the flag/ semaphore signal is asserted. Taking the analogy further with a different example; only one train can enter a shared section of track at a time when signaled using a semaphore (serialization). Both trains can't enter at the same time without mishap, just as two programs can't use the same data storage or common code at the same time without mishap.

Pre-emptive multitasking systems rely on the processor storing all of its registers and variables on the stack when a task is completed, before switching to the next task. Several CPU instructions will do this, including return from interrupt and software interrupt instructions. These instructions cause the CPU to store a copy of its registers and local variables on the stack. The exact whereabouts of these registers and variables is pointed to by the stack pointer. Hence the stack pointer must be saved in that task's TCB. When the CPU next returns to this task, it will be programmed to load the old value of the stack pointer back from the task control block. The task switcher then uses this stack pointer to restore the values and registers back off the stack, allowing the CPU to continue processing the task from the point it was pre-empted. The use of the stack in this way prevents that task's variables from being overwritten and ensures that it remains a re-entrant function. Because functions always store their local variables on the stack, functions should be used as much as possible to encapsulate code that is used in the tasks. Functions allow the use of local variables that will be stored on the stack during an interrupt, so there is no chance of another task changing these variables in the meantime.

When the scheduler blocks a low-priority task midway through its execution and begins the execution of a higher-level task, there is a danger that the CPU has interrupted the processing at a point that is not re-entrant. Calls to DOS are, for example, non-re-entrant. A construct such as a loop or selection may have been interrupted and a variable may have been changed before the scheduler has returned to execute the rest of the code. Jean J Labrosse (Labrosse, p69) offers this example in the C/C++ language:

```
int Temp;
void swap(int *x, int *y)
{
 Temp = *x;
 *x = *y;  // task interruption at this point
 *y = Temp;
}
```

In this code two pointers are passed to the function 'swap' and the values which they point to are supposed to get swapped. Unfortunately, only the value of pointer x is swapped into global variable Temp in the first instruction before the CPU is dispatched to a higher-level task. Now suppose this higher-level task also uses the variable Temp, and in the course of its execution overwrites this variable's value from 4 to 6. The next time the CPU begins processing this lower-level task, it will take the last value of Temp (6 not 4) and place it in the memory location pointed to by pointer y, yielding the wrong result. The sequence of events that caused this bug may only occur a couple of times in a year. The bug will only show up occasionally when the CPU interrupts this lower-level task at this exact point to execute the exact snippet of code in this particular higher-level task. Imagine how difficult this code would be to debug. This is the inherent problem with writing tasks for pre-emptive RTOS.

This code can be made re-entrant by declaring Temp to be a local variable. The parameters *x and *y will also be stored on the stack along with Temp so that this code will now be re-entrant. See the amended version below.

```
void swap (int *x, int *y)          // parameters are stored on the stack
{
int Temp;       // now local and stored on the stack
      Temp = *x;
      *x = *y; // task interruption at this point
      *y = Temp;
}
```

In any high-level language that is used for multitasking, a study must be done of the way the language stores its variables so that the programmer is aware of the re-entrancy problems that these can cause.

Many high-level languages have built-in functions that are not re-entrant . For example, the getche() function in the C/C++ language is not re-entrant. Care must be taken with any high-level language to check out which inbuilt functions are not re-entrant and which variables are not stored on the stack, because they must be either encapsulated or avoided.

Two methods can be used to make functions re-entrant if they are not inherently re-entrant:

1. Semaphores can be used to prevent another task using this shared data area or shared code.

2. Atomic code can be used to ensure that this code cannot be interrupted during the critical time it is using the shared code or data storage.

Some simplified examples are given next to illustrate the two methods:

```
/* Semaphore version of getche(). */
char sgetche(void)
{
 char ch;
 set_semaphore(&out); */ set semaphore
 ch = getche();
 clear_semaphore(&out);          */ clear semaphore
 return ch;
}
/* Atomic version of getche(). */
char sgetche(void)
{
 disable();                      */ disable interrupts */
 char ch;
 ch = getche();
 enable();                       */ enable interrupts
 return ch;

}
```

Of the two methods given above, the semaphore method is the most economic, since it allows multitasking to continue and does not slow down the operation of other tasks. However, since the second method disables all maskable interrupts, it will also disable the task switcher, stopping other tasks from being run as well. A general rule is to use the disabling of interrupts only for major system functions that are small in size and to use semaphores for the serialization for all other, more localized code.

To use semaphores correctly, code must do the following:

- Wait for the semaphore flag to become free (zero).
- Start running.
- Assert the flag again (to 1) to indicate to any other processes that they can't enter the critical region.
- Enter the critical region and execute the critical code.
- Reset the flag (to zero) again to indicate to pending processes that they can now enter the critical region.

A snippet of C/C++ code that would use this simple semaphore system is shown below. This is a *binary semaphore* since the value of the semaphore can only be either 1 or 0.

```
While (flag); /* wait for flag to become 1 */
flag = 0; /*set the flag to prevent another task entering */
/*code to access critical region goes here*/
flag =1; /* reset the flag to allow another pending process to enter */
```

Dijkstra (Di65) developed the semaphore system in 1965. His, the first semaphore system, was a *counting semaphore*. Counting semaphores are essentially integers. Taking a semaphore decrements the count, while releasing it increments the count. If a task tries to take a semaphore when the integer is zero, then that task is blocked. Semaphores are protected variables, and the counting semaphore can be accessed and altered only by the operations called P and V and an initialization operation that sets up the semaphore in the first place. The P operation essentially takes a semaphore, while the V operation puts it back.

> The P operation on semaphore S operates as follows:
> If S >0
> Then S:= S-1
> else (wait on S)
>
> The V operation on semaphore S operates as follows:
> If (one or more processes are waiting on S)
> Then (let one of the processes proceed)
> else S := S+1 (Deitel 1984 p89)

The following snippet of sample-C code indicates how these counting semaphores are used to provide mutual exclusion in accessing a shared data area by two tasks. Mutual exclusion means that, when one task has access, the other is excluded, and vice versa.

```
semaphoreinitialisation(active, 1) /* set the counting semaphore to 1 initially */
/* Task 1*/
While (True) do
    {
    P(active); /* decrement S to 0 and enter critical section other wise loop until task 2 exits the critical code and sets S back to 1 */
            /* do critical code section here */
    V(active); /*increment S back to 1 to allow task 2 to go ahead */
    }
/*Task 2 */
While (True) do
    {
    P(active); /* decrement S to 0 and enter critical section other wise loop until task 1 exits the critical code and sets S back to 1 */
            /* do critical code section here */
    V(active); /*increment S back to 1 to allow task 2 to go ahead */
    }
```

Note that, in the above example, both tasks are running at the same time. If task 2 was to get the CPU first, then it would exclude task 1, (S goes from 1 to 0) perform the critical code, and then signal that task 1 can now go ahead (S goes from 0 to 1). Hence there is no priority system here, and task 1 does not necessarily have the first opportunity to take a semaphore.

Counting semaphores are particularly useful when a resource is to be allocated from a pool of identical resources. The semaphore is first initialised to the number of resources in the pool. As each resource is used the P operation decrements the semaphore (S=S-1), and as each resource is returned the V operation increments the semaphore (S= S +1), indicating that it could be reallocated. If a P operation is attempted when there are no more resources (S=0), then that task must wait until a resource is returned to the pool.

The test-and-set instruction

In any language that is used to program semaphores, a test-and-set instruction is required. The language must have the ability to test for an available semaphore and take a semaphore within the same statement. If an interrupt occurred between the test for the available semaphore and the taking of the semaphore, then another task could already have entered the critical code in the meantime. Finally, the original task will erroneously take a semaphore and also

enter the critical code. For this reason the test-and-set instruction is required within one instruction. Here, assembler instructions have an advantage since the processor cannot be interrupted during the execution of a single instruction. Numerous instructions in assembly language can be used to perform a test-and-set instruction. The 80x86 TEST instruction, for example, compares data in one register (AX or AL) with that of a memory location and sets the overflow, sign, carry and zero flags all within the one instruction and without the possibility of being interrupted. The 'C' language does not have a test-and-set instruction that cannot be interrupted since each 'C' statement will be compiled into a number of assembly language instructions, allowing an interrupt to occur at any place within the compiled code. Fortunately, the disable() and enable() commands can be used to protect a test-and-set function in C from being interrupted.

```
int Set-semaphore(name)
{
 disable();
 int name ;
 (name>0) ? name = 0 : name = 1;       /* if semaphore called name is not already */
 enable() ;                /*already taken, so return 1 indicating that*/
 return (name);  /* taken (>0) take it and set it to 0. Else */
}                /* the attempt to take the semaphore failed*/
```

It is the programmer's responsibility to store the additional registers and restore them as required. When an interrupt occurs:

- The flags are pushed onto the stack (saved).
- The TF and IF flags are cleared to prevent other maskable interrupts.
- The current CS is pushed on to the stack (saved).
- The CS for the Interrupt Service Routine (ISR) is loaded from the interrupt vector table.
- The current IP value is then pushed onto the stack (saved).
- The new IP address is loaded from the interrupt vector table.
- Execution of the ISR then continues at the newly loaded CS:IP location obtained from the vector table until an interrupt return instruction IRET is encountered.
- The IRET instruction then pops the values of IP, CS and the Flags registers back off the stack and reloads the processor with these register values.
- The interrupted task then continues from the point it was at before it was interrupted.

Note that the registers are pulled off the stack in reverse order by the IRET instruction in the opposite order in which they were pushed on to the stack — Last In First Out (LIFO).

You are able to declare a function to be of type **interrupt**. You can then use this function as an interrupt service routine since it automatically preserves all the registers on the stack except for the Stack Pointer (SP) and Stack Segment (SS) and terminates the routine with the required IRET instruction. To use the **interrupt** modifier, the exact order in which the registers are stored must be known. Remember that the Flags CS and IP registers have already been stored on the stack before the interrupt modifier function is called. The C/C++ modifier function then pushes the rest of the registers on the stack in the following order:

1. push ax
2. push bx
3. push cx

4. push dx
5. push es
6. push ds
7. push si
8. push di
9. push bp

The stack after entering the ISR will be

BP ← Stack pointer
DI
DS
ES
DX
CX
BS
AX
IP
CS
FLAGS

We can use the C/C++ language's ability to directly manipulate the processor's registers and produce an image of the stack as it enters the interrupt service routine. The first time through, our multitasking operating system will have to allocate space for sets of registers on the stack, called *stack frames*, one for each of the tasks. After a task has run once, having been pre-empted, it will have its register values recorded in its own stack-frame space on the stack. To provide the space required for each stack frame on the stack, the following structure can be used as a direct image of the stack space needed for each task:

```
typedef struct int_regs
{
 unsigned bp;
 unsigned di;
 unsigned si;
 unsigned ds;
 unsigned es;
 unsigned dx;
 unsigned cx;
 unsigned bx;
 unsigned ax;
 unsigned ip;
 unsigned cs;
 unsigned flags;
 };
```

The following description is based on Schildt's code for a dual tasking RTOS kernel. (Schildt, 1989, p216). In order to locate each task's stack frame on the stack, we need to keep a copy of the stack segment and stack pointer for each stack frame. This will allow the OS to find the last state of a pre-empted task and reload it from the stack so that the task can continue running from the point where it was interrupted. The following array of structures, one for each task, can be used to keep the location of each task's stack frame on the stack. This is known in real time operating system jargon as the *Task Control Block* (TCB) because it stores the information required to run each task. There is also a pointer to each structure.

```
struct task_struct
{
    unsigned sp;
unsigned ss;
unsigned char *stck;
} tasks[2];
```

Now with these structures in place we can allocate the stack frame space for each task on the stack. The C/C++ function malloc() allocates memory space and returns a pointer that points to each block of memory allocated. This pointer will point to the stack space required for operating each task as well as sufficient space to store a copy of all the registers. This pointer to the stack frame is stored in the TCB.

```
tasks[id].stck = malloc(stck_size + sizeof(struct int_regs));
```

During the initialisation process using the code above, there will be memory allocated for that task's registers as well as general useable stack space for the task and a pointer to this memory stored in the TCB (called *task-struct*). See Fig 9. Note that the extra space above stck_size allows the processor to store its registers on the stack when entering the ISR as previously discussed. Also during this process, the initial values of CS, IP, DS and ES must be set in the register image area for each task . Each task will need to use some general stack area for its own operations, so it is important to have the register image stored at the top of the stack frame so that it is not overwritten by the task's stack. The stack grows down in memory and the bottom address of the stack frame is pointed to by the stack pointer in the TCB. First, a calculation is done to find the start of the space to be occupied by the register image and a pointer r is set to the base of the image area. Refer to Fig. 9.

```
struct int_regs *r;
r = (struct int_regs *) tasks[id].stck +
      stck_size - sizeof(struct int_regs);
```

Below r is the stack area that can be used by the task, while r also denotes the base of the register image structure above.

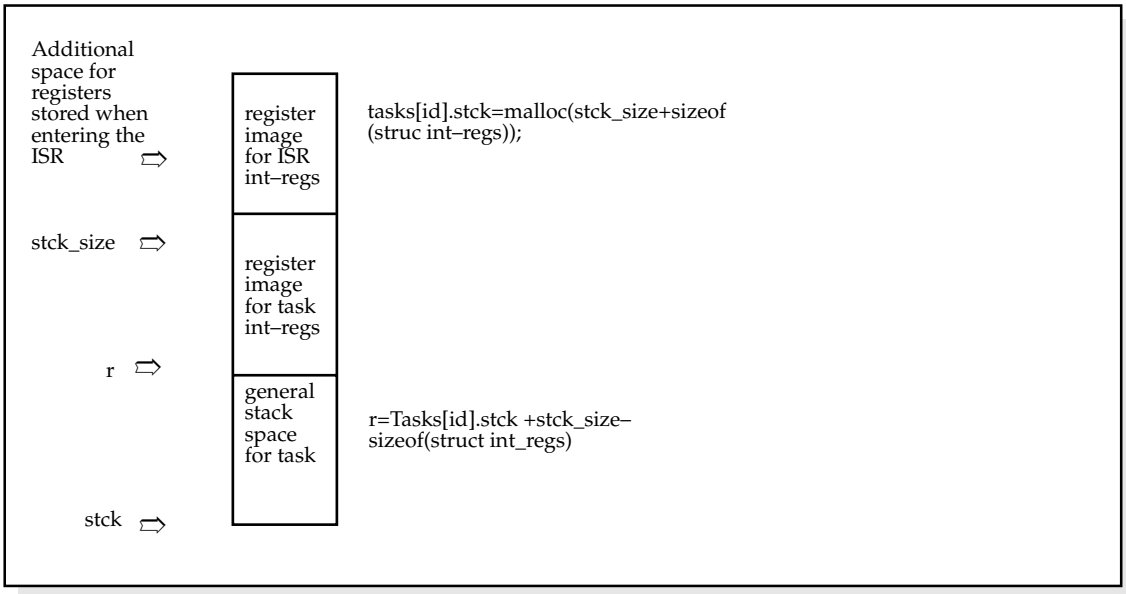

Fig 9 *Setting up the stack frame for a task.*

The segment and the offset of the first instruction of a particular task are stored in the CS and IP register respectively. The C/C++ functions FP_SEG() and FP_OFF() obtain the values of the CS and IP of each task that is used. During the first time through these tasks will begin their execution from the values already set up in the register image. After being pre-empted, the scheduler interrupt function, (which contains the IRET instruction) will overwrite this same register image with the current processor status of these registers.

```
/* set up task code segment and IP */
  r->cs = FP_SEG(task);
  r->ip = FP_OFF(task);
  /* set up DS and ES segments */
  r->ds = _DS;
  r->es = _ES;
  /* enable interrupts */
  r->flags = r->flags | 0x200;
```

The current data segment and extra segment which are used for pointing to data and string data are obtained by the C/C++ functions _DS and _ES and are also stored in the register image. The interrupt flag must be set to enable interrupts because that is how the task switcher is going to switch between tasks. The interrupt flag corresponds to 0x200 in the flag's register, and this line reads whatever the flag's register is, OR's that with 0x200 to set up the interrupt flag, and therefore enables the interrupts. The whole of the code required to set up the stack frames for multitasking is given below. Note that this function cannot be interrupted when running because the C/C++ functions disable() and enable() are used to disable interrupts during the time this code is running, thus making this critical portion of code atomic. The entire function required to set up the stack frame ready for multitasking is given below.

```
/* Define a new task */
 void make_task(task_type task, unsigned stck_size, unsigned id)
 {
      struct int_regs *r;
 disable();
tasks[id].stck = malloc(stck + sizeof(struct int_regs));
      r = (struct int_regs *) tasks[id].stck +
      stck_size - sizeof(struct int_regs);
  /*initialise task stack */
  tasks[id].sp = FP_OFF((struct int_regs far *) r);
  tasks[id].ss = FP_SEG((struct int_regs far *) r);
  /* set up task code segment and IP */
  r->cs = FP_SEG(task);
  r->ip = FP_OFF(task);
  /* set up DS and ES segments */
  r->ds = _DS;
  r->es = _ES;
  /* enable interrupts */
  r->flags = r->flags | 0x200;
 enable();
}
```

The above function is called for each task to be used by code similar to the following:

```
/* Initialise the tasks */
 make_task((task_type) f1, 1024, 0);
 make_task((task_type) f2, 1024, 1);
```

Setting up the task control blocks and the stack frames for each task is the hardest part of the RTOS design. The next part of the RTOS to be designed is the task switcher. Time slices must be created in order to run each task in its

own time slot. A regular source of interrupts can be used to achieve this, such as the system clock interrupt that causes interrupt 8 to be executed every 18.2 ms on PCs. This interrupt can be used to switch between tasks. The task switcher must intercept the timer 8 interrupt and run any existing ISRs that use interrupt 8 before performing the task switch. During the task switch, the stack pointer SS:SP of the old task must be saved. Remember that the interrupt has already saved the status of the old task in the register image area of its own stack frame when that task was interrupted. To complete the task switch, all that remains is to load the stack segment and stack pointer with the values for the new task. When the IRET instruction is encountered, the processor will be loaded with the register image pointed to by the SS:SP of the new task and will run the new task from where it left off last time, from the instruction pointed to by CS:IP. At this point the task switch is completed. The stack address for the new task's stack frame has been cleverly swapped for that of the old task. Remember, this interrupt routine was supposed to have been serviced before going back to process the old task, which will now never happen because SS:SP has been loaded with the new task's values and now points to a different part of the stack where the new task's register image resides. The code below shows the simple task switcher.

```
/* This intercepts timer int8 and performs a task switch */
 void interrupt int8_task_switch(void)
 {
 (*old_int8)(); /* call old int8 function */
 tasks[tswitch].ss = _SS; /* save old task stack */
 tasks[tswitch].sp = _SP;
 tswitch = !tswitch; /* switch tasks */
/* Note that tswitch may need to be initialised as a Boolean value to work with some compilers*/
 _SS = tasks[tswitch].ss;
 _SP = tasks[tswitch].sp;
 }
```

Some housekeeping is required before multitasking can take place. The old interrupt vector for interrupt 8 must be saved as we still need to call it (it may be doing something important) and return it when the multitasking is finished. The C/C++ function getvect() gets the address of the old interrupt handler, and the C/C++ function setvect() allows us to place the address of our task switcher ISR at this interrupt vector location. This will mean our task switcher ISR will be called first. Finally we need to direct the processor to start executing our first task by loading SS:SP with the location of the first instruction in our first task.

```
/* Start up the multitasking kernel. */
 void interrupt multitask(void)
 {
      disable();
      /* reset interrupt 8 */
  old_int8 = getvect(8);
  setvect(8, int8_task_switch);
  /* save original stack segment and pointer */
  oldss = _SS;
  oldsp = _SP;
  /* reroute stack to first task */
  _SS = tasks[tswitch].ss;
  _SP = tasks[tswitch].sp;
  enable();
  /* task switch begin running upon return */
 }
```

A minimal multitasking kernel follows (Schildt,1989,p 216):

```
/* A minimal two tasking kernel */
#include 'stdio.h'
#include 'dos.h'
#include 'alloc.h'
typedef struct int_regs
{
      unsigned bp;
 unsigned di;
 unsigned si;
 unsigned ds;
 unsigned es;
 unsigned dx;
 unsigned cx;
 unsigned bx;
 unsigned ax;
 unsigned ip;
 unsigned cs;
 unsigned flags;
 };
 struct task_struct
 {
```

```
    unsigned sp;
unsigned ss;
unsigned char *stck;
} tasks[2];
typedef int (far *task_type) (void);
void interrupt multitask(void);
void interrupt int8_task_switch(void);
void interrupt (*old_int8) (void);
void make_task(task_type task, unsigned stck, unsigned id);
void f2(void), f1(void);
unsigned oldss, oldsp;
int tswitch = 0;
char tasking = 1;
int i= 0, j=0;
main()
{
/* Initialise the tasks */
make_task((task_type) f1, 1024, 0);
make_task((task_type) f2, 1024, 1);
/* start multitasking */
multitask();
printf('\ndone\n');
printf('%d %d',i,j);
}
/* This intercepts timer int8 and performs a task switch */
void interrupt int8_task_switch(void)
{
(*old_int8)(); /* call old int8 function */
tasks[tswitch].ss = _SS; /* save old task stack */
tasks[tswitch].sp = _SP;
tswitch = !tswitch; /* switch tasks */
_SS = tasks[tswitch].ss;
_SP = tasks[tswitch].sp;
if(!tasking) /* stop tasking */
{
    disable();
 _SS = oldss;
 _SP = oldsp;
 setvect(8, old_int8);
 free(tasks[0].stck);
 free(tasks[1].stck);
enable();
}
}
/* Define a new task */
void make_task(task_type task, unsigned stck_size, unsigned id)
{
```

```
    struct int_regs *r;
disable();
                tasks[id].stck = malloc(stck_size + sizeof(struct int_regs));
    r = (struct int_regs *) tasks[id].stck +
    stck_size - sizeof(struct int_regs);
 /*initialise task stack */
 tasks[id].sp = FP_OFF((struct int_regs far *) r);
 tasks[id].ss = FP_SEG((struct int_regs far *) r);
 /* set up task code segment and IP */
 r->cs = FP_SEG(task);
 r->ip = FP_OFF(task);
 /* set up DS and ES segments */
 r->ds = _DS;
 r->es = _ES;
 /* enable interrupts */
 r->flags = r->flags | 0x200;
enable();
}
/* Start up the multitasking kernel. */
void interrupt multitask(void)
{
    disable();
    /* reset interrupt 8 */
 old_int8 = getvect(8);
 setvect(8, int8_task_switch);
 /* save original staack segment and pointer */
 oldss = _SS;
 oldsp = _SP;
 /* reroute stack to first task */
 _SS = tasks[tswitch].ss;
 _SP = tasks[tswitch].sp;
 enable();
 /* task switch begin running upon return */
}
/* Task 1 */
void f1(void)
{
    for(;;i++)
 {
    if(!(i%10)) printf('%d %d\n',i,j);
 if(i==1000) tasking = 0;
 }
}
/* Task 2 */
void f2(void)
{
    for (;;j++);
}
```

Protected mode considerations

The advantages of *protected mode* over *real mode* are that the memory above 1 Mbyte (extended memory) can be accessed allowing the use of up to 4 gigabytes of memory by a 32 bit processor such as the i386. The other advantage is that this memory is protected memory, looked after by the processor's on-board memory manager, so that code cannot be erroneously written over by another program.

The advantage of using protected mode in producing a RTOS is that each task can run its own protected segment of memory protected by the CPU's onboard memory manager. Alternatively, the RTOS kernel can be run in protected mode while the tasks are run in real mode, allowing DOS and BIOS calls from the tasks.

In protected mode, just about everything is different from real mode. In contrast to the simple base and segment register addressing scheme used in real mode, the protected mode addressing scheme uses a system of selectors and descriptors which contain privilege levels. This means that code with lower privilege levels must use a special gateway to access code of higher privilege such as the operating system. The descriptors are used to define the privilege levels and segments of memory where groups of code are placed. The selector is an address that points to the descriptor. The descriptors must be set up before processing takes place, since the x86 processor will load the descriptors and make addressing decisions based on them at run time. The processor's memory manager uses the contents of the descriptors in the descriptor tables to ensure there are no memory violations.

Programming the descriptors is difficult in assembly language. However, in the C language, this can be done using a linker that will automatically supply the code to set up the descriptor tables when the correct target is specified. The programmer simply specifies the environment, and the linker does the rest. DOS based compilers only produce code for real mode. Translating real mode addresses into protected mode addresses, and vice versa, is a difficult task. Fortunately, commercially available locaters can be used to do this address translation, allowing the real mode code to run in protected mode.

DOS or real mode programs can be run in virtual 8086 mode where they will run in protected memory and not be written over by other errant programs or code. They will also have access to extended memory. In addition, the memory management unit (MMU) on the CPU is invoked, providing protection from

errant code overwriting the kernel or other tasks. If another program or operating system has already set the i/o privilege level (IOPL) lower than 3, which is very likely, the simple multitasking kernel previously described will not run in virtual 8086 mode. For this reason, the simple multitasking kernel previously described should be run in the real mode (DOS environment). Since all virtual 8086 mode programs are run at a privilege level of 3 (PL3), they may not be able to use a number of PL0 assembler instructions, including the interrupt instructions CLI, STI and IRET needed for the simple multitasking kernel. However, all interrupts are emulated in this mode to prevent code in one segment of protected memory from crashing the system that may be being used by other programs also resident in protected memory. The rationale is that it is much better for the execution of one portion of code to be stopped on an interrupt violation rather than bringing down the entire system. An interrupt monitor looks after this process using a system of virtual interrupts. These virtual mode interrupts emulate the real mode interrupt system previously explained.

In protected mode, an interrupt descriptor table (IDT), that can exist anywhere in memory, is used to locate the address of the interrupt handler. The IDT takes the place of the Interrupt Vector Table (IVT) in real mode. Once again this descriptor table must be set up in memory before a switch to protected mode takes place and once again a linker, or a locator in the case of a DOS-based C compiler, can be used to resolve this.

Virtual interrupts took too long to respond to as instigated on the i386 and some i486 CPUs. Task switching time was required to switch between the virtual 8086 program and the virtual 8086 monitor, and the emulation routines also took too long to run. Intel got around this by bringing out virtual 8086 mode extensions on i486DX 4 and Pentium CPUs. These extensions are available on chips with CPU identification (CPUID), and that is how systems with this feature are identified. The VME bit in CR4 is used to enable the interrupt extensions. When this bit is asserted the virtual mode interrupt extensions are enabled, giving rapid interrupt response. When this bit is set to 0, the Pentium chips and SL-enabled 486 CPUs exactly emulate i486 and i386 virtual 8086 mode. Because there are a number of assembler instructions, namely, PUSHF, POPF,LOCK, INTn, IRET, CLI and STI that will cause general-protection faults if the current privilege level (CPL) is not correct, the VME bit in CR4 may need to be set to 0 on some legacy code written for real mode. Alternatively in C, the source code can be recompiled to take advantage of the faster interrupt extensions. In this case, the VME bit is set to 1 and the Pentium provides a hardware virtual interrupt flag (VIF) and a virtual interrupt pending flag (VIP) for the virtual 8086 mode. Instead of changing the interrupt flag in the EFlag register as occurs

in real mode, the VIF flag is set in CR4, establishing a virtual interrupt . (Messmer 2002, p 328). Once again, the CPU target and mode can be set through a linker, locator, or linker /locator and through recompiling the C code for a new target CPU.

A particularly beneficial mode for the embedded engineer is an undocumented mode that has been termed *flat mode* or *unreal mode*. This is the mode you get when you return to real mode from protected mode. Returning from protected mode leaves a 4Gbyte limit for linear addressing much more than the 1 Mbyte that is available under real mode. In addition, it is compatible with BIOS and DOS. Unfortunately, those BIOS interrupts and other programs that switch in and out of protected mode will corrupt the extended 4 Gbyte limit (Fine, 2002). This mode can be entered through an assembly language program (Fine, 2002, p6).

In summary the multitasking C code is easily shifted to work with various target processors in a variety of modes. This would not be as easy when working in assembler due to the complexity of switching modes, handling address translations into protected mode, setting up descriptor tables, and handling interrupt vectors for each new target CPU. The features available on 3rd party linkers and locators should be checked for more options.

Conclusion

Producing a small multitasking kernel to run on any processor requires the following steps:

1. Study the interrupt system of the processor and set up a source of regular interrupts, often called *real time interrupts*.

2. Study the stacking of the processor's registers during interrupt servicing, and set up stack frames containing space for these registers, which will eventually provide the context for each task.

3. Work out where to set up the pointer to each task's stack frame. This will be used to reload the registers back off the stack frame and run the next task.

4. Look at how the 'return from interrupt' instruction works in pulling registers back off the stack.

5. Using your knowledge of how the return from interrupt instruction works, swap the return address while inside the ISR so that it points to the next task to be run when the 'return from interrupt' instruction is encountered.

The C/C++ language offers sufficient inbuilt functions to build an RTOS. Using an optimizing C/C++ compiler, small, tight code can be produced that is as small and as fast as hand-coded assembly language. The portability benefits of the C/C++ language allow the same RTOS to be used on different target processors with little or no programming effort. Cross compilers are available that allow the RTOS to be ported to a range of embedded processors. Care must be taken when compiling for a different target processor that the correct interrupt, stack and register-specific library functions are used. Using the special C functions for handling interrupts simplifies the programmer's task. The following table includes some but not all of Borland C/C++ functions that are of assistance in creating the kernel for an RTOS:

disable() disable interrupts. Only the NMI (non-maskable interrupt) is allowed from any external device.
enable()-enable interrupts, allowing any device interrupts to occur.
FP_OFF() gets or sets the offset of the far pointer
FP_SEG() gets or sets the segment value of the far pointer Return Value
geninterrupt()can be used for testing interrupt routines during development.
getvect() reads the value of the given interrupt vector and returns that value as a (far) pointer to an interrupt function.
setvect() sets the value of the given interrupt vector to the address of the new interrupt service routine(ISR). The ISR must have first been declared as interrupt function.
interrupt() this modifier is specific to Borland C++. interrupt functions are designed to be used with interrupt vectors.
Interrupt functions compile with extra function entry and exit code so that all CPU registers are saved. The BP, SP, SS, CS, and IP registers are preserved as part of the C-calling sequence or as part of the interrupt handling itself. The function uses an IRET instruction to return, so that the function can be used as harware and software interrupts.
_cs, __cs, _ds, __ds, _ss, __ss, _es, and __es are near pointers to the specified segment registers.

In particular the interrupt modifier is of great benefit, because the 80x86 assembly instruction IRET does not store all of the processor's registers on the stack. To save a task's context, the assembler programmer must make arrangements to do this. C/C++ programmers do not have to do this. They simply declare an ISR function to be of type interrupt and the interrupt modifier does this for them.

No test-and-set instruction is available in C/C++ that would allow, for example, a semaphore to be tested and set in one operation. However, through the use of atomic code or semaphores, this can be achieved. The temptation to use inline assembly within the C/C++ code should be avoided since that will make the code processor-dependent and prevent the RTOS being portable.

When the tasks are written in a higher-level language, the functions of the higher-level language must be checked for re-entrancy. If they are not already re-entrant they can be encapsulated so that they become re-entrant. A check of the data types should also be undertaken to determine which ones are stored on the stack and can therefore be used without problem and which are not. Global and static variables, if used, must be encapsulated to allow the serial use of them by other tasks.

The corollary of using semaphores and atomic code to provide reentrancy and protect data is that there will be fewer places in the code where the RTOS kernel can perform a task switch. Shouldn't this affect the performance? Generally, it will not if that encapsulated code is cohesive and performs only the one task and is arranged so that, if it waits for an external event, it is segregated. Predetermining just where code may be interrupted may even be beneficial as it makes faults on the RTOS easier to isolate.

Bibliography

Bentham, J. (2000) . *TCP/IP LEAN Web servers for embedded systems*. Lawrence, KS: CMP Books.

Deitel, H. (1984). *An introduction to operating systems.* Reading, MA: Addison-Wesley.

Fine, J. (2002). Homepage, Retrieved August 2002 from http://www.execpc.com/~geexer/johnfine/segments.htm

Labrosse, J. (2000). *Embedded systems building blocks* (2nd Ed). Lawrence, KS: CMP Books.

Laplante, P. A. (1997). *Real time systems design and analysis: An engineers handbook.* Piscataway, NJ: Press Marketing.

Mesmer, H. (1997). *The indispensible PC hardware book.* (3rd Ed). Essex, England: Addison Wesley Longman.

Mesmer, H. (2002). *The indispensible PC hardware book.* (4th Ed). Essex, England: Addison Wesley Longman.

Murray, W., & Pappas, C. (1986). *80386/80286 assembly language programming*. Berkeley, CA: Osborne McGraw-Hill.

Simon, D. (1999). *Embedded software primer*. Boston, MA: Addison Wesley Longman.

Schildt, H. (1989). *Born to code in C*. Berkley, CA: Osborne McGraw Hill.

Van Gilluwae, F. (1994). *The undocumented PC*. Reading, MA: Addison-Wesley.

Wyatt, A. (1987). *Using assembly language.* Carmel, IN: Que Corporation.