7 Real-Time Operating System

We now come to the Real-Time Operating System (RTOS) and we start by giving the general principles behind the RTOS concept. An explanation of the three main variations of RTOSs which we will deal with is given, namely the round-robin, co-operative and pre-emptive versions of the RTOS. These categories are explained in section 7.2 I have developed operating systems of these three main versions of RTOSs and will be explained in detail in the following chapters. There are the ParrOS (assembly language version, Appendix A) and SanctOS (C language version, see Chapter 8 and Appendix C) which are both of the round-robin type. Then there is the PaulOS RTOS (see Chapter 9 and Appendix B and Appendix D) which is of the co-operative type and finally I have the MagnOS (see Chapter 10 and Appendix E) which falls in the pre-emptive category.

7.1 What is a Real-Time Operating System

This chapter introduces the concept of the RTOS. Such a system is not something out of this world, and once the concept is understood, one would be able to modify and expand the programs listed in this book, to suit his/her own requirements. It should be pointed out at the outset, that all the programs found in this book, are experimental. They all work, but I do not accept any responsibility if they are used in a system.

The RTOS is an operating system that guarantees a certain capability within a specified time constraint. For example, an operating system might be designed to ensure that a certain part or item is available at a certain point on an assembly line. In what is usually called a "hard" real-time operating system, if the program code for such an operation cannot be performed in time for making the part available at the designated time and place, the operating system would terminate with a failure. In a "soft" real-time operating system, the assembly line would continue to function but the production output might be lower as objects fail to appear at their designated time, causing the operator (or robot) to be temporarily unproductive. Some real-time operating systems are created for a special specific application and others are more of a general purpose type which can be adapted to various situations. It immediately becomes clear that real-time does not have an absolute value of time but the reaction time can vary depending on the application.

One might already be familiar with other operating systems used on personal computers, such as DOS, Windows, Unix or Linux. The RTOS to be discussed in this chapter is a much smaller version, written specifically for small micro-controllers.

The general idea is to write an operating system which would take overall control of the whole situation, particularly scheduling tasks (routines) at appropriate times according to the program logic or algorithm.

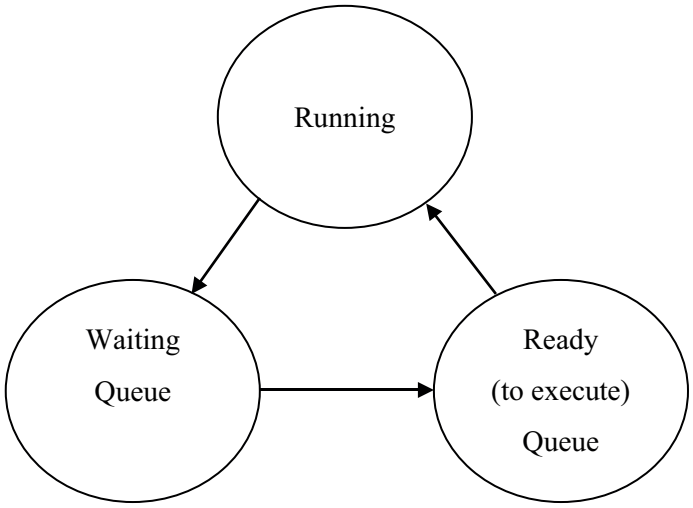

Figure 7-1 RTOS Task states diagram

The application program would be split up into "short" ENDLESS programs (routines, functions or procedures) known in the RTOS environment as TASKs. The RTOS can then be thought of as an organiser or scheduler of these individual tasks, controlling which task should be running and which task should run next. There would of course be only one task running at a particular time, but tasks would be switching in and out so fast that they would give the impression of running simultaneously or multi-tasking. The RTOS is simply time-slotting each task, in a time-multiplexing technique.

As show in Figure 7-1, a task can be in one of the following three states:

- RUNNING: Only one task would be in this state, since the micro-controller can only execute one program at any one time. As the name implies, this would be the task currently executing.
- WAITING: Tasks end up here after running for a preset time or because the task itself requested to wait. Tasks can be made to wait for:
  - Timeout: wait for a specified time
  - Signal: wait for a signal which would come from another task
  - Interrupt: wait for some external interrupt
  - Some other event: usually, tasks which have finished waiting are placed in the Ready queue.
- READY: This is a queue where all the tasks which are ready to execute are held whilst waiting their turn. Once the task currently running either has its time-slot expired or itself opts to wait, then the task next in line in this ready queue will take over and become the running task.

It should be emphasised here that each task should be written as an endless loop or sub-program. Since tasks would be switching on and off, and each particular task can make use of a certain number of registers and it might also push on the stack some registers (or addresses in case of some call instructions), the biggest problem of the RTOS is how to handle these situations so that the tasks do not overwrite the registers used by other tasks, and they do not disturb the stack area.

Since the 8051 family of microprocessors have 4 register banks, allocating a different register bank to each task would solve the first problem. This would restrict the number of tasks to 4 and therefore is not that good unless you have only a very limited number of tasks.

The second problem could be solved by having a different stack area for each task, and loading the stack pointer (SP) accordingly before switching tasks. Now the following question arises: How can we switch tasks? If we remember what happens during an interrupt (or call instruction) we would be on the right track to answer the question. Whenever a CALL is executed, the address of the next instruction in the program is pushed on the stack. This address is retrieved by the RET (or RETI) instruction, using the locations pointed to by SP, so that the program continues where it left from. The RTOS operates on an interrupt basis, usually using a Timer interrupt at regular (for example 1ms) intervals. All that is required to be done in order to change tasks is therefore to have on the stack the address of the next task instead of the present one. This can be done by changing the actual address on the stack or else (and this is the method used in our RTOSs), point the SP to a different stack area, where the address of the new task is stored. With this method, the task swapping would be done and moreover, each task would have its own stack area for pushing registers, call routines etc.

7.2 Types of RTOSs

We could split the RTOSs into 3-types. There is the round-robin , co-operative and pre-emptive RTOS. These are described in more detail in later chapters but as a brief explanation the main differences and properties can be mentioned here.

7.2.1 Round-robin RTOS

The round-robin RTOS is a very simple operating system which allocates each task a specific time to operate. After this time elapses, the RTOS would stop this task, save its environment, replace the environment with that of the next task and then start the next task again for a specific time. In general, the time for each task would be the same, but not necessarily so.

It should be made clear at this point that once this allocated time slot is over, the processor shelves this task at whatever instruction it happens to be executing and starts (or continues) the next task in the queue. Each task could be a very small routine, written as an endless loop, which would be repeating on and on, and executing in bursts, a few instructions at a time (depending on the slot time) when its allocated time is used by the micro-controller.

If a task needs to be stopped permanently at some time, there could be a

```
Here:   SJMP Here       (or while(1); if written in C)
```

so that the task would actually be running when its slot time comes around, but not doing anything useful. This is not so efficient since it would still be wasting valuable processor time. A more efficient way would be to kill the task completely by removing it from the queue.

More on this type of RTOS can be found in the assembly language version program ParrOS (see Appendix A) and in the SanctOS, written in C which is fully described in Chapter 8.

7.2.2 Co-operative RTOS

The co-operative RTOS (such as PaulOS in Appendix B and D and Chapter 9), is a further improvement on the round-robin RTOS. In this case, a task, which is again written as an endless loop would run until the task itself would issue an RTOS command which would cause a change of task. These commands would depend on the operating system itself and for the case of PaulOS, there are commands which would cause the task to pause and go into a waiting queue, thus giving up its processor time to another task. The task may for example wait for a specified time delay, or it may wait for an interrupt or for some signal from another task.

Once this command is executed, the RTOS would initiate a change of task process similar to the round-robin case, and the next task in the queue which is ready to execute would take over. There could be instances where all the tasks would be waiting for a time delay or for an event to occur, in which case there would not be any tasks ready to execute. In this case, the main() code would run, which would normally be executing a useless *while(1);* loop (also setting the controller in an idle mode to save energy).

If all the tasks are independent and all have the RTOS instruction to wait for say 5ms, then the co-operative RTOS would be working exactly the same as the round-robin type, with each task coming into action in sequence.

More on this type of RTOS can be found when describing the PaulOS RTOS in Chapter 9.

7.2.3 Pre-Emptive RTOS

The pre-emptive RTOS such as MagnOS is still a further improvement on the previously mentioned RTOSs. Here each task is given a priority number and the task with the highest priority is given the go-ahead to execute by the RTOS. Unless the task itself executes a 'wait' instruction to give up its time, then it will continue to run since a task of a lower priority would not be permitted to run and thus interrupt the higher priority one. On the other hand, if a low-priority task is currently running and a higher priority task moves to the ready (to execute) queue (for example because it was waiting for some time delay which has now passed), then the RTOS would stop the lower priority task and place it in the ready queue and the higher priority task would then take over and start/continue to execute. More information on the MagnOS RTOS is given in Chapter 10.

In this environment it is very important to allocate the right priorities to the tasks so as to be sure that all tasks are given a chance to run. Various theories or ideas exist about priority allocation techniques, the Rate Monotonic Scheduling or Algorithm ([19] C.L. Liu and J.W. Layland) being one of the most popular.