

The engineering of real-time embedded systems

Real-time Operating Systems

Book 1 – The Theory

Jim Cooling

The engineering of real-time embedded systems



Real-time Operating Systems

Book 1 - The theory

Jim Cooling

For Nieve and Gavin

Who give us the best possible reason to visit Australia
Tá tú go deo inár gcroí ar

Published by Lindentree Associates © 2017

The rights of James E Cooling to be identified as author of this work has been asserted by him in accordance with the Copyright, Design and Patents Act 1988.

Real-Time Operating Systems - Table of Contents

Opening pages

Preface

[What is this book series about?](#)

[Who should read this book series?](#)

[What is this book about?](#)

[How should the book be read?](#)

[Acknowledgements](#)

Chapter 1 Real-time operating systems - things you really ought to know.

[1.1 Setting the scene](#)

[1.2 Producing quality software](#)

[1.3 Modelling the software](#)

[1.4 The importance of time and timing](#)

[1.5 Handling multiple jobs](#)

[1.6 Handling complex multiple jobs](#)

[1.7 Using interrupts as the execution engine - simple quasi-concurrency](#)

[1.8 Basic features of real-time operating systems](#)

[1.9 Executives, kernels and operating systems](#)

[1.10 Task-based software design - a recap](#)

[Review](#)

Chapter 2 Scheduling - Concepts and implementation.

[2.1 Introduction](#)

[2.2 Simple cyclic, timed cyclic and cooperative scheduling](#)

[2.3 Round-Robin \(time slicing\) scheduling](#)

[2.4 Task priorities](#)

[2.5 Using queues](#)

[2.6 Priority pre-emptive scheduling](#)

[2.7 Implementing queues - the task control block](#)

[2.8 The process descriptor](#)

[2.9 The tick](#)

[2.10 Priorities and system responsiveness](#)

[2.11 By-passing the scheduler](#)

[2.12 Code sharing and re-entrancy](#)

[2.13 The unpredictability of run-time behaviour.](#)

[Review](#)

[Chapter 3 Control of shared resources - mutual exclusion.](#)

[3.1 The problem of using shared resources](#)

[3.2 Mutual exclusion using a single flag](#)

[3.3 The semaphore](#)

[3.3.1 The binary semaphore](#)

[3.3.2 The general or counting semaphore](#)

[3.3.3 Semaphore limitations and problems](#)

[3.4 The mutex](#)

[3.5 The simple monitor](#)

[3.6 Mutual exclusion - a final comment](#)

[Review](#)

[Chapter 4 Shared resources and contention issues](#)

[4.1 Resource contention - the deadlock problem in detail](#)

[4.2 Producing deadlock-free systems](#)

[4.3 Preventing deadlocks](#)

[4.3.1 Simultaneously sharing resources](#)

[4.3.2 Allowing request pre-emption](#)

[4.3.3 Controlling resource allocation](#)

[4.4 Priority inversion and task blocking](#)

[4.4.1 The priority inversion problem](#)

[4.4.2 Basic priority inheritance protocol](#)

[4.4.3 Immediate priority ceiling protocol](#)

[4.5 Deadlock prevention and performance issues](#)

[Review](#)

[Chapter 5 Intertask communication](#)

[5.1 Introduction](#)

[5.1.1 Overall aspects of intertask communication](#)

[5.1.2 Coordination versus synchronization](#)

[5.2 Task interaction without data transfer](#)

[5.2.1 Task coordination mechanisms](#)

[5.2.2 Task synchronization using event flags - the unilateral rendezvous](#)

[5.2.3 Task synchronization using signals - the bilateral rendezvous](#)

5.3 Data transfer without task synchronization or coordination.

5.3.1 Overview

5.3.2 Pools

5.3.3 Queues

5.4 Task synchronization with data transfer

Review

Chapter 6 Memory usage and management

6.1 Storing digital information in embedded systems

6.1.1 Introduction

6.1.2 Non-volatile data stores

6.1.3 Volatile data stores

6.1.4 Memory devices - a brief Flash-RAM comparison

6.1.5 Memory devices - a brief SRAM-DRAM comparison

6.1.6 Embedded systems - memory device organization

6.2 Memory aspects - conceptual and physical.

6.3 Eliminating inter-task interference

6.3.1 Controlling memory accesses - a simple approach

6.3.2 Controlling memory accesses with a memory protection unit

6.3.3 Controlling memory accesses with a memory management unit

6.4 Dynamic memory allocation and its problems

6.4.1 Memory allocation and fragmentation

6.4.2 Memory allocation and leakage

6.4.3 Secure memory allocation

6.5 Memory management and solid-state drives.

Chapter 7 Multiprocessor systems

7.1 Embedded multiprocessors - why and what?

7.1.1 When one processor just isn't enough.

7.1.2 Processor structures - a general overview.

7.1.3 Multicore processors - symmetric and asymmetric types.

7.1.4 Multicomputer structures.

7.2 Software issues - job partitioning and allocation.

7.2.1 Introduction.

7.2.2 Structuring software as a set of functions.

7.2.3 Structuring software as a set of data processing operations.

7.3 Software control and execution issues.

7.3.1 Basic OS issues.

- [7.3.2 Scheduling and execution in AMP systems.](#)
- [7.3.3 Scheduling and execution in SMP systems.](#)
- [7.3.4 Scheduling and execution in BMP and mixed-mode systems.](#)
- [7.3.5 A last comment.](#)

[Review](#)

[Chapter 8 Distributed systems](#)

- [8.1 Software structuring in distributed systems.](#)
- [8.2 Communication and timing aspects of distributed systems.](#)
- [8.3 Mapping software onto hardware in distributed systems.](#)

[Review](#)

[Chapter 9 Analysis and review of scheduling policies](#)

- [9.1 Overview](#)
- [9.2 Priority-based non-pre-emptive scheduling policies.](#)
- [9.3 Priority-based pre-emptive static scheduling policies - general.](#)
- [9.4 Priority-based pre-emptive static scheduling - rate monotonic scheduling.](#)
- [9.5 Priority-based pre-emptive static scheduling - combining task priority and criticality: a heuristic approach.](#)
- [9.6 Priority-based pre-emptive dynamic scheduling policies - general.](#)
- [9.7 Priority-based pre-emptive dynamic scheduling - earliest deadline scheduling.](#)
- [9.8 Priority-based pre-emptive dynamic scheduling policies - computation time scheduling.](#)
- [9.9 Priority-based pre-emptive dynamic scheduling policies - spare time \(laxity\) scheduling.](#)
- [9.10 Improving processor utilization - forming rate groups](#)
- [9.11 Scheduling strategy - a final comment](#)
- [9.12 Scheduling timing diagrams - list of symbols](#)

[Review](#)

[Chapter 10 Operating systems - basic structures and features.](#)

- [10.1 Setting the scene.](#)
- [10.2 Simple multitasking via interrupts.](#)
- [10.3 The Nanokernel.](#)
- [10.4 The Microkernel.](#)
- [10.5 A general-purpose embedded RTOS.](#)

[Review](#)

[Chapter 11 Performance and benchmarking of RTOSs](#)

[11.1 Introduction.](#)

[11.2 Measuring computer performance - Benchmarking.](#)

[11.2.1 Introduction.](#)

[11.2.2 Computation performance benchmarks.](#)

[11.2.3 OS performance.](#)

[11.3 Time overheads in processor systems.](#)

[11.4 OS Performance and representative benchmarks.](#)

[11.5 OS Performance and synthetic benchmarks.](#)

[11.5.1 Overview.](#)

[11.5.2 Basic requirements.](#)

[11.5.3 Test categories.](#)

[11.5.4 Baseline \(reference\) test data.](#)

[11.5.5 Test stressing methods.](#)

[Review](#)

[Chapter 12 The testing and debugging of multitasking software](#)

[12.1 Setting the scene.](#)

[12.2 Testing and developing multitasking software - a professional approach.](#)

[12.3 In-target testing - practical tool features.](#)

[12.3.1 Overview.](#)

[12.3.2 RTOS testing using dedicated control and data collection units.](#)

[12.3.3 RTOS testing using on-chip data storage methods.](#)

[12.3.4 RTOS testing using host-system data storage facilities.](#)

[12.4 Target system testing - some practical points.](#)

[12.4.1 Introduction.](#)

[12.4.2 Testing the concurrency of individual tasks.](#)

[12.4.3 Implementing and testing concurrent operations.](#)

[Review](#)

[Chapter 13 Epilogue](#)

[13.1 Tasks, threads and processes](#)

[13.1.1 General aspects](#)

[13.1.2 Code execution in an embedded environment - a simple-man's guide](#)

- [13.1.3 Software activities, applications and tasks](#)
- [13.1.4 Concurrency within tasks in a single processor system](#)
- [13.1.5 Running multiple applications](#)
- [13.1.6 Summary](#)
- [13.2 Running multiple diverse applications - time partitioning](#)
- [13.3 RTOS's vs. general-purpose OS's.](#)
- [13.4 Bibliography and reference reading material.](#)

[Index](#)

Preface

What is this book series about?

These books set out to provide a firm foundation in the knowledge and skills needed to develop and produce real-time embedded systems. They fall into two categories:

- Those providing a firm foundation in the fundamentals of the subject.
- Those showing how to use and apply specific design and development skills.

Engineers from the well-established professions (electronic, mechanical, aeronautical, etc.) fully understand the distinction between these two aspects. Moreover, experienced engineers recognize that to effectively apply your skills you must truly have a good grasp of fundamentals. Regrettably, this view is sadly lacking in the area of Software Engineering.

Who should read this book series?

This is intended to meet the needs of those working - or intending to work - in the field of software development for real-time embedded systems. It has been written with four audiences in mind:

- Students.
- Engineers, scientists and mathematicians moving into software systems.
- Professional and experienced software engineers entering the embedded field.
- Programmers having little or no formal education in the underlying principles of software-based real-time systems.

What is this book about?

This book deals with the fundamentals of the subject. Simply put, it sets out to answer the following major questions:

1. Just what is a real-time operating system (RTOS)?
2. Why should you use an RTOS in your designs?
3. Are there downsides in using an RTOS?
4. What are the building blocks of embedded real-time operating systems?
5. Modern embedded systems use single processor, multiprocessor, multi and multiple computers. How do we employ RTOSs across such a range of platforms?
6. How can you evaluate the performance of your RTOS and, if necessary, improve it?
7. How do you go about debugging RTOS-based designs?

The table of contents shows what to expect in more detail; moreover, each chapter begins with a clear statement of its objectives. I recommend that you briefly review these to get a good idea of the scope and intent of the book as a whole.

The original book has been retitled as ‘Real-time Operating Systems, Book 1 - The Theory’. The reason for doing this is that a new book has been produced as a companion piece: ‘Real-time Operating Systems, Book 2 - The Practice’. This contains a set of exercises for you to work on (if you wish to, of course), to help your understanding of the topic. These exercises relate to the core aspects of the subject, that covered in chapters 1 to 5. I recommend that your practical work should run in parallel with the theoretical material. Doing this will put you in a very good position to take on real RTOS-based design problems.

And lest I be accused of sexism please note that I use 'he' as shorthand for 'he or she'.

How should the book be read?

Please, please, everybody, experienced or not, read chapter 1. And not only read it but absorb its message. Because if you don't truly understand the issues discussed here you're going to struggle to produce good designs.

Chapters 2 to 6 cover the fundamentals of the subject. They aim to show not only how multitasking designs can be implemented but also why things are done in certain ways. The target audience here are those new to tasking design and implementation in real-time embedded systems. But, important point, it places the work mainly in the context of single processor units. Chapters 7 and 8 broaden this to include multiprocessors and distributed systems (and it needs saying that the boundary between these two is somewhat fuzzy).

Chapter 9, essentially theoretical with a practical bias, takes a much broader view of task scheduling techniques. The reason for leaving it to this later stage is to make it easier for the reader to assimilate the material. Provided he is au fait with the basic concepts of the subject this should be a fairly straightforward chapter.

Chapters 10 to 12 are geared towards the practicalities of the subject. If you are new to the RTOS field, chapter 10 will give you a good grounding in the differences between OS structures. This may prove to be especially useful when you come to select your first RTOS. By contrast, the material detailed in chapters 11 and 12 will be particularly useful when you've built your system. They are, in essence, concerned with the behaviour, quality and reliability of the software at run time.

Acknowledgements

Figure 1.5 photo montage:

RAF SEPECAT Jaguar GR3

Avro Vulcan B2 XH558

Images courtesy of Tim Beach at FreeDigitalPhotos.net RAF Typhoon

Image courtesy of Bernie Condon at FreeDigitalPhotos.net Figure 3.4:

Railway Signal

Image courtesy of Simon Howden at FreeDigitalPhotos.net Figure 5.2:

Automated materials handling.

Image courtesy of Lambert Material Handling, Syracuse, New York 13206

Figures 12.6, 12.7, 12.8, 12.9 and 12.10.

Images courtesy of Percepio AB

And lastly, many words of thanks to my son Niall for his diligence in checking the manuscript. He (fortunately for me) highlighted a number of issues with my code fragments and so saved me from much public embarrassment.

Good reading, ladies and gentlemen. I wish you well.

Jim Cooling

Markfield, May 2017

(About the author: if you're *really* interested, see www.lindentreeuk.co.uk)



Chapter 1 Real-time operating systems - things you really ought to know.

The objectives of this chapter are to

- Show you why there are good reasons for using real-time operating systems (RTOSs).
- Show you that there are also drawbacks when RTOSs are used (simply put, 'win a bit, lose a bit').
- Describe how RTOS support simplifies the functional design of software, thus making it easier to produce high-quality systems.
- Highlight the special importance of time and timing when using an RTOS-based design.
- Describe the fundamental objectives, structure and operation of task-based design.

1.1 Setting the scene

In the world of large computers, operating systems (OSs) have been with us for quite some time. In fact the elementary ones go back to the 1950s. Major steps were made in the 1960s, and by the mid 1970s their concepts, structures, functions and interfaces were well established.

The micro arrived about 1970. It would seem logical that operating systems would find rapid application in microprocessor-based installations. Yet by the mid 1980s few such implementations used what could be described as formally-designed real-time operating systems. True, CP/M was released in 1975, and was later put into silicon by Intel. But it made little impact on the real-time field; its natural home was the desktop machine.

Two factors affected the take-up of RTOSs, one relating to machine limits, the other to the design culture surrounding the micro. The early micros were quite limited in their computing capabilities, speed of operation and memory capacity. Trying to impose an operating system structure on this base was quite difficult. Moreover, the majority of those programming embedded systems had little or no background in operating systems.

Nowadays things are quite different. The key workhorses of modern embedded designs are 16/32-bit complex microcontrollers. These are low-cost, high-performance devices, incorporating extensive on-chip memory and peripheral features. Moreover, many many commercial RTOSs are on the market. But, just because you can do something doesn't mean that you should do it. So, why exactly should you choose to use an RTOS in your next design? First, though, we need to consider a much more basic question; just *how* should we tackle the design of software in embedded systems. And *that* is the key point of this chapter. It lays the foundations for a pragmatic design technique and shows exactly how the RTOS fits in with this.

1.2 Producing quality software

At first sight it might seem strange to start this chapter by talking about the quality of software. Not a lot to do with operating systems, it would seem. This, in fact, is not the case; we can learn some valuable lessons here.

If you were asked to define the meaning of 'high quality' software, what would you say? How about:

- It should do its job correctly ('functional' correctness).
- It should do its job in the right time ('temporal' correctness).
- Its behaviour should be predictable.
- Its behaviour should be consistent.
- The code shouldn't be difficult to maintain (low complexity).
- The code correctness can be analyzed (static analysis).
- The code behaviour can be analysed (coverage analysis).
- Run-time performance should be predictable.
- Memory requirements should be predictable.
- The code can, if needed, be shown to conform to relevant standards.

You may, of course, extend this list to meet your own particular requirements.

Consider the application of these principles to the following small, relatively simple, real-time system, figure 1.1.

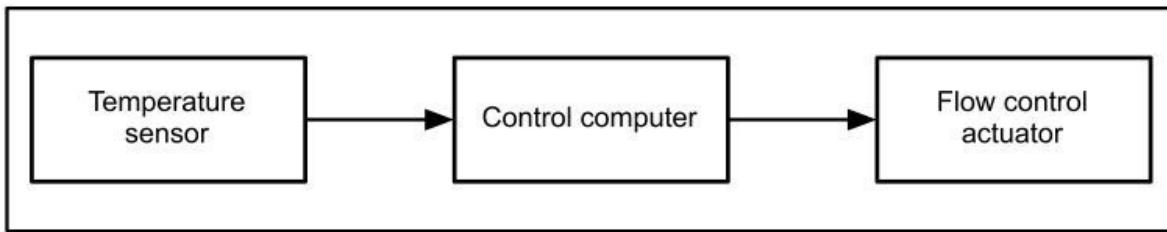


Figure 1.1 A simple processor-based real-time system

Here the requirement is to control liquid temperature by varying its flow rate. This is done by:

- Measuring the liquid temperature using a temperature sensor.

- Comparing this with the desired temperature value, and then
- Generating a control signal to set the position of an actuator that controls coolant flow.

The software has to perform:

- Data acquisition.
- Signal linearization and scaling.
- Control calculations.
- Actuator drive.

However, this is a safety-critical SIL 4 (Safety Integrity Level 4) subsystem within a nuclear reactor control system. As such the use of interrupts is forbidden.

There is no unique code solution, but it certainly will be of the following form (listing 1.1):

```
Loop;
  MeasureTemperature;
  LinearizeSignal;
  ScaleSignal;
  ComputeControlSignal;
  SetActuatorPosition;
  DelayUntilTime = xx milliseconds;
Goto Loop;
```

Listing 1.1

What we have is an example of 'application-level' code, here formed as a single sequential program unit. Note also that low-level details are hidden from us.

In the main, low-level operations are concerned with system hardware and related activities. Even if a high-level language is used, the programmer must have expert knowledge of the machine hardware and functioning. And that highlights one of the issues related to conventional programming of micros: the

hardware/software expertise required to achieve good designs. Even for this simple example the programmer needs a considerable degree of hardware and software skills.

1.3 Modelling the software

There are a number of distinct stages in producing a running program, figure 1.2.

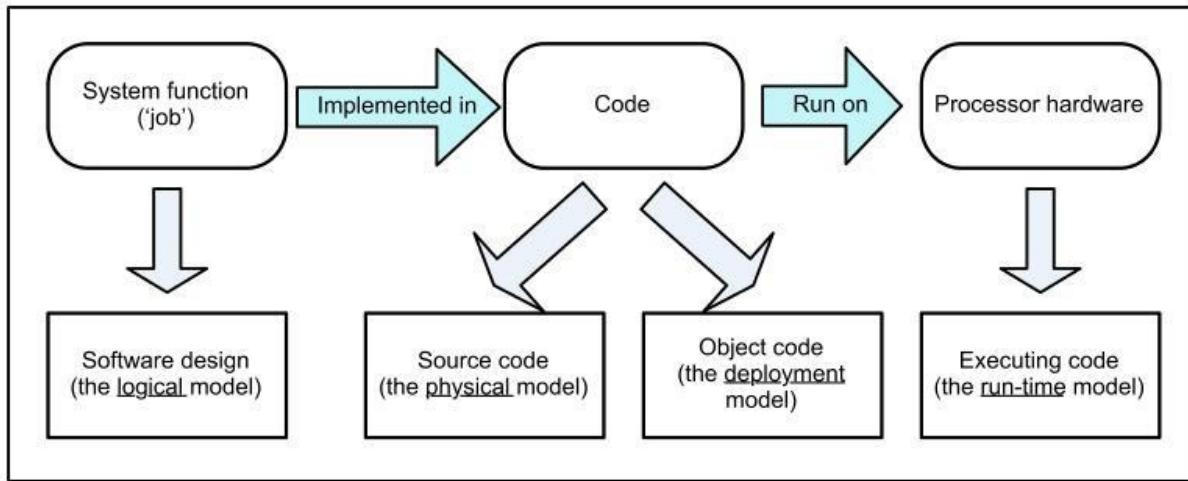


Figure 1.2 From design to run-time.

First there is the design of the software, resulting in the design or logical model. This is then implemented as source code, the physical model. The source code is compiled, linked and located ('built') to give object code, the deployment model. Finally the object code is loaded onto the processor and then executed, the run-time model. Note that in many modern integrated development environments (IDEs) building and downloading are handled as a single activity.

Here our main concerns are the code and run-time models. For the moment we'll put the design model to one side and concentrate on the other two.

Figure 1.3 shows the essential elements of the code model.

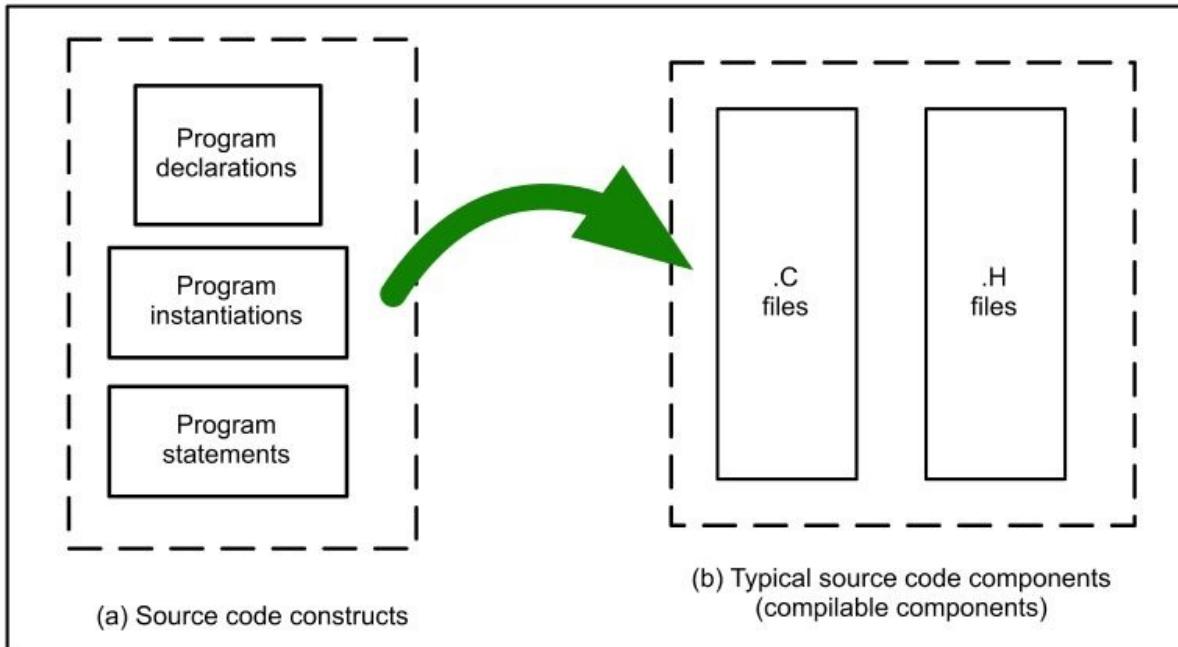


Figure 1.3 The essentials of the code model.

Developing and handling this is very easy when using modern toolsets. However, what can't be automated is the allocation of the source code constructs to the source code components. It is up to the programmer to make these key decisions, a topic we'll return to when dealing with the design model.

The code model gives us a static view of the software. In contrast the run-time model, a combination of code, data and processor, figure 1.4, represents software in execution. This is defined to be a software process, also known as a task in the embedded world (the terminology is contentious and more will be said later on this issue). Put simply, a task represents the execution of a single sequential program.

For the time being we'll call the run-time model the tasking model.

Changes to the code model may (usually do) affect the tasking model. Thus it is essential that the developer fully understands and documents their relationship.

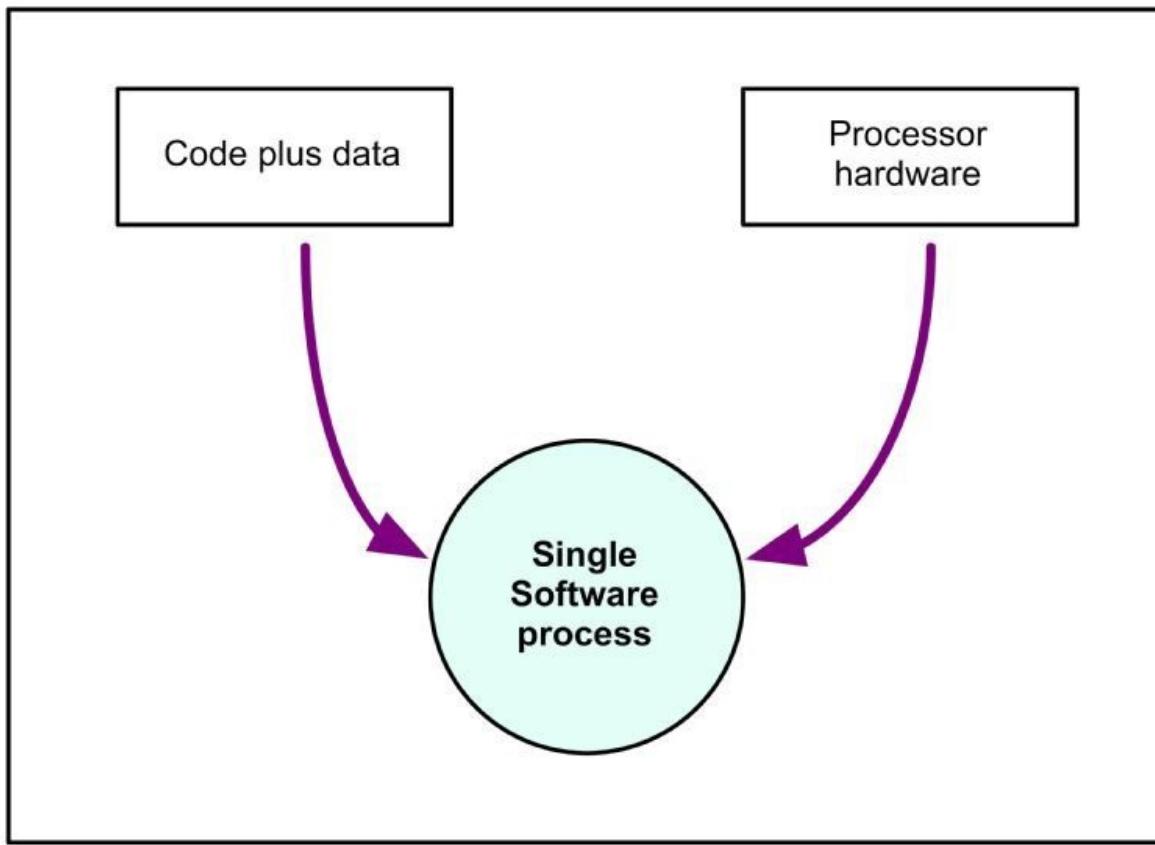


Figure 1.4 The software process run-time model.

1.4 The importance of time and timing

Now for a brief digression to hammer home a point crucial to the design of embedded systems. Figure 1.5 shows three generations of aircraft, each type having a three-axis autostabilizer system. Fundamentally the real-world job to be carried out is the same



Figure 1.5 Control systems - three generations of enabling technologies.

for each aircraft yet the enabling technologies are quite different. And the essential difference lies between the microprocessor-based system and the other two: discrete versus continuous operation.

In continuous (“analogue”) electronic systems all operations can, if required, take place simultaneously (“concurrently”). Not only that, but processing is done instantaneously. But this just isn't possible in processor-based systems as these are fundamentally discrete in operation:

- A processor can do only one thing at a time (a sequential machine) and

- Operations take time - things don't happen instantly.

These are the two factors that give us so much angst in our design work. And that's why if you begin development without knowing your system timing needs, be prepared for nasty surprises.

With this in mind, let's revisit the earlier design exercise, figure 1.1. It can be seen that the task code:

- Is executed in a continuous loop.
- Completes its work on each loop ('run-to-completion' semantics).
- Has a deadline for work completion (T_d).
- Takes time to complete its work (task execution time - T_e).
- Is repeated at regular intervals or periods ('periodic' operation - T_p).
- Does nothing while it's delayed (its spare time T_s).
- Needs a timing mechanism to control the periodic time (here we're most likely to keep track of things by using a hardware timer).

In this particular design T_p and T_d are system requirements, T_e depends on our code solution and T_s is what we end up with. For example, suppose that T_p is 100ms and T_e is 5ms; this gives a T_s value of 95 ms. It can be seen that the processor is executing code for 5ms in every 100ms, a Utilization (U) of 5%. These timings are depicted in figure 1.6.

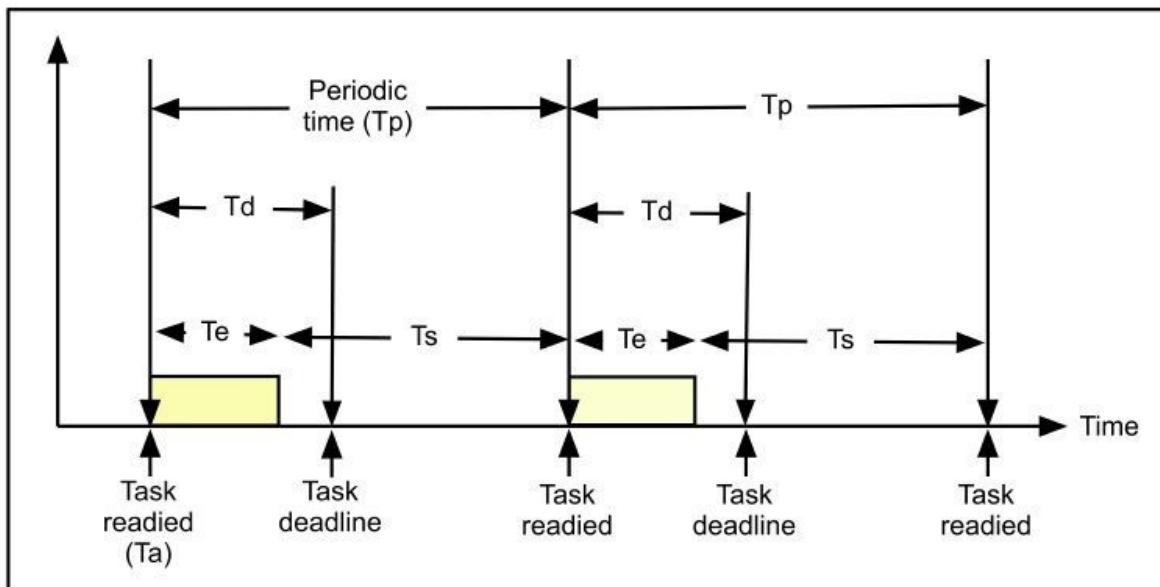


Figure 1.6 Task timing - some basic definitions.

What are the design implications of these figures? First, in embedded systems, tasks are intended to run to a completion point each time they're activated. Second, a practical system must have some spare time. Third, just because the code can be executed within a period doesn't mean its performance is acceptable. The input-to-output processing delay ('latency') may cause problems with the behaviour of the overall system.

1.5 Handling multiple jobs

The code structure of the previous example is one that lets us build the best possible software. But, as shown, it consists of only one well-defined job housed in one program unit. So how well does this approach scale up? Is it possible, for instance, to match the quality of the single-job design when handling multiple jobs? Well, it all depends, as we'll see shortly.

Let us suppose that we are asked to produce the software for a microprocessor-based two-axis pan and tilt camera stabilizer system. Thus the overall job or function, 'stabilise the image', consists of two sub-jobs, as shown in figure 1.7

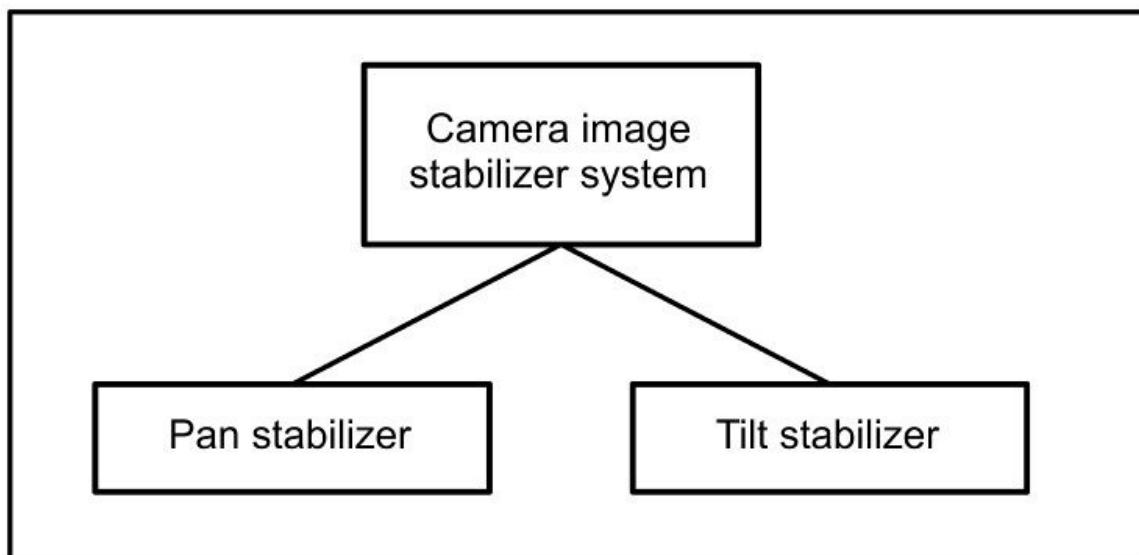


Figure 1.7 Functional description of an image stabilizer system.

Let us assume that the timing requirements for the axes are as follows:

Pan: 25 Hz sampling rate (25 times per second, $T_p = 40\text{ms}$).

Tilt: 50 Hz sampling rate ($T_p = 20\text{ms}$).

To develop the run-time software we can extend the approach used for the simple system, figure 1.8.

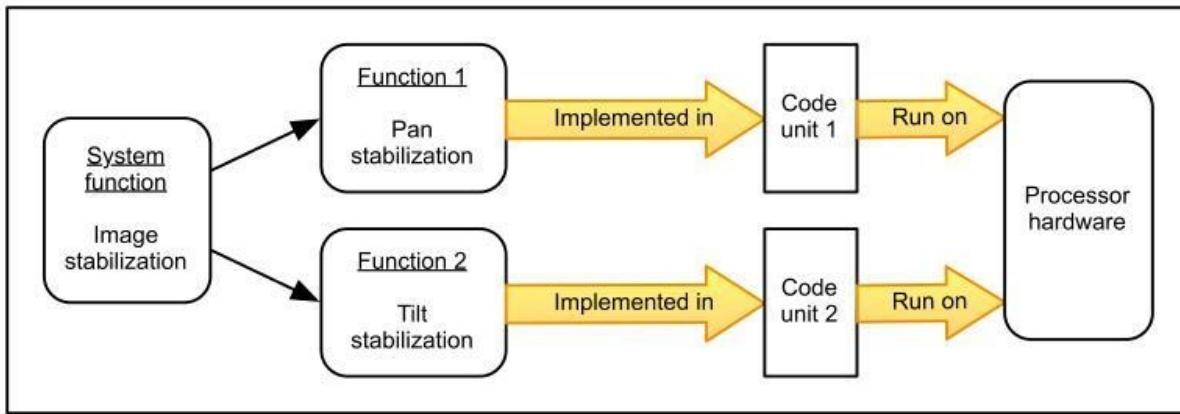


Figure 1.8 From design to run-time - multiple jobs

Now, however, we not only have to run the code units but also have to run them at different rates. This brings in the need to have some overall coordinator software: a form of execution control 'engine'. To implement this example in C-based software, we could take the approach depicted in figure 1.9.

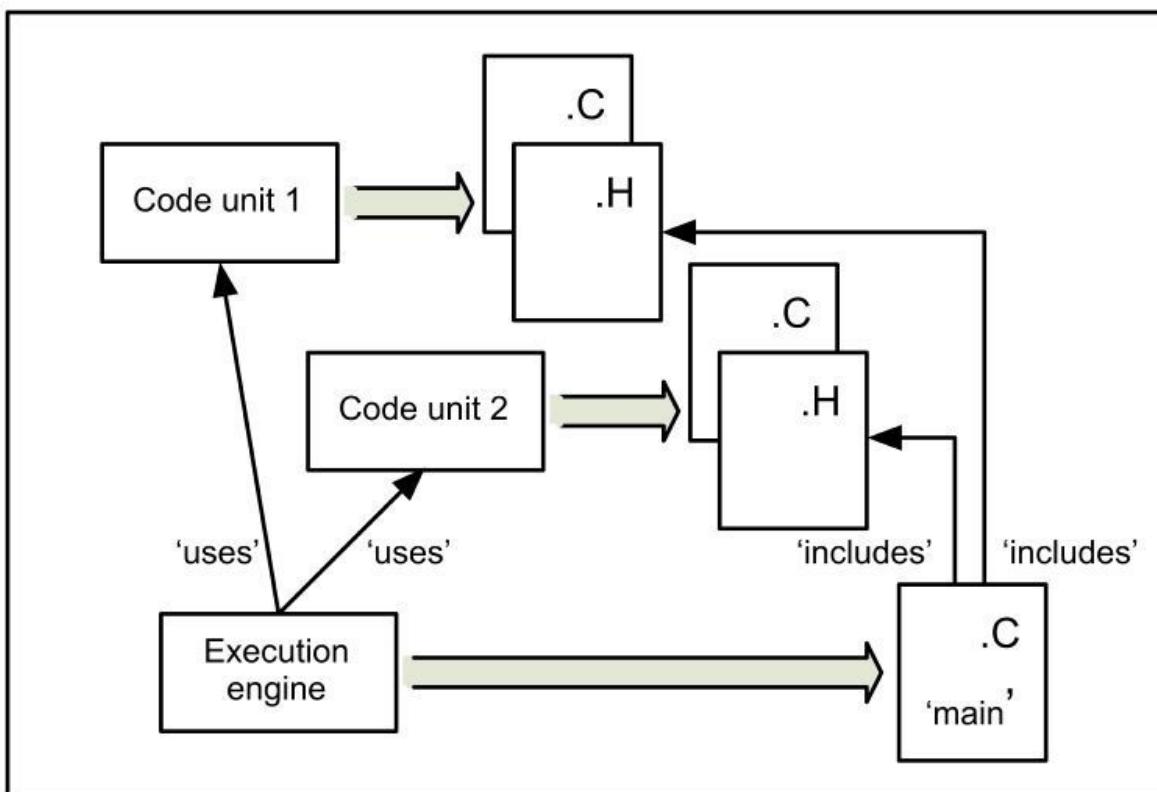


Figure 1.9 Code model for the camera stabilizer system.

Each code unit is implemented as a C function, e.g.

Code unit 1 (function 1): RunPanStabilizer();

Code unit 2(function 2): RunTiltStabilizer();

Execution engine: Implemented in the code of main (as for example in listing 1.2).

```
int LoopCounter      = 1;
int TwentyMilliseconds = 20; /* platform specific */

while(1)
{
    if (LoopCounter == 1)
    {
        RunPanStabilizer();
        RunTiltStabilizer();
        LoopCounter = 0;
    }
    else
    {
        RunTiltStabilizer()
        LoopCounter = 1;
    } /* end if */
    DelayUntilTime(TwentyMilliseconds);
} /* end while */
```

Listing 1.2

Please note; this is written for clarity, not efficiency.

What we've seen here is the basis for a sound design approach. However, it's still only a basis as it has, unfortunately for us, significant shortcomings.

1.6 Handling complex multiple jobs

Now examine a somewhat more complex system, figure 1.10, typical of many small-to-medium applications. Its primary function is to control the temperature in an engine jet pipe. This temperature is measured using thermocouples, the resulting analogue signal is digitized, an error signal is calculated internally in the computer, and a correcting signal is output to the fuel valves. The unit, however, is also required to perform several secondary jobs. First, the pilots need to have access to all system information via the keypad/display unit. Second, the flight recorder must be able to acquire the same information using a serial data link. Based on system information we now go ahead and develop the software system. The resulting model, based on a functional design approach, is that of figure 1.11.

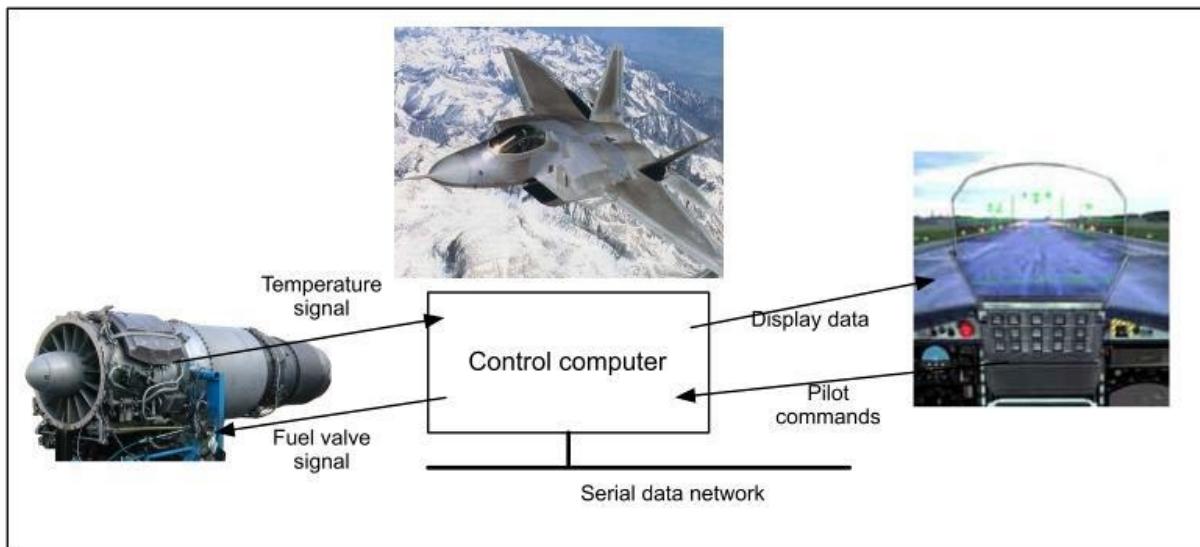


Figure 1.10 A more complex system.

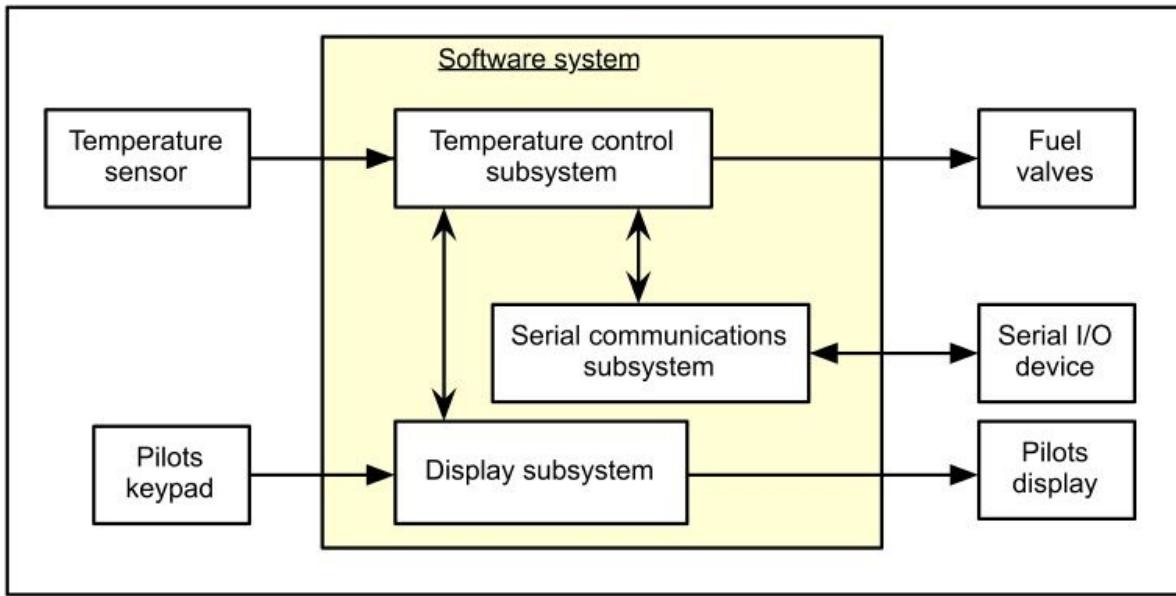


Figure 1.11 Example system - outline software design model.

The next stage in the development process is to implement each subsystem as an individual code unit. That's the easy bit. But trying to get the execution engine to run these units correctly is another matter; given the operational requirements stated above, such a solution is unlikely to work.

The problem stems from the inherent 'asynchronous parallelism' (independent concurrency) of the functions. What this means is that we have a number of distinct jobs, which, in the real world may have:

- To be serviced at regular intervals - periodic functions
- To be serviced at random (and not preset) times - asynchronous or aperiodic functions .
- To be processed simultaneously.
- Very different timing needs.

A look at the timing requirements of this system will show exactly why the earlier approach falls over.

Timing information.

- Control loop: Periodic, with a sampling rate of 10 Hz, i.e. a period of 100ms.

- Control loop: A predicted computation time of 5ms.
- Serial communications: The design here uses a 4-byte buffer receiver-transmitter (RT), has a data rate of 1 Mbit/sec and a message length of eight bytes. Note this means that the buffer will fill up in 32 microseconds when a message stream arrives.
- Display: The computer must have an acceptable response time to keypad operations. A value of 250ms should be quite sufficient.

If there is a buffer overrun in the RT unit, existing data will be corrupted. To prevent this happening current data has to be read before overrun occurs. If polling is used to detect RT data, then it must be done at a very high rate (every 32 microseconds to a first order approximation).

The result is that it just isn't possible to code the elegant software design model in a simple way. To use a well-known phrase, the wheels have come off the wagon of our implementation technique. So, what can we do to get us out of this hole? The answer, it turns out, is to change our execution engine; use the processor interrupt mechanism to drive the software.

1.7 Using interrupts as the execution engine - simple quasi-concurrency

A timer times out. A switch is pressed. A peripheral device needs attention. These are typical of real-world events in computer-based systems. But exactly how does the software know that things have happened in the real world? There are, in fact, only two ways to detect such events: look for the events (polling) or signal directly to the processor (hardware interrupts).

In the previous work we've used polling within a single, sequential program unit. Now we'll look at hardware interrupts, a signal that is applied electronically to the processor. When the interrupt is generated it invokes a predetermined response (as defined by the programmer) to execute some specified code. Thus the interrupt can be seen to be the code execution engine of a particular piece of software. Applying this technique to the engine temperature control system results in the run-time model of figure 1.12 (please note that this is based on our experiences and prejudices; design solutions are not unique).

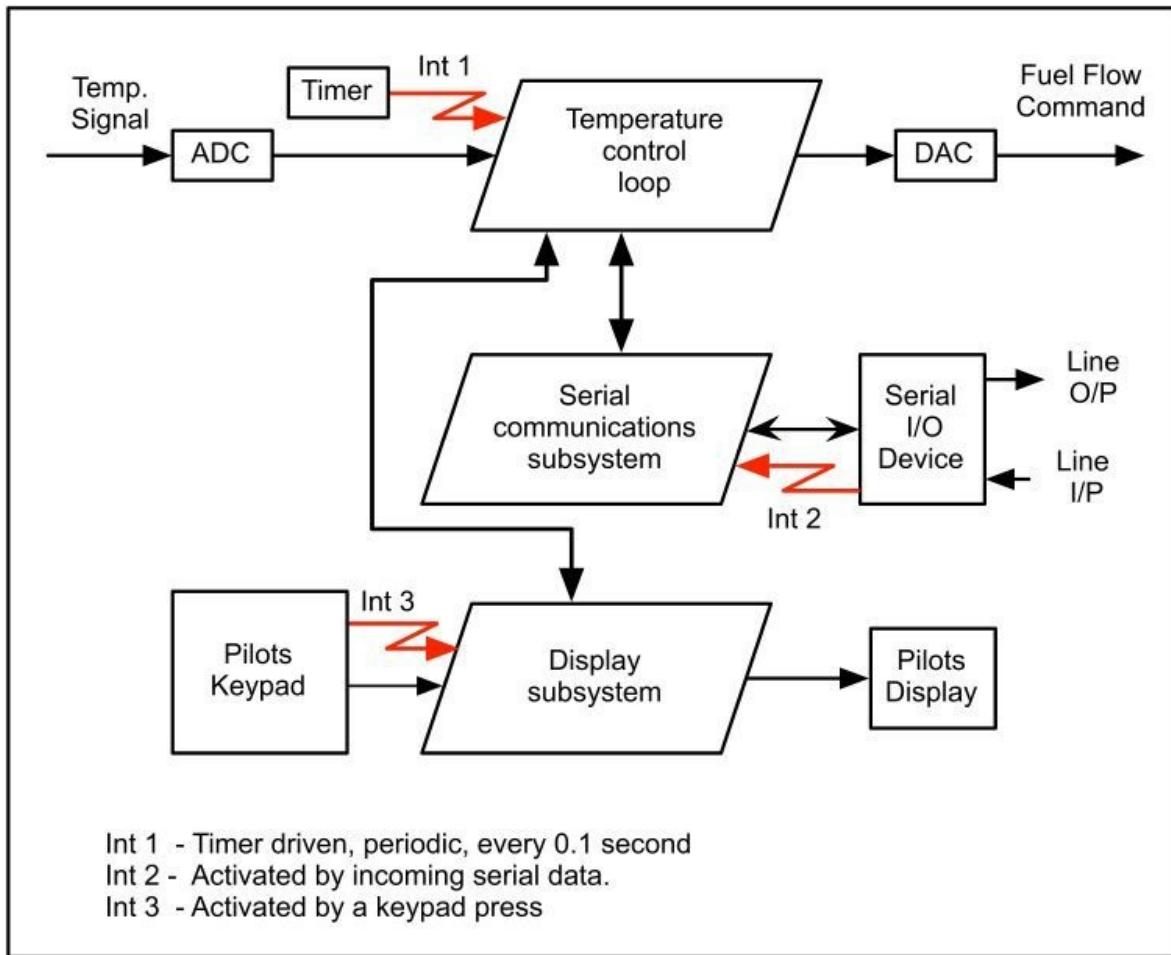


Figure 1.12 Example system run-time model - interrupt-driven design.

Here the interrupts, shown by the zig-zag arrowed lines, represent the source of the interrupting signals. Interrupt 1 is produced, for example, by an output logic signal from a timer chip. When activated, the temperature control loop starts executing, runs to completion, then stops, waiting for the next interrupt. Interrupt 2 is also generated by a logic signal (one produced by the RT chip), this invoking the communications code. Interrupt 3, which occurs whenever a button is pressed on the pilots keypad, causes the display code to be executed.

What we now have are a set of software subsystems, activated by interrupts and cooperating with each other. If this design is implemented on a single-processor system then the code units cannot execute concurrently; they must work in a time-shared manner. Yet from a real-world point of view, if they appear to run concurrently, time-sharing isn't a problem. Another point to bear in mind is that when each code unit runs it has complete use of the computer resources; it

appears to 'own' the processor. This means that we can design the software as if each function will run on its own processor, the 'abstract' processor.

So what we have here is the apparent concurrent execution of a set of individual processes (tasks), so-called 'quasi-concurrency'. In other words, by using interrupts, we can run multiple tasks 'simultaneously': a form of simple or 'poor man's' multitasking. The key conclusions to be drawn from this section are that:

- The interrupts are key-enabling mechanisms to make the design work.
- Interrupts simplify the functional design of the system.
- Interrupts simplify the handling of differing timing and response requirements.
- Actual run-time behaviour cannot (in most cases) be predicted or statically analysed. This is one of the most important issues in concurrent software design, and will figure strongly in later work.

With interrupt-driven tasking we can map logical models to code in a straightforward way, so preserving the design structure. But the downside is twofold. First, to apply it effectively, developers must be highly skilled in both hardware and software terms. Second, the way in which tasks are time-shared is very restrictive; when employed by inexperienced designers the resulting behaviour and performance can leave much to be desired.

The central function of an operating system is to remove this burden from the code writer (all other OS features follow from this). It screens the complexities of the computer from the programmer, leaving him to concentrate on the job in hand. Detailed knowledge of interrupts, timers, analogue-to-digital converters, etc is no longer needed. As a result, the computer can be treated as a 'virtual' machine, providing facilities for safe, correct, efficient and timely operation. In other words, it makes life easy (or at least easier).

1.8 Basic features of real-time operating systems

Taking into account the factors discussed above, our hardware and operating system software must support:

- Task structuring of programs.
- Task implementations as logically separate units (task abstraction).
- Parallelism (concurrency) of operations
- Use of system resources at predetermined times.
- Use of system resources at random times.
- Task implementation with minimal hardware knowledge.

These apply to all operating systems; but that doesn't mean that all OSs are designed in the same way or with the same objectives.

In all computer applications, two major benefits stem from using commercial operating systems: reduced costs and increased reliability. Nevertheless, the way machines are used has a profound effect on the design philosophy of their OSs. For instance, a mainframe environment is quite volatile. The number, complexity and size of tasks handled at any one time are probably unknown (once it's been in service for a while, that is). In such cases a primary requirement is to increase throughput. On the other hand, in embedded applications, tasks are very clearly defined. The processor must be capable of handling the total computer loading within quite specific timescales (if it doesn't, the system is in real trouble). Therefore, although the OS must be efficient, we are more concerned with predictability of performance. Moreover, reliability of operation is paramount.

Now let's take a more detailed view of the problem. Consider again our example system which has three major tasks: JPT control, flight recorder interfacing and pilot interfacing. Each one could be run on a separate processor, i.e. multiprocessing. In such a small system this would be expensive, complex and technical overkill. Therefore one processor only is used. This, in fact, is the situation in most embedded systems. In this text, single processor multitask designs are called 'multitasking' systems, to distinguish them from multiprocessing techniques (this definition isn't quite correct, but fits in with common use).

What we have then are three separate but interdependent tasks. This raises a number of interesting problems, the solutions being provided by the multitasking software of the operating system, figure 1.13. First, we have to decide WHEN

and WHY tasks should run - 'task scheduling'. Then we have to police the use of resources shared between tasks, to prevent damage or corruption to such resources - 'mutual exclusion'. Finally, as tasks (in this example) must be able to 'speak' to each other, communication facilities are needed - 'synchronization and data transfer'. In a multitasking system it is possible to have tasks that perform independent and separate functions, i.e. are functionally independent. A simple example of this is the implementation of several separate control channels on a single board digital controller. These tasks proceed about their business without any need to communicate with each other. In fact, each task acts as if it has sole use of the computer. But that doesn't necessarily mean that each task has its own resources; there may still be a need to share system facilities. For instance, each control channel may have to report its status regularly to a remote computer over one shared digital link. At some point there will be contention between the tasks for the use of this resource. It is necessary to resolve any such contention, usually by the use of mutual exclusion features.

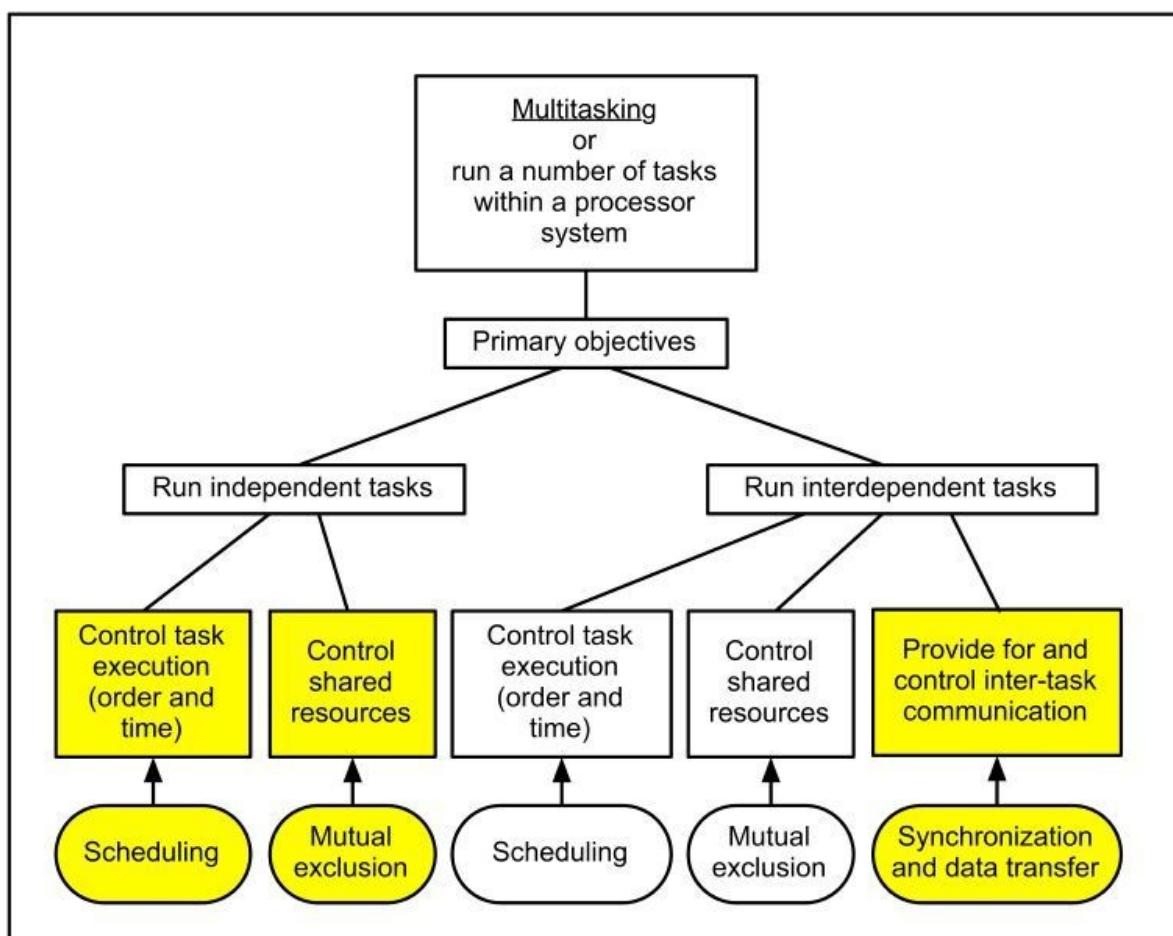


Figure 1.13 Objectives of multitasking software

1.9 Executives, kernels and operating systems

We haven't yet said what an OS is. One starting point is the Oxford Dictionary of Computing definition as an OS being 'the set of software products that jointly controls the system resources and the processes using these resources on a computer'. Nevertheless, trying to define precisely what an OS is and how it is constructed is more difficult. Many have similar overall structures but differ, often considerably, in detail. Embedded operating systems are smaller and simpler than mainframe types. The structure shown in figure 1.14 is typical of modern designs, formed from a relatively small software set. Note that it consists of a series of well-defined but distinct functions.

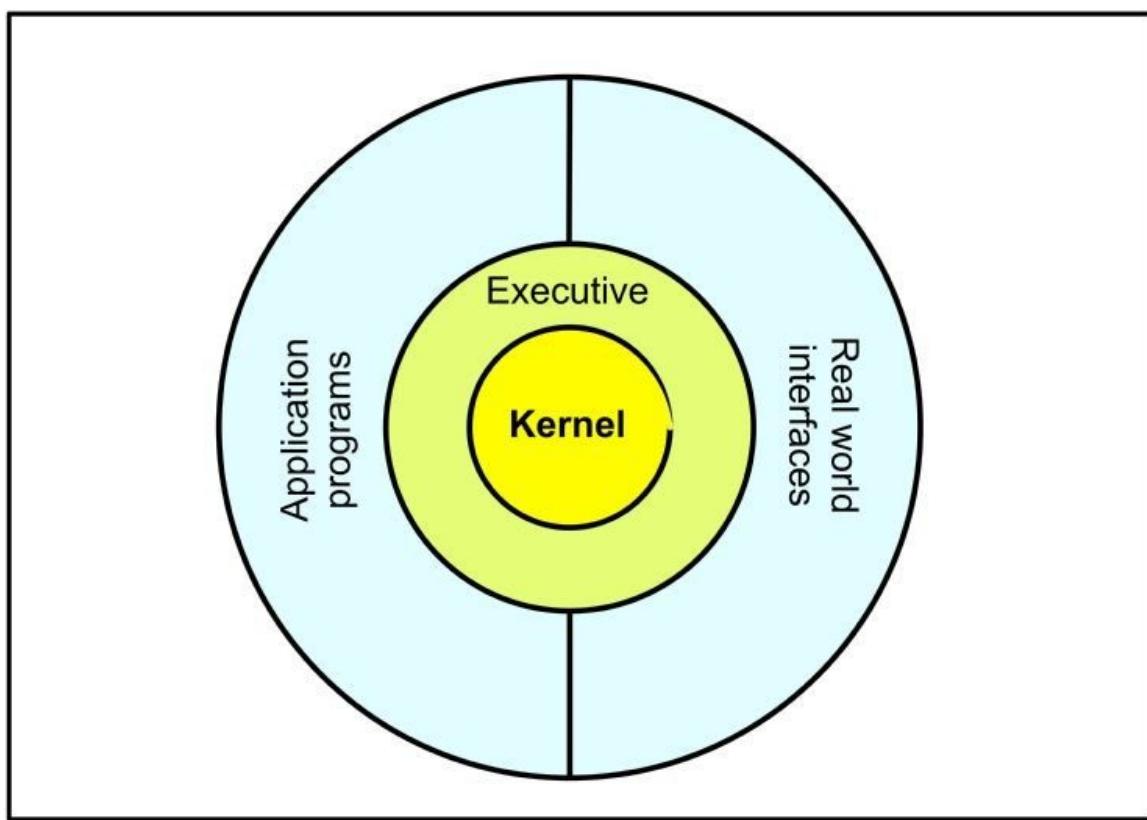


Figure 1.14 Overall operating system structure - simple model

First consider the second ring, the 'executive'. This is where control functions are concentrated, the executive being the overall controller of all computer programs. User tasks (programs) interface with other system activities (including other tasks) via the executive. This is quite different from a sequential (single 'thread' of execution) design where task actions are distributed throughout the

program. Here each task is written separately, calling on system resources via the executive. The executive itself directly controls all scheduling, mutual exclusion, data transfer and synchronization activities. To do this it calls on detailed facilities provided by the inner ring, the kernel. In a sense, the executive behaves as the manager, the kernel as the executor. At this stage the detailed functions of the executive/kernel combination won't be spelt out; instead they will be developed as we go along.

One part of the outer ring, labelled 'application programs' is self-explanatory. These use the RTOS software by calling on the resources via application programming interfaces (APIs). The other part, 'real-world interfaces', consists of software that handles the hardware of the system. Such hardware, which varies from design to design, is driven by standard software routines. This typically includes programmable timers, configurable I/O ports, serial communication devices, analogue to digital converters, keyboard controllers, and the like. Tasks can access I/O devices directly; more commonly they interact with the real world via OS provided routines.

In summary then, each task may be written as if it is the sole user of the system. The programmer appears to have complete access to, and control of, system resources. If communication with other tasks is required, this can be implemented using clear and simple methods. All system functions may be accessed using standard techniques (as developed for that system); no knowledge of hardware or low-level programming is needed. Finally, the programmer doesn't have to resort to defensive programming methods to ensure safe system operation.

How well we achieve these goals depends entirely on the design of the executive and kernel.

1.10 Task-based software design - a recap

Six steps are involved in going from the specification to the running of a task-based design. The first one, step 1, defines clearly the extent of the system to be implemented. This establishes the overall system level view of the problem, i.e. its overall function, figure 1.15.

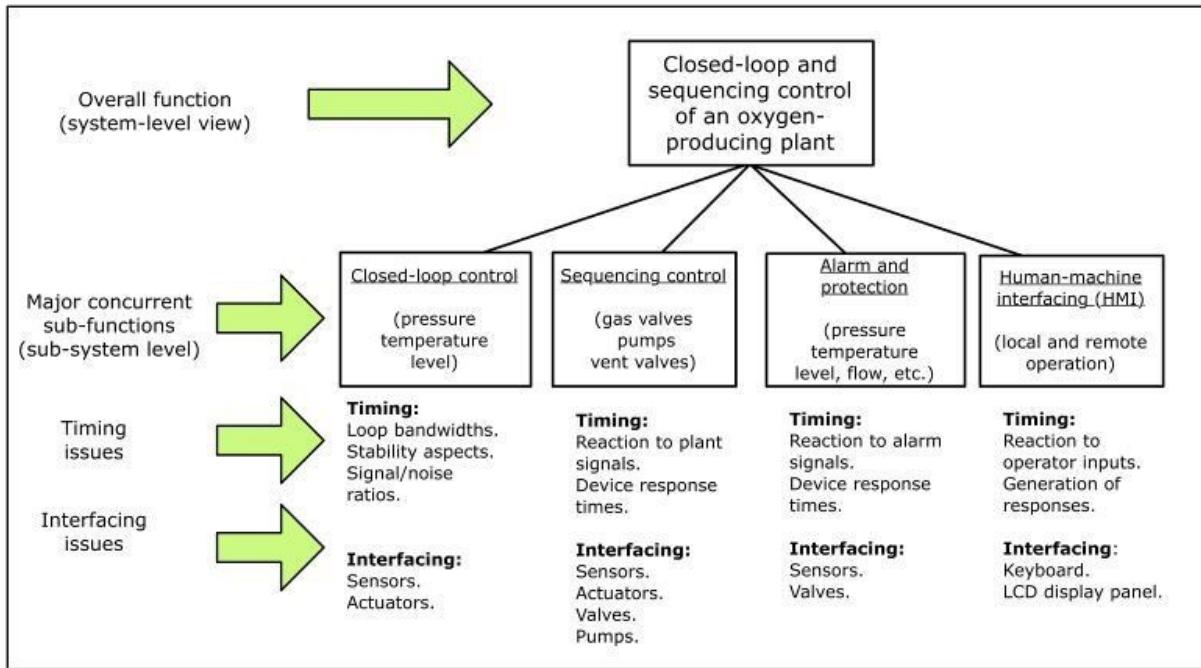


Figure 1.15 Structuring software - system functions and sub-functions

Step 2 identifies the major concurrent subsystems or sub-functions of this system. In this example the overall system consists, in our opinion, of four subsystems. Note well; this is a personal design decision, which depends entirely on the views of the designer.

Next, in step 3, the key timing and interfacing aspects of the system are specified. This is an essential part of the process, one critical to the development of real-time software. It's not unusual that such information is incomplete when design begins, but all unknowns must be explicitly noted.

Step 4 consists of developing the design and code models of each task based on the idea that:

- Each concurrent subsystem runs on its own processor (or set of processors)
- The supporting processors will provide all required computing resources ('abstract' or 'virtual' processors, figure 1.16).

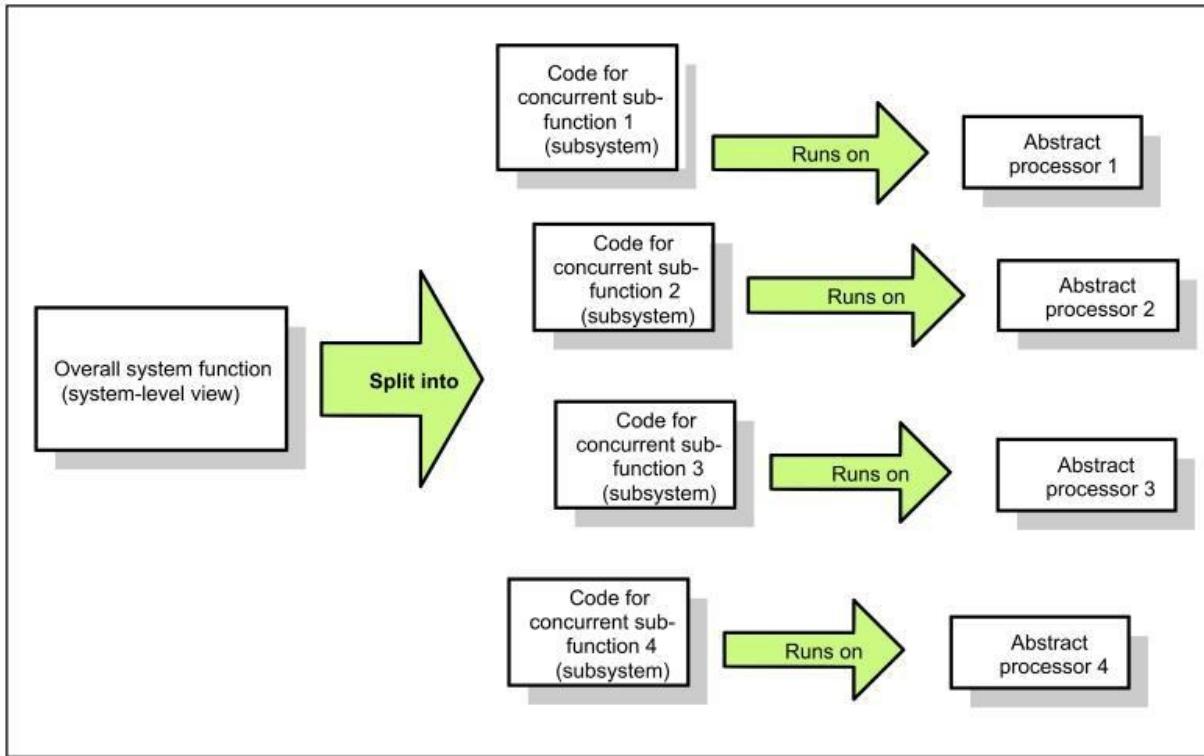


Figure 1.16 Task development - an ideal (abstract) solution

Step 5 associates each code unit with a specific abstract processor to form what we'll loosely call an abstract task. And this is where the RTOS first enters the development process. To produce an abstract task we employ the task creation software of the RTOS, typically using an API named 'CreateTask'. What this does is create a named task unit and link the function code with this unit. In its most basic form the code for this is: CreateTask (NameOfCodeUnit, DesignatedNameOfTask);

where DesignatedNameOfTask, the name used by the OS, represents the abstract task.

This must be done for all tasks. For the example above we would expect to find the following (or something similar) in the set-up code, listing 1.3:

```
CreateTask(ClosedLoopControlFunction, "ClosedLoopControllerTask");
CreateTask(SequencingControlFunction, "SequencingControllerTask");
CreateTask(AlarmProtectionFunction, "AlarmProtectionTask");
CreateTask(MMIFunction, "MMITask");
```

Listing 1.3

And note; these tasks generally cooperate and communicate with each other. An abstract task becomes a real one when it has use of the machine resources, figure 1.17.

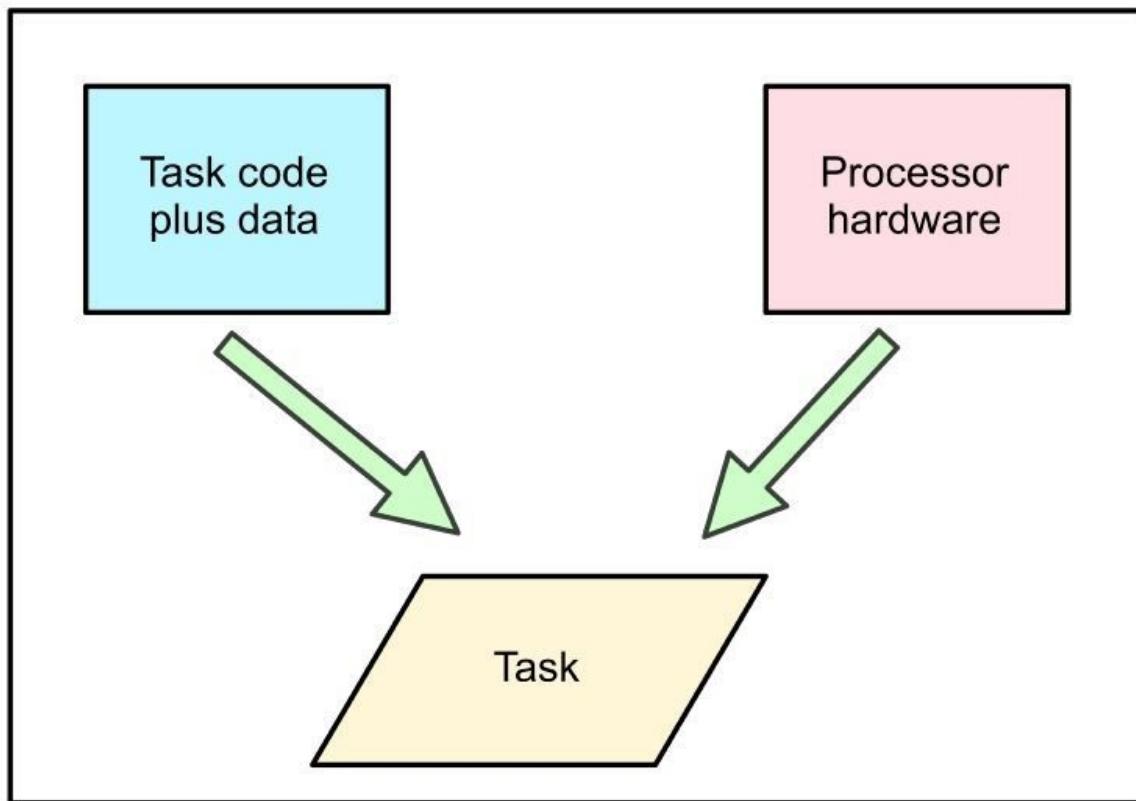


Figure 1.17 Task - syntax and semantics

Thus, strictly speaking, each task depicts the execution of a single sequential program (compare figure 1.17 with figure 1.4).

Step 6 runs each abstract processor on real hardware. Once again we use RTOS software to start and run the tasks, using a command of the form: StartTasks();
Please note that APIs and their formats are RTOS-specific.

Our task set could be run on various hardware configurations including (figure 1.18):

- A single processor (a uni-processor solution) or

- A number of processors (multiprocessor, multiple processor or multi-core units).

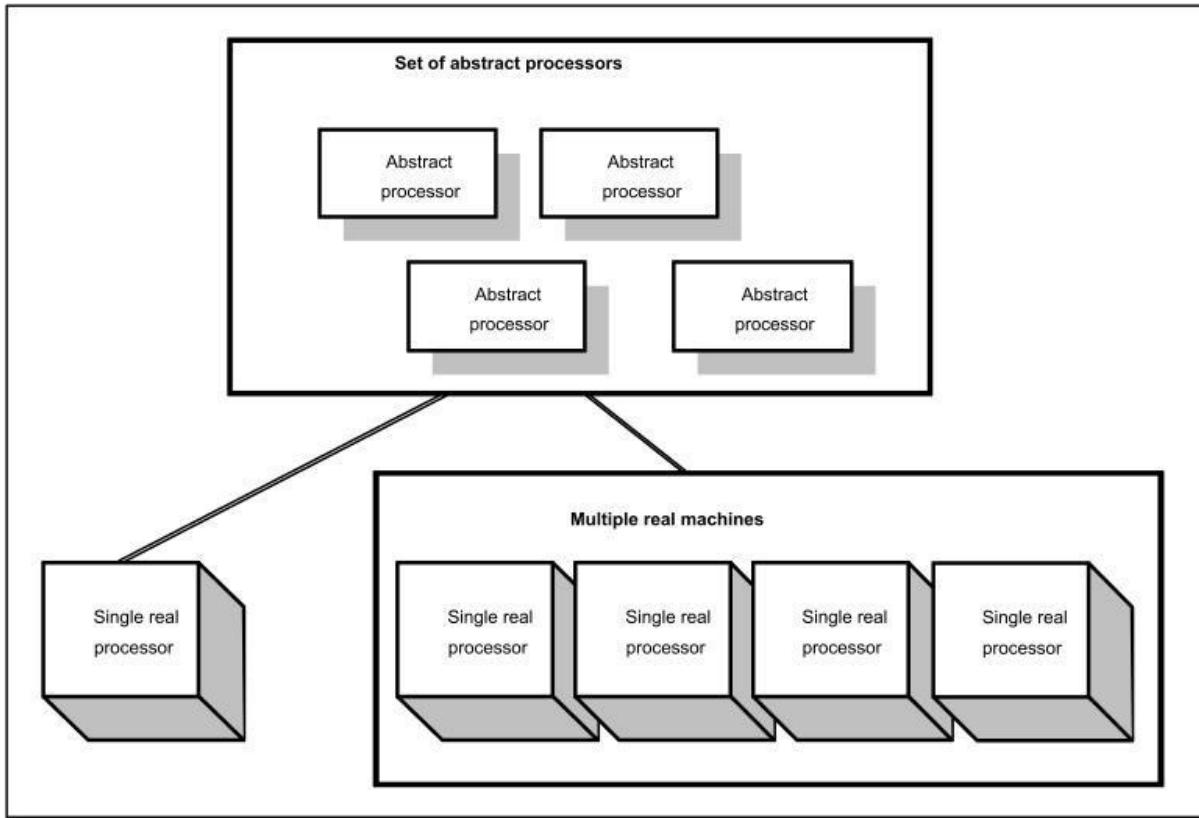


Figure 1.18 Platforms for abstract processors

Clearly, where a number of abstract processors are housed on one machine, they must run in a time-shared manner. And that defines the central responsibility of the operating system: running tasks in a time-shared manner.

When the hardware platform consists of multiple real machines the situation is somewhat more complex. We'll deal with that later in a separate chapter; for now let us concentrate on RTOS issues in single processor systems.

Review

You should now:

- Know the attributes of high quality software.
- Know how to develop high quality software.
- Understand why such code is easy to use in simple applications.
- Understand why such code is difficult to use in complex applications.
- Appreciate the nature and use of the logical, physical and deployment models of software.
- Realize how important time and timing data are in embedded systems.
- Be able to define the meaning of process and task.
- Recognize the difference between periodic and aperiodic tasks.
- Understand what concurrency, quasi-concurrency and multitasking are.
- Know that interrupt-driven designs provide a simple form of multitasking.
- Realize that, in multi-tasking implementations, overall behaviour cannot usually be determined statically; run-time behaviour is unpredictable.
- Understand the basic features of RTOSs.
- Understand the fundamentals of multi-tasking design.

Chapter 2 Scheduling - Concepts and implementation.

The objectives of this chapter are to:

- Explain the basic ideas of task scheduling.
- Describe these in the context of simple cyclic, timed cyclic and cooperative scheduling techniques.
- Show the essential difference between pre-emptive and non-pre-emptive scheduling methods.
- Explain the rationale, use and importance of having task priorities.
- Introduce the type and usage of the various queues met in multitasking designs.
- Introduce the task control block and the process descriptor.
- Show how fast response and secure code sharing is attained in task-based designs.
- Highlight the unpredictability of run-time behaviour.

2.1 Introduction

This section deals with the underlying concepts of scheduling, using an analogy to a real-world non-computer task. Let's make one restriction; it applies to single CPU systems only.

The problem is a simple (somewhat unrealistic) one. Assume that a firm has only one truck (the CPU), but has a number of drivers (tasks or processes). Only one driver can use the truck at any one time. Further, each driver is specialised in one, and only one, job. In these circumstances, what is the 'best' way for the transport manager to organise the delivery schedules (the 'scheduling' problem)?

2.2 Simple cyclic, timed cyclic and cooperative scheduling

One of the simplest solutions is that shown in figure 2.1, the 'simple cyclic scheduler'.

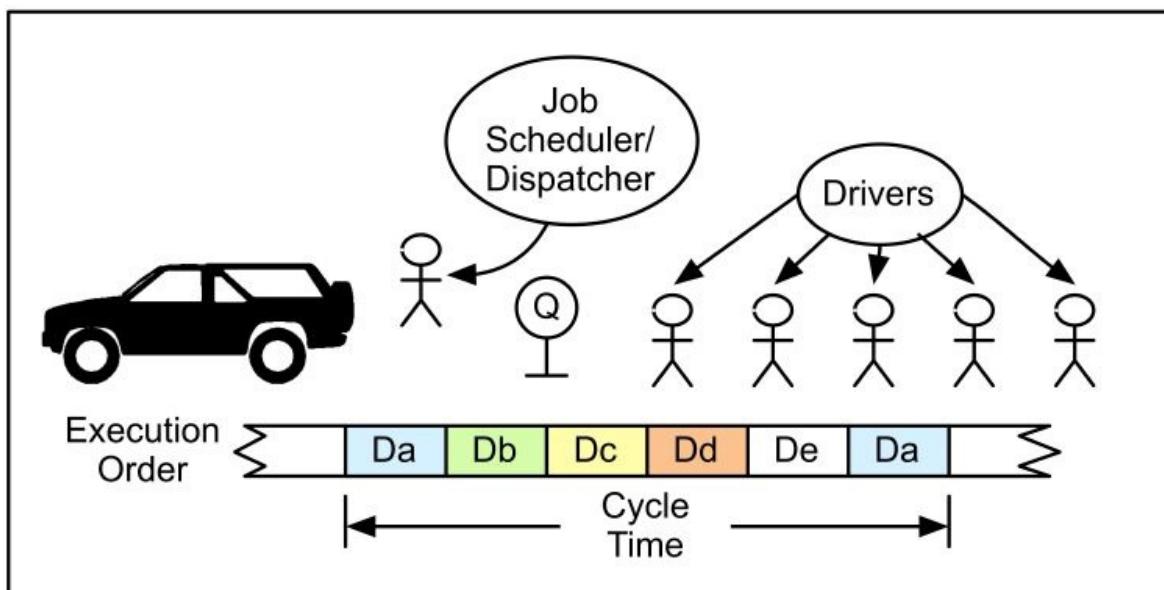


Figure 2.1 Simple cyclic (FIFO) scheduling

First, a queue of drivers is formed. The 'head of queue' is then given full use of the truck until he completes his job. It is then passed on to the next driver in the queue, who carries out his work, who then passes it on, and so on. All changeover activities are controlled by a scheduler/dispatcher (part of the executive); he does not, though, control task activities themselves. This method, described as first-in first-out (FIFO) scheduling, is most suitable for running tasks that:

- Aren't critical to the operation of the system.
- Aren't time critical and
- Run to completion each time they're executed.

This, however, is not typical of tasking operations in embedded systems. What we do need though are facilities to deal with task sets that:

- Are run repeatedly when the system is operational and
- Run to completion each time they're executed.

- Must be re-run at predetermined time intervals.

To achieve this we need to add a timing unit and another task: the do-nothing or 'idle' one (figure 2.2).

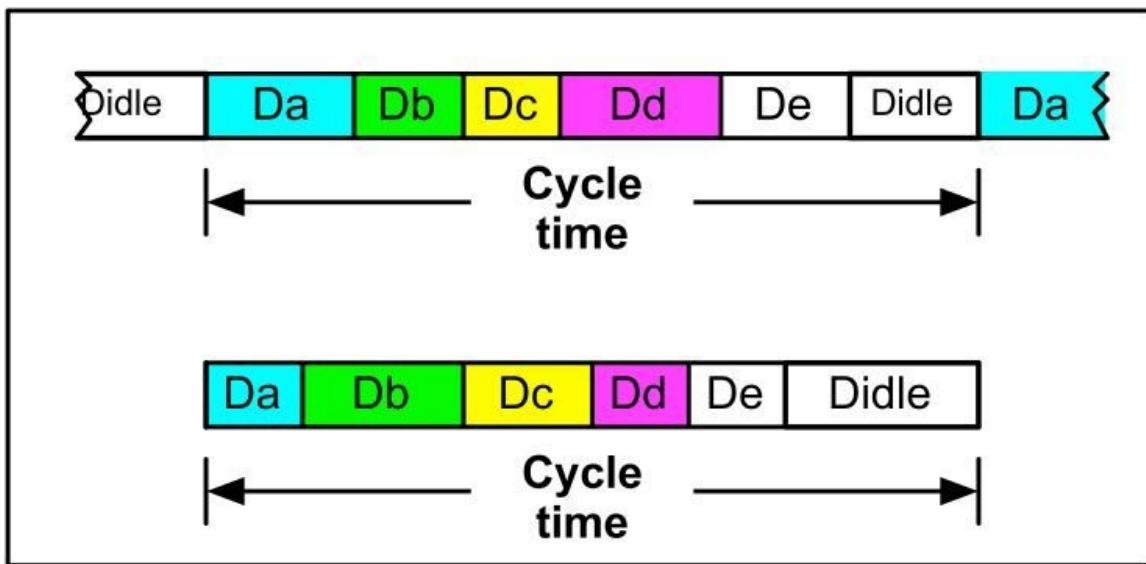


Figure 2.2 Timed cyclic scheduling

Using this, execution of the task set can be done at predefined intervals, the 'cycle time'. This is the responsibility of the timing unit. Note that without the idle task, variations ('jitter') of individual task execution times would also vary the cycle time.

With simple cyclic scheduling the processor is working flat out; its utilization (U) is 100%. With timed cyclic scheduling this isn't the case; we've sacrificed utilization to attain control of execution time. Here the utilization is:

$$U = ((\text{Cycle time}) - (\text{Idle time})) / (\text{Cycle time})$$

For example, if the cycle time is 10 ms and the idle time is 2ms, then the processor utilization is 0.8 (or 80%).

More generally U is measured over a specified test time interval (say, for example, one second)

$$U = (\text{Time spent executing tasks}) / (\text{Test time interval})$$

These simple scheduling schemes, while very easy to implement, have some significant limitations. First, consider what happens if a driver fails to return the truck (i.e. a task gets stuck). In the basic cyclic scheduler the system instantly

grinds to a halt. When timing control is used the situation is different, but just as serious. Here the dispatcher always retrieves the truck from a stuck task, thus restarting the cycle. Thus, each timed cycle begins correctly. But tasks only execute until the 'stuck' one is reached; following ones are never activated.

Second, a new driver (task) joining the system goes to the end of the queue. Accordingly, there could be a long delay before that job is carried out. In other words, the organisation reacts slowly for requests to run new tasks.

Third, system performance is affected mostly by long, not short, tasks. No matter how short tasks are, they still have to wait for their designated slot. Yet it might be that one short job could be done many times within the time slot of the long jobs. It might even be satisfactory to interleave the long job with multiple executions of the short one (figure 2.3). This usually makes better use of system resources (i.e. the truck or CPU). It isn't easy to implement this with simple FIFO scheduling so we normally employ a different method: cooperative scheduling.



Figure 2.3 Interleaving of jobs

The basic idea here is that a task 'cooperates' with others by explicitly giving up the processor; at that point another task starts executing. Central to this is that the decision is made by the task, not the scheduler. Its code must contain some form of 'yield' statement which, in the example above, also designates the next task to run. More will be said later on this point.

These limitations are unacceptable for most real-time systems, especially where fast and/or critical responses are needed. One improvement is to set timing constraints on task execution, the 'time slicing' approach.

Note that we very grandiosely call the task execution rules (e.g. FIFO) the scheduling algorithms.

2.3 Round-Robin (time slicing) scheduling

Here the scheduler/dispatcher is given a clock for the timing of jobs and the means to recall trucks (figure 2.4).

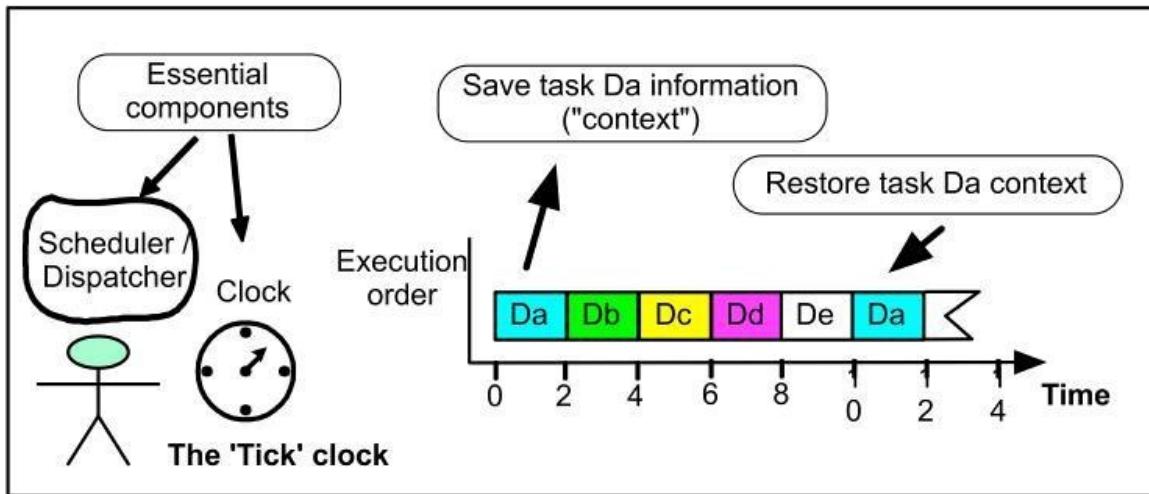


Figure 2.4 Time Slicing of tasks

Tasks are still carried out on a first-in first-out basis. Now, though, each driver is allocated a fixed time for use of the truck, the 'time slice' (here, two time units). When time is up, the truck must be passed on to the next user (task), even if the current task hasn't finished. This task is resumed at its next allotted time slot, exactly from where it left off. It is then run for a preset period of time, put into suspension, and so on. Note: the basic unit of time is called the 'tick'.

Such a process is called 'pre-emptive' scheduling. That is, the running task is replaced (pre-empted) by the next-to-run task. More generally, this applies whenever the resource (the CPU) can be taken away from the current user. Now this raises a problem not found in simple cyclic (non pre-emptive) operations, where tasks run to completion. In pre-emptive systems tasks don't necessarily finish in a single time slot; a number are needed. Stopping and restarting must be done without these being apparent to the user (i.e. 'transparent'). The only way to do this is to restart using precisely the conditions pertaining at shut-down. And, to achieve this, we must do two things. First, save all task information at the time of pre-emption. Second, restore this at the restart time (note that this saving and

restoring also applies to cooperative scheduling).

So, whenever a task is pre-empted, two extra operations take place. Initially, current information is stored away for later retrieval, then information relating to the new task is retrieved from its store. Such information is referred to as the 'context' of a task. Storing and retrieving is called 'context switching', the related time being the 'context switch time'. Context switching is an important factor in real-time operations because it takes up processor time. Consequently it reduces the available computing time, becoming a system overhead.

This scheduling, where tasks are dispatched FIFO for preset time slots, is also called 'round-robin' scheduling. Its advantages are improved responsiveness and better use of shared resources. But in practice it needs to be modified because tasks:

- Vary in importance
- Don't always run at regular intervals
- May only run when specified conditions are met

2.4 Task priorities

So far we have assumed that all tasks have equal status or priority. Hence the execution sequence is arbitrary, depending on how the system was set up in the first place. In reality a particular execution order may be required, and so tasks are allocated priorities (figure 2.5).

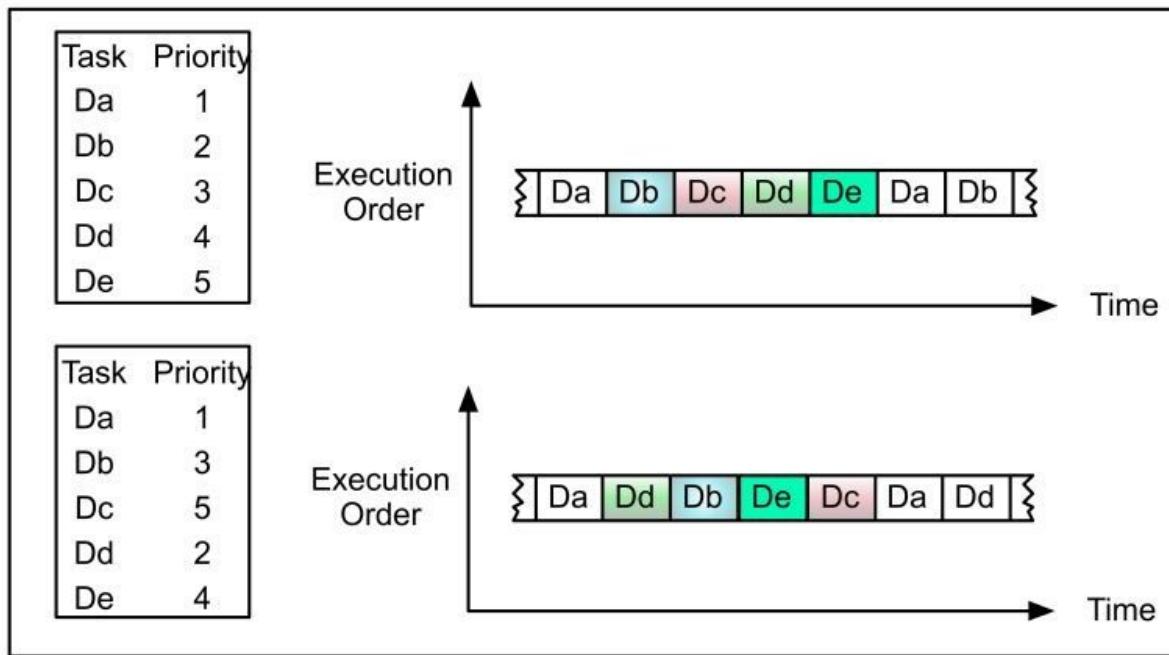


Figure 2.5 Setting task priorities

Here the lower the number, the higher the priority.

If this order remains fixed then we have a static priority scheme. If it can be changed during program execution then it is said to be dynamic. Priorities can be changed either by some external event or by a running task. Suppose, for instance, that during one run of task Db it changes the original priority order, setting De higher than Dd (figure 2.6). This doesn't produce an immediate effect; its consequences can be seen later on in the execution time sequence. Such implementations are defined as 'priority scheduling algorithms'.

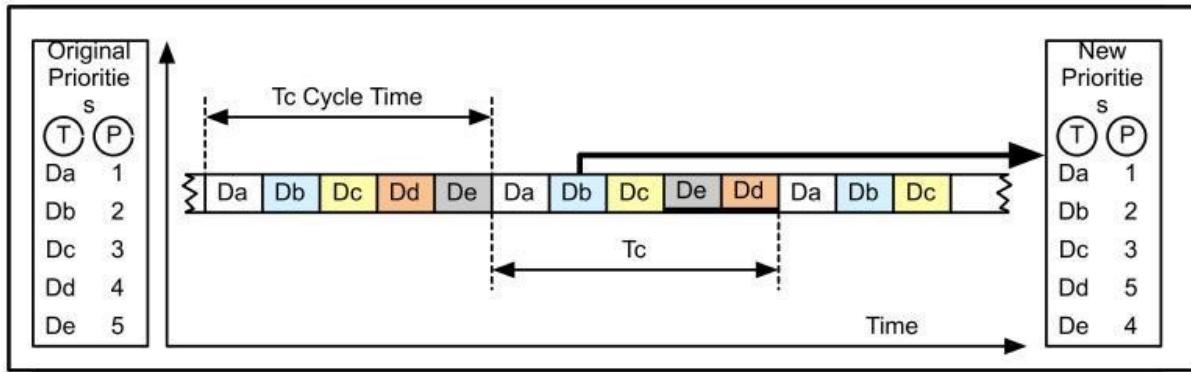


Figure 2.6 Dynamically changing priorities

The reason for using dynamic priority schemes is to improve flexibility and responsiveness. For example, when using cyclic scheduling, priorities may be changed where:

- In specific situations a task needs to very quickly act on the output of the running task, but isn't scheduled to run next.
- Systems have mode changes (e.g. from surveillance mode to tracking mode to engage target), each mode having specific schedules.

It is, however, more complex and also imposes a greater time overhead. Further, it has an inbuilt danger, that of blocking out tasks for long time periods. Observe that, when priorities are changed, the queue order is shuffled. And the task which next runs at the end of each time slice is that having the highest priority. Hence, if priorities continually change as programs run, low priority tasks could be blocked out. Not permanently, of course. They will eventually run at some time. But the responsiveness of such tasks may be pretty awful.

2.5 Using queues

So far only one queue has been used, consisting of all drivers, ready to perform their tasks as and when called upon. This we'll define to be the READY queue. In reality the situation is more complex because tasks aren't always ready. They may, for instance, have to wait until specific conditions are met before becoming runnable. While 'not-ready' they are said to be in the blocked or suspended state, and are held in the waiting queue (figure 2.7).

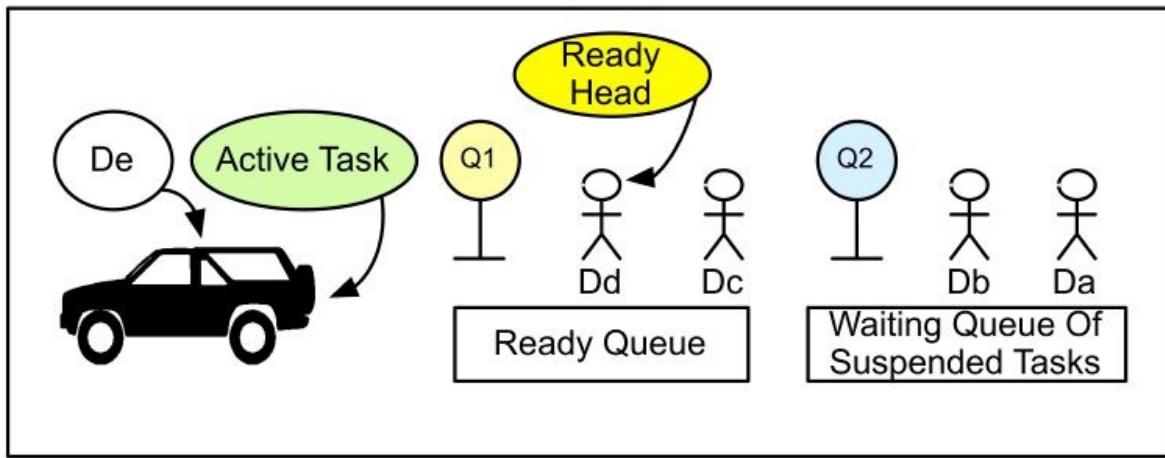


Figure 2.7 Ready and suspended task states

This is still a simplified situation because, in practice, it is normal to use a number of suspended queues (figure 2.8).

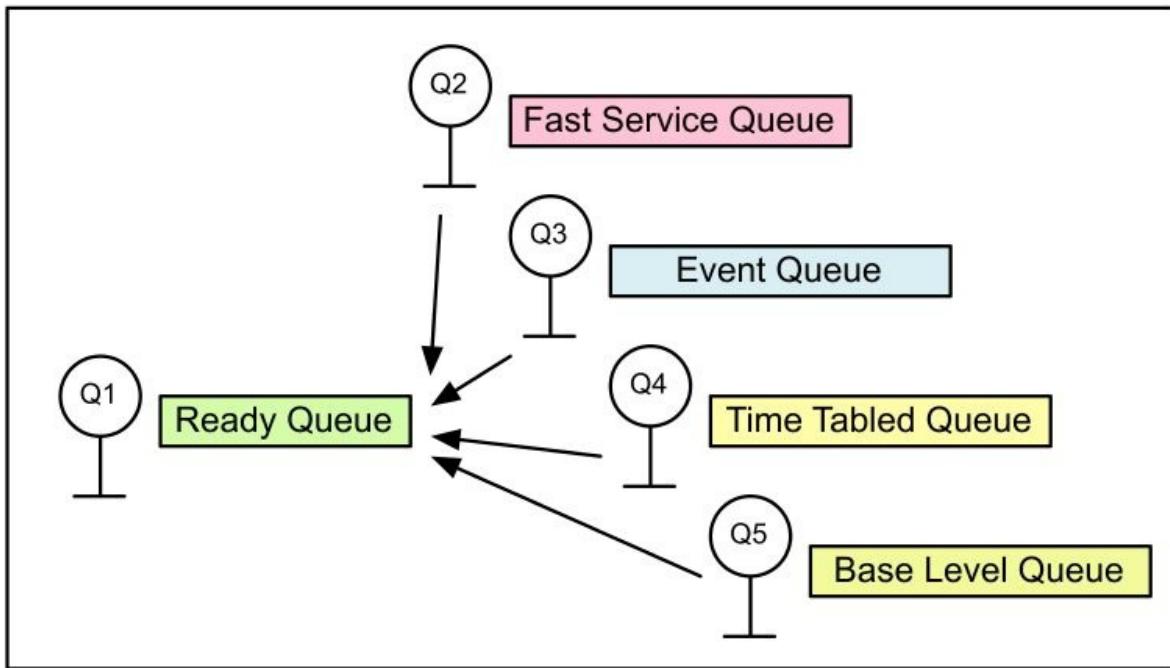


Figure 2.8 Types of queue

First there are task requests that must be serviced very quickly, a 'fast service' operation. An example of this is the need to respond rapidly to incoming signals on serial communication lines.

Second, some tasks are suspended until specific events occur, as with keyboard handling tasks. These would normally be left blocked until a key is pressed (the 'event').

Next there are tasks that have to be run at regular, predetermined intervals, that is, timetabled jobs. Measuring sensor input signals, for instance, is something that would be done in this way.

Finally, jobs that don't fall into any of these categories are done only when free time is available. These are defined to be 'base level' tasks (frequently these are always ready to run). As an example, the updating of non-critical display data could very well operate like this.

2.6 Priority pre-emptive scheduling

We've already met the scheduling of priority-based non-pre-emptive tasks; now for priority, pre-emptive scheduling. In this case a running task can be switched out if a task of higher-priority is readied.

From the preceding information, we can see that a task may be in one of three states: running (executing), ready or suspended. With priority pre-emptive scheduling task behaviour over time can be modelled using a state transition diagram, figure 2.9.

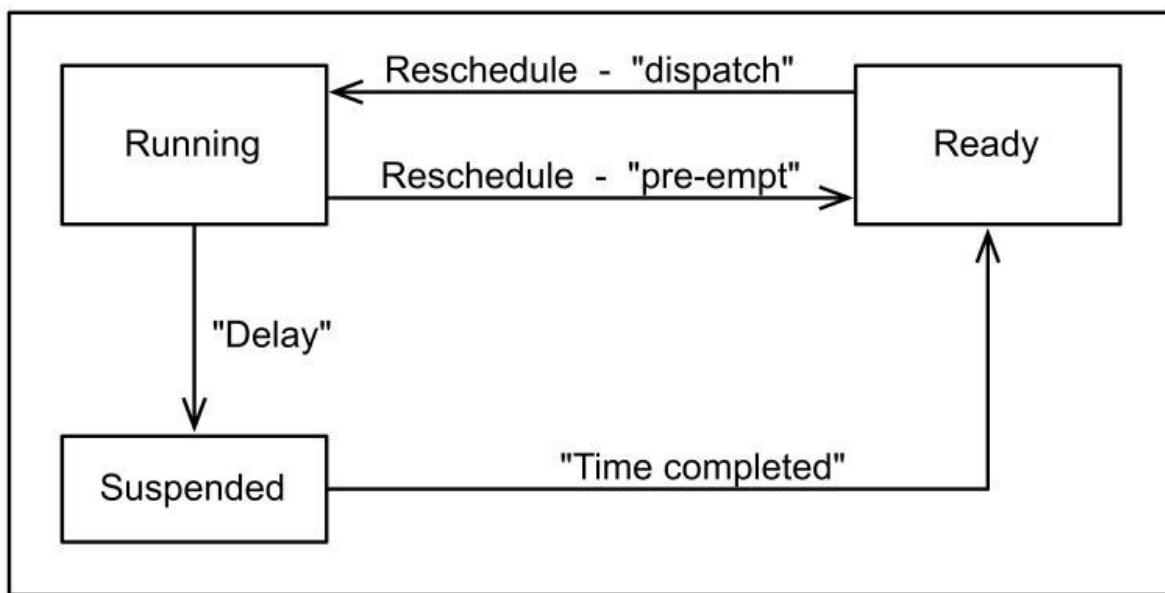


Figure 2.9 Task dynamics - basic state model for priority-based scheduling

A task can only enter the running state when it is dispatched by the RTOS. Observe that it must be in the ready state for this to occur; moreover, in a priority-based system it must be the ready head task. By contrast a task may leave the running state in one of two ways, suspension or pre-emption.

Suspension takes place when a task has either finished executing or else, for some reason, cannot continue (in this example it has to wait for a specified time delay). When a task goes into the suspended state it 'releases' the processor for use by other tasks (figure 2.10). Release is either self-induced or else forced by the RTOS. Self-release can be initiated by the running task itself, in two ways. First, it may have completed the required operation and so gives up the processor. Second, it may relinquish control as a result of signals generated

within its own program (internal events). Otherwise it can be forced to give up the processor for the reasons shown in figure 2.10.

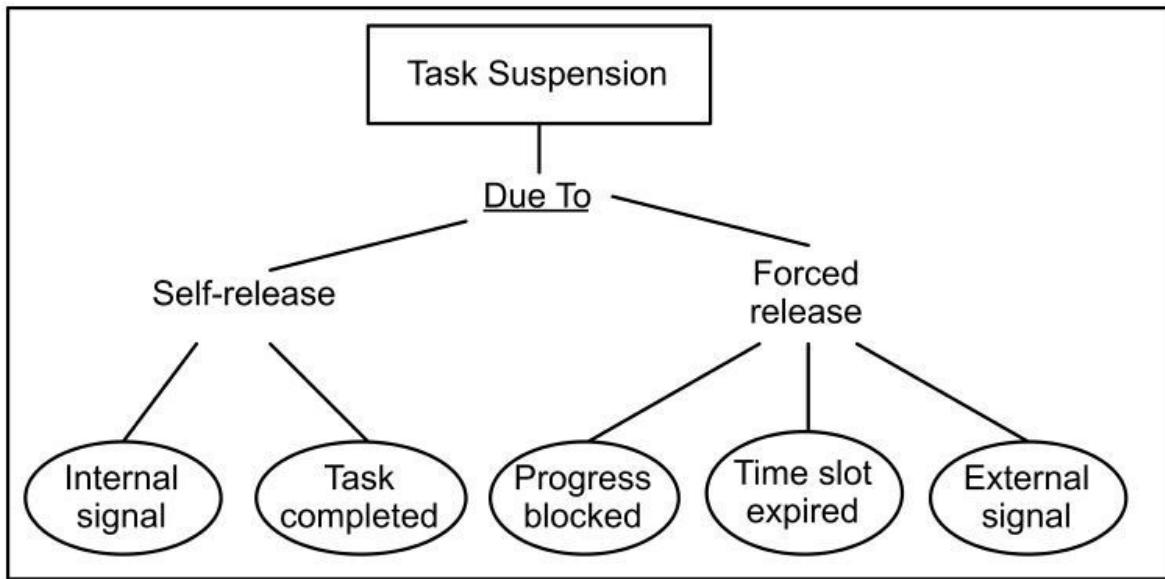


Figure 2.10 Reasons for task suspension

With pre-emption a running task is forced to give up the processor, even if it hasn't completed its work. Note well that in this case the task does not get suspended. Instead it returns to the ready queue, its position being determined by its priority. It remains in the ready queue until it is once again dispatched.

The more general state model of task dynamics is shown in figure 2.11, which is self-explanatory. It can be seen that tasks are readied from the suspended state when particular conditions occur. These include the completion of some event, elapse of a specific time interval, or some combination of event and time. Of course, not all suspended states apply to all tasks.

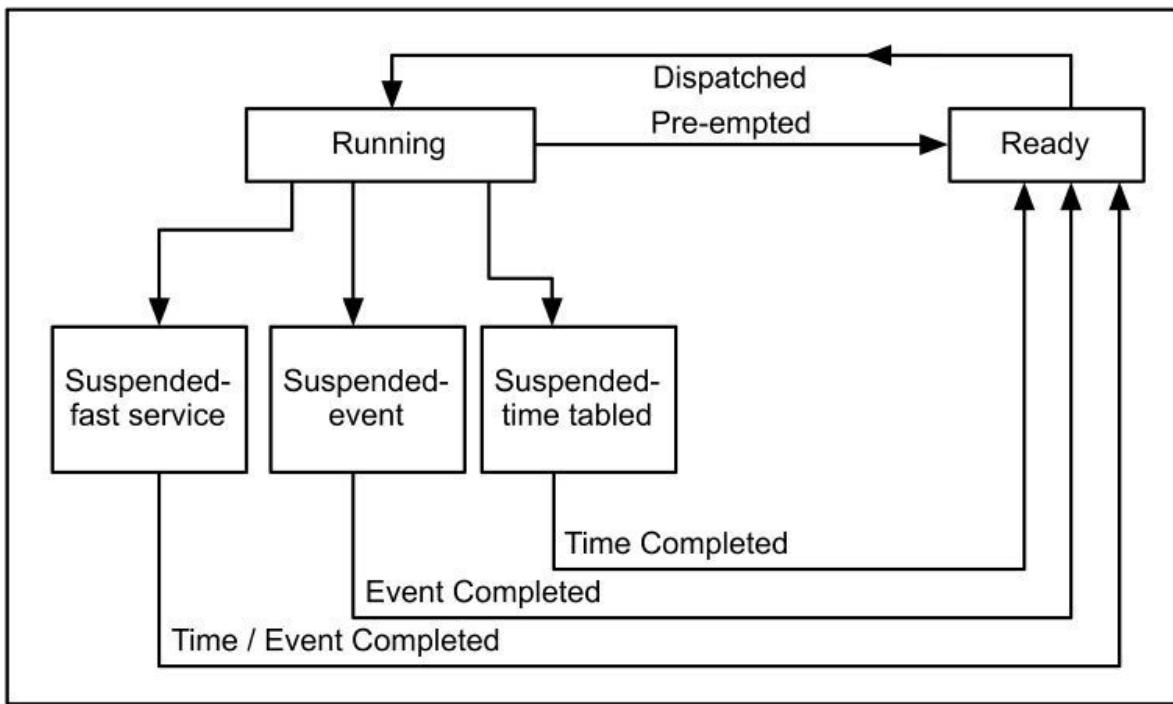


Figure 2.11 Task states - general model

The state model can, if desired, be extended to show the creation and deletion of tasks.

In priority-based systems the readying and rescheduling operations can become complex. A simple instance is given in figure 2.12, showing how priorities determine task positions in both ready and suspended queues. It also shows how the ready queue changes as new tasks are readied.

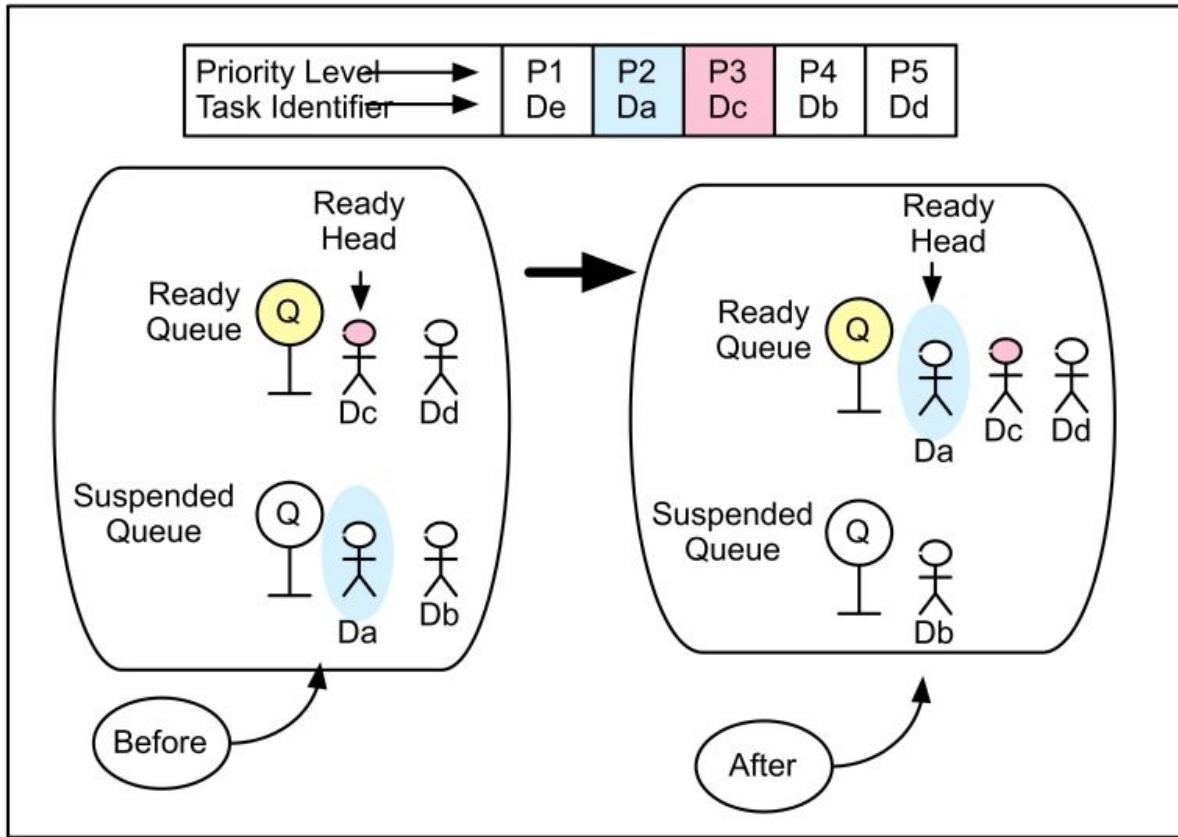


Figure 2.12 Effect of priorities on queue states - task awakening

2.7 Implementing queues - the task control block

The basic item used for constructing queues is a task description unit or 'task control block' (TCB). Information held in the TCB enables the executive to control all scheduling activities. Note, however, that TCBs carry task status and control information; they rarely hold the program (task) code itself.

There isn't a unique design of TCB. Nevertheless, certain features have to be included, figure 2.13.

1	Task Identifier
2	Status
3	Priority
4	NEXT TASK

Figure 2.13 Task control block structure

Within the TCB are a number of elements or 'fields'. These are used as follows:

- 1 - Identifies the task.
- 2 - Shows whether the task is ready to run or is suspended.
- 3 - Defines the priority of the task within the system.
- 4 - Gives the identifier of the task which follows (this applies to ready queues, suspended queues, and any other queues which may be used in the system). This field is used only when tasks queues are organised using linked list constructs.

Queues ('lists') are formed by linking individual TCBs together using pointers.

For instance, the ready list (figure 2.14) consists of task X followed by Y, then A, finally terminating in the Idle task.

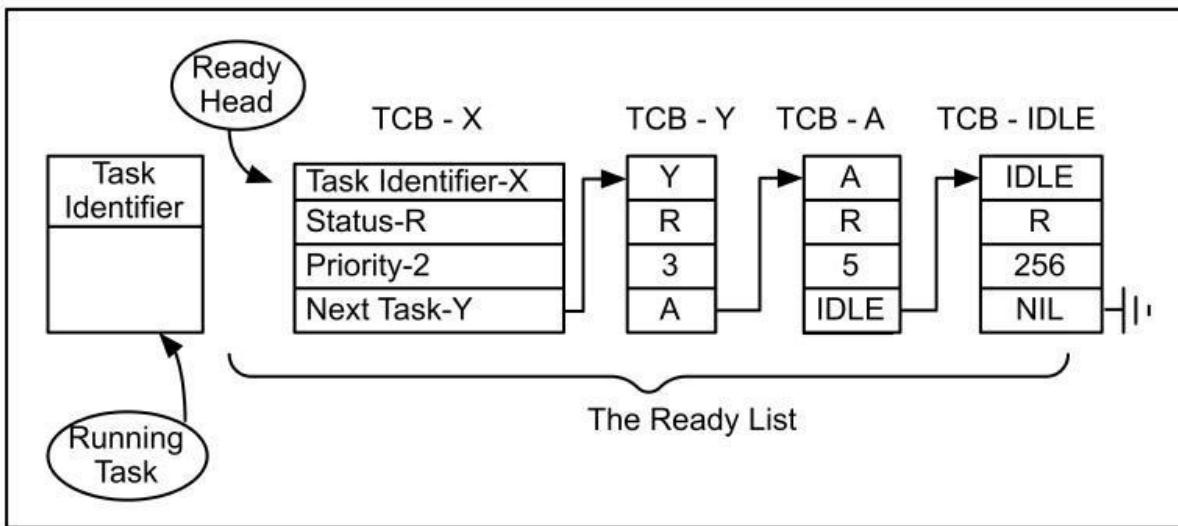


Figure 2.14 Ready list organisation

The Idle task, usually needed in embedded work, shows that it is the end point by pointing to 'nil'.

Suspended lists can be formed in much the same way, figure 2.15

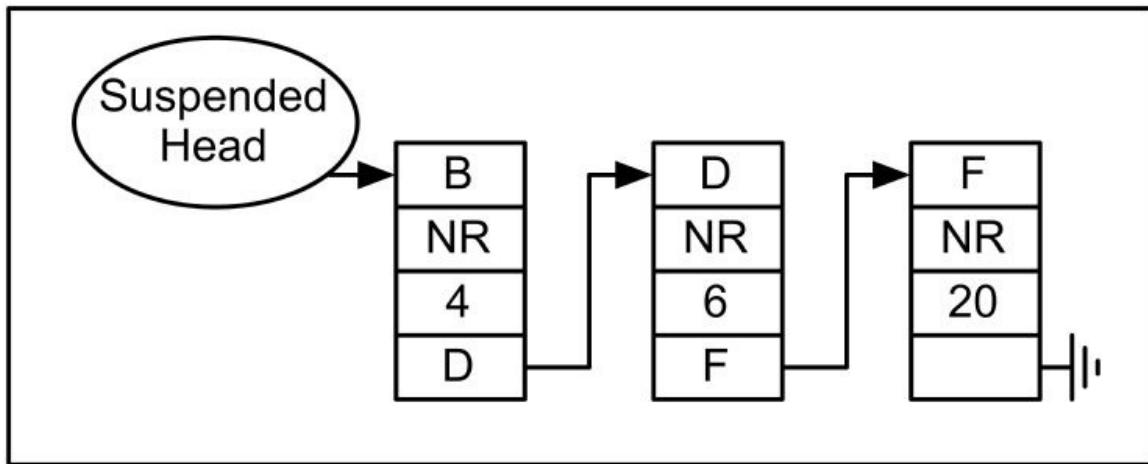


Figure 2.15 Suspended list organisation

The operating system may use one or more suspended list constructs; this depends entirely on individual designs. Task order within individual lists can be changed using one simple mechanism: the pointer construct. Reordering of tasks,

moving tasks between lists, adding new tasks to the system - all can be achieved merely by altering pointer values.

The task identifier field may be used to fulfil an extra role: a pointer to the so-called 'process descriptor' (PD).

2.8 The process descriptor

Earlier the point was made that each task thinks that it has sole use of the processor; it doesn't know that time sharing is taking place. It was also noted that the key to this is the saving and restoring of task information when context switches take place. And central to this is the process descriptor (PD).

Dynamic information concerning the state of the process (task) is held in the PD, figure 2.16. Each task has its own private PD and, in some designs, the descriptors may be located within the TCB itself.

One important point concerning the TCB and the PD is that both contain dynamic information. Thus they have to be located in read/write memory, usually RAM.

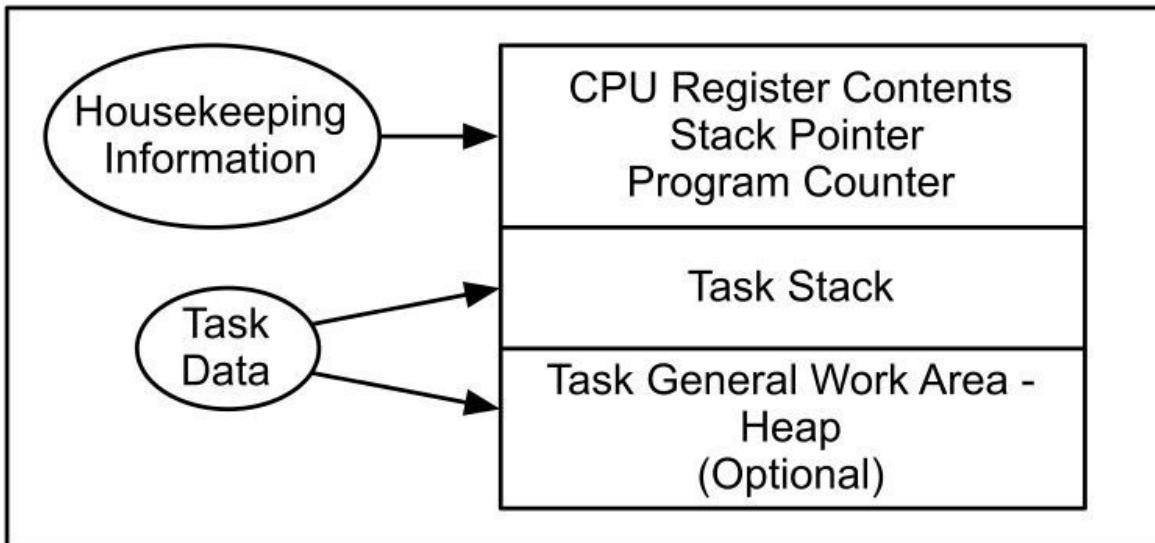


Figure 2.16 Process descriptor structure

We are now in a position to look at a practical example of task creation, based on the embOS RTOS, listing 2.1.

```

/* Example code to illustrate the creation of two tasks */

OS_TASK      PanControl;
OS_TASK      TiltControl;

OS_STACKPTR  PanControlStack[50];
OS_STACKPTR  tiltControlStack[50];

unsigned char      PanCtrlPriority    = 1;
unsigned char      TiltCtrlPriority   = 1;
unsigned char      PanCtrlTimeSlice  = 2;
unsigned char      TiltCtrlTimeSlice = 1;

/* This is the code for the pan control loop task */
void PanCtrlFunction(void)
{
    while(1)
    {
        /* Code of the pan control loop */
    }
} /* end PanCtrlFunction */

/* This is the code for the tilt control loop task */
void TiltCtrlFunction(void)
{
    while(1)
    {
        /* Code of the tilt control loop */
    }
} /* end TiltCtrlFunction */

void main(void)
{
    while (1)
    {
        OS_CreateTask(&PanControl, NULL, PanCtrlPriority, PanCtrlFunction, PanControlStack,
                      sizeof(PanControlStack), PanCtrlTimeSlice);

        OS_CreateTask(&TiltControl, NULL, TiltCtrlPriority, TiltCtrlFunction, TiltControlStack,
                      sizeof(TiltControlStack), TiltCtrlTimeSlice);

    }
} /* end main */

```

Listing 2.1

The code is generally self-explanatory, all key aspects being clear. Please note this is not meant to be an embOS tutorial; rather it is intended to provide a link between theory and practice. Moreover it is intended to show that code-level work is really quite straightforward.

2.9 The tick

The tick is an elapsed time counter, updated by interrupts from the system real-time clock (figure 2.17).

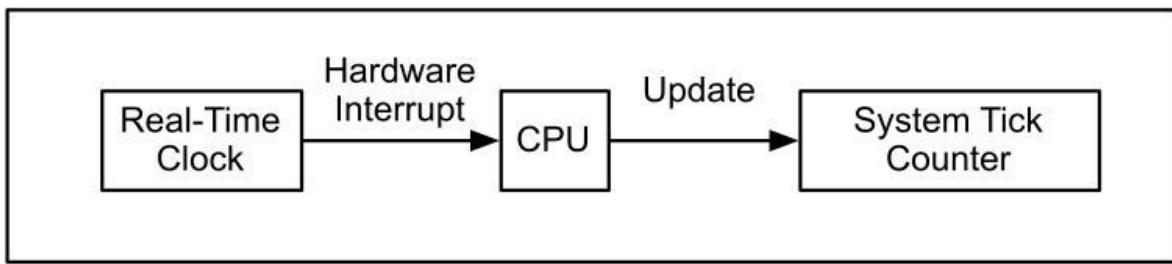


Figure 2.17 Updating the tick

It is often implemented as a time-of-day (TOD) counter, supporting four major functions:

- Scheduling timing.
- Polling control (related to scheduling).
- Time delay generation (related to scheduling).
- Calendar time recording.

(a) Scheduling timing

Here the real-time clock sets the time slot for scheduling. Each time the tick counter generates a signal it invokes an interrupt service routine that calls a reschedule. This would normally be used to underpin pre-emptive round-robin and periodic task scheduling algorithms.

(b) Polling control

Where tasks are event-driven, how is the processor made to respond to the event? One common solution, especially for time-critical events, is to use interrupts. But for many applications, interrupt-driven solutions aren't right; polling for status information is preferred. This solution could, for instance, be used with keypad signalling by employing a keyboard-scanning task. Here the keyboard status is scanned ('polled') at regular intervals as set by the tick. When a key-press is detected the scanning task readies the appropriate keyboard

handler task; otherwise the handler task remains suspended. With this approach the executive never loses control of the system, which may happen when using interrupts. It is sometimes called a 'deferred server' technique.

(c) Time delay generation

This feature is needed in most real-time systems, especially for process control work. Moreover, many applications use multiple time delays, with significant variations in the timing requirements. For instance, a combustion controller may wait for 250 milliseconds while checking flame conditions after generating an ignition command. On the other hand, a temperature controller may wait for one hour between turning a heater on and running the control loop. These diverse demands can be met, fairly painlessly, by using the tick.

(d) Calendar time recording

In specific instances system activation, control and status recording must be tied to the normal calendar clock. A count of hours, minutes and seconds - a 24-hour clock - can be implemented using the tick counter. The tick could also be used for months, days, etc. But remember, when processor power is switched off, everything stops (including the tick). Therefore, for embedded applications, it is better to use special battery-backed time-of-day (TOD) clocks for long-term timing.

2.10 Priorities and system responsiveness

How quickly does a task get serviced in a multitasking environment? Unfortunately there isn't a simple answer - it's interlinked with task priorities (figure 2.18).

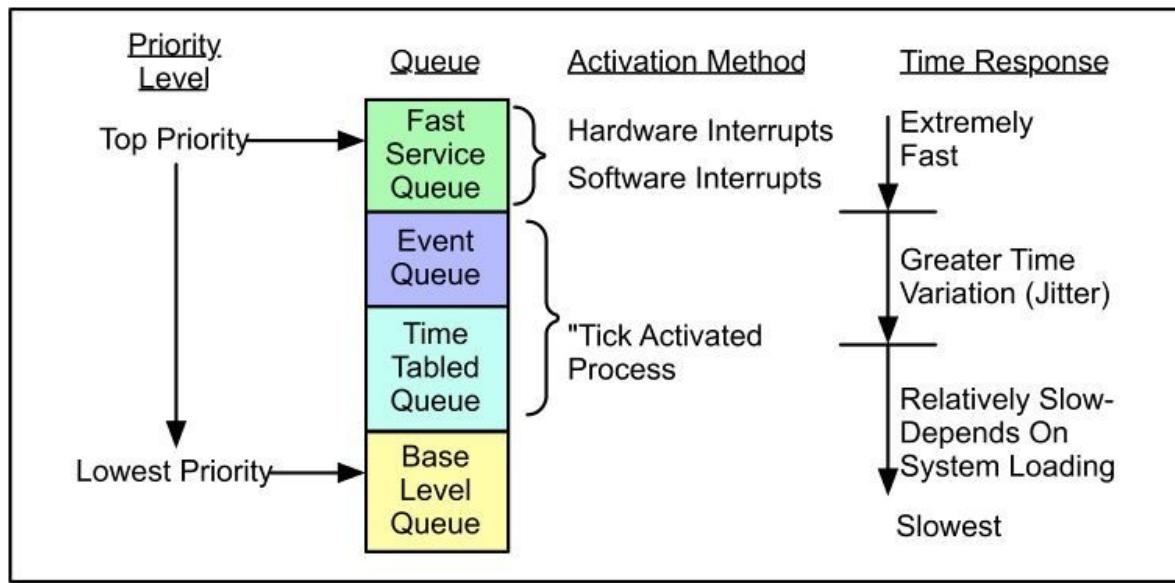


Figure 2.18 System responsiveness

High priority tasks of the 'fast service queue' type are readied only when service is required, using interrupt signalling. Usually the reaction time ('interrupt latency') is small, typically in the range 1-100 microseconds. This assumes, though, that the task is allowed to run; after all, a higher priority one may be executing at the time of interrupt.

When task execution is tied in with the tick, some time variation or 'jitter' is experienced. This is small for the higher-priority jobs, getting larger as priorities reduce. Only tasks that can tolerate quite slow responses are implemented at the lowest (base) level.

From what has been said, the tick period has a significant effect on system responsiveness. The issue isn't clear-cut, however, being interlinked with scheduling strategies. For real-time working it is a primary design requirement that all tasks be completed within an allotted time period. So CPU performance is the limiting factor here. In cases like this there seems little point in pre-empting important tasks, i.e. a task, once activated, should run to completion. This is a reasonable approach provided the current task has highest priority, any

waiting (ready) tasks being of lower priority. But now suppose that a task is readied having a higher priority than the running one. In this instance task pre-emption takes place. The running task is returned to the head-of-queue position, its place being taken by the new task. To support this strategy each task is given a specific priority; all tasks above the base level must have individual, different, priority setting (but see comment below on sub-policies).

Assume that a currently executing task reaches completion. It then stops, indicating a 'not-ready' status; only at the next 'tick' does the executive regain control of the system. To minimise this 'wasted' time it makes sense to have a short tick period. Yet this may cause problems. The tick, remember, is updated on an interrupt from the real-time clock. This results in the current task being replaced by the tick handler (a 'context switch'). All current information must be saved before this handler is loaded up and set going. The time taken for this is nothing but an overhead, thus reducing processor utilization. So, as the tick period is reduced, the available task processing time also falls. Ultimately, for very high rates, tasks would never be executed; the processor would spend its time just servicing the tick (a ridiculous situation, of course).

Choosing the tick timing rate of the system isn't easy. It is determined both by response requirements and scheduling techniques. For fast real-time applications it usually lies in the range 1-100 milliseconds.

Within this framework it can make practical sense to run a number of tasks at the same priority level. The reason for doing this is to get a good balance between responsiveness, throughput and functional behaviour. To make this work a mix of scheduling policies must be used. Usually this consists of a primary policy augmented with one or more sub-policies. In the majority of RTOSs the primary policy is the priority pre-emptive scheme. However, at any particular priority level tasks may be run using other schemes, typically cooperative or round-robin scheduling (details are RTOS-specific). So, across the task set the normal priority pre-emption rules apply; within the individual priority level the rules of the sub-policy are enforced.

With basic tick-driven tasking a reschedule takes place only at tick time (the tick ISR calls an RTOS-provided 'reschedule' function). This is nice and simple to implement: nothing complicated. Unfortunately it is likely to reduce processor utilization, especially where time slices are long. The reason is that if a task stops executing between ticks (e.g. it completes its work or is forced to stop), then nothing happens until the next reschedule point. Thus most RTOSs provide mechanisms to call a reschedule in such circumstances (independent of the tick).

2.11 By-passing the scheduler

In most circumstances the scheduler handles all tasking operations. For instance, when an interrupt occurs, the interrupt handler only readies the appropriate task; it doesn't dispatch it - that is the responsibility of the scheduler. As mentioned earlier, the task joins the ready queue at a position determined by its priority. Unfortunately, in some cases this could result in a long delay between the interrupt requesting service and actually getting it. And there are situations where this just cannot be tolerated, e.g. system exceptions.

To cope with such issues we can employ special interrupt service routines that completely by-pass the operating system. In effect they have a higher priority than that of the tick. What the ISRs do depends on system applications, design requirements and the exceptions that invoked them in the first place. But be careful with these special ISRs; use them with great care and thought.

2.12 Code sharing and re-entrancy

The most widely-used building blocks of modern programs are the subprogram (for high-level languages) and the subroutine (assembly language programming). These are invoked as and when required by running tasks. Now, in a multitasking system, each application process (task) is written separately and the object code loaded into separate sections of memory. As a result tasks may appear to be independent, but this isn't true. They are interlinked via their common program building blocks, the subprograms or subroutines.

Consider the following scenario. A general procedure (e.g. one that is available for use by all tasks) uses a specified set of RAM locations. It is invoked by task 1, which begins to manipulate data in the RAM locations. Task 1 is pre-empted by task 2, which then also calls the same procedure and operates on the same data locations. Some time later task 1 resumes, being totally unaware that the RAM data has been altered - and chaos ensues. To avoid this problem each task (process) is allocated its own private stack and workspace. All subprogram parameters are normally kept on the stack, all local variables residing in the workspace. Now shared code can be used with safety because each task keeps its own data to itself. Such code is said to be 're-entrant'.

2.13 The unpredictability of run-time behaviour.

Let's round off this chapter by looking at a really very important aspect of task-based systems: the run-time behaviour of the software. We'll do this using a simple example (table 2.1) that consists of three tasks having the following features:

TASK	PRIORITY	EXECUTION TIME	PERIOD
A	1	10 milliseconds	80 milliseconds
B	2	45 milliseconds	None – aperiodic
C	3	20 milliseconds	None – aperiodic

Table 2.1

Scheduling is done using priority pre-emptive methods, with a time slice of 10 milliseconds. All three tasks are readied at time $t = 0$. Switching takes place only at the end of a time slice.

1. What is the resulting system run-time behaviour over the first 100 milliseconds of operation?
2. Repeat this if the period of task A is changed to 40 milliseconds.
3. Repeat (2) when task B is re-readied at $t = 85$ milliseconds. Show what happens for the first 160 milliseconds of operation.

The answers to this are shown in figure 2.19.

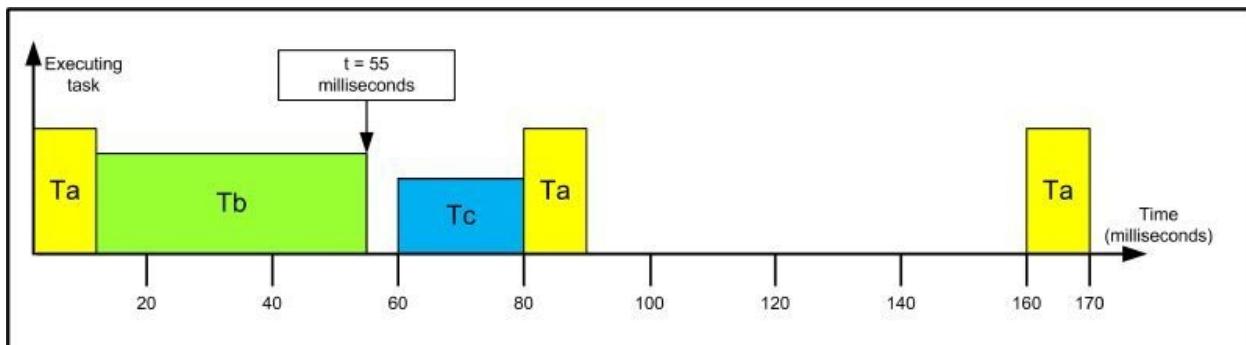


Figure 2.19(a) Unpredictability of run-time behaviour - 1

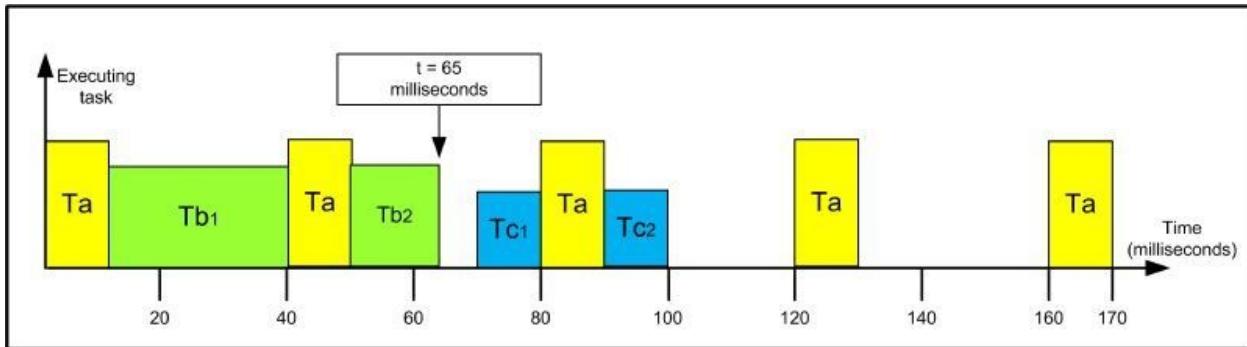


Figure 2.19(b) Unpredictability of run-time behaviour - 2

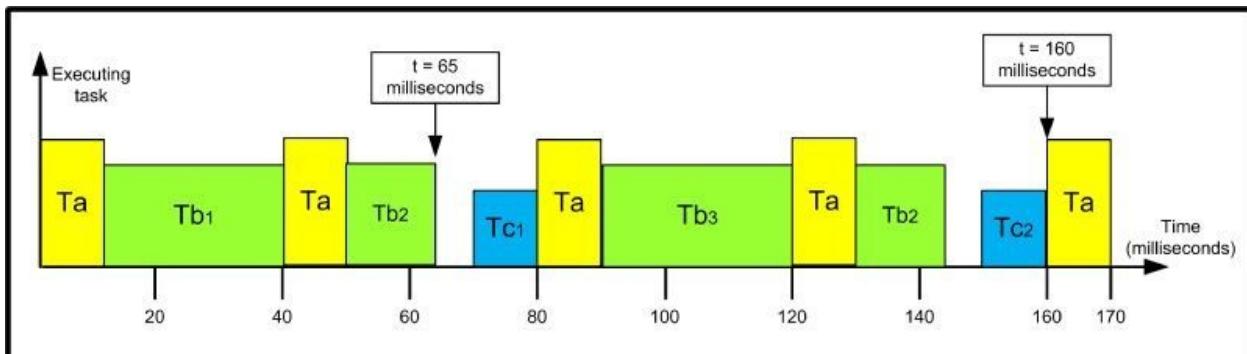


Figure 2.19(c) Unpredictability of run-time behaviour - 3

Please study this carefully to make sure you understand why this run-time behaviour has come about. This shouldn't be too challenging; but the more important point is 'what can we deduce from this?'. This example does, in fact, illustrate nine important factors:

- Priority pre-emptive scheduling can result in complex run-time behaviour.
- Timing predictions are difficult to make. When aperiodic tasks are present it may well be impossible to do this.
- Actual run times cannot be deduced directly from the execution time of individual tasks.
- Task starvation is a practical issue (in RTESSs, this means that tasks are late in producing results: hence poor reactive performance).
- Behaviour is much easier to predict when using periodic tasks only (and the results are likely to be accurate).
- The use of aperiodic tasks should be minimized whenever possible OR their impact should be minimized (using, for example deferred servers).
- To achieve the response requirements of fast reactive systems processor

utilization needs to be restricted.

- To achieve the response requirements of fast reactive systems the number of tasks should be minimized.
- A 'pencil and paper' approach to modelling the behaviour of real designs is likely to be difficult and time consuming. Automated tools are required.

Review

You should now:

- Understand the basis of simple cyclic, timed cyclic, round-robin and priority pre-emptive scheduling algorithms.
- Appreciate the merits and drawbacks of the various techniques.
- Know what the tick is and what it does.
- Realize why it is important to set sensible tick times.
- Know the reason for and use of ready and suspended queues.
- Know how queue ordering is changed at reschedule time in both priority and non-priority scheduling schemes.
- Know what the ready head is.
- Be able to describe task behaviour using a state transition diagram.
- Know what the roles of the task control block and the process descriptor are.
- Appreciate how priorities, time responses and task activation methods are interlinked.
- Understand why special interrupt handling routines may by-pass the scheduler in critical situations.
- Recognize why code re-entrancy is essential in multitasking systems.
- Understand why it may be extremely difficult or even impossible to predict the overall run-time behaviour of the task set.

Now is the time to start on the practical exercises, specifically the part 1 set (see 'The practical exercises - introduction' in book 2). This will reinforce your understanding of the material covered in these last two chapter of this book. It should also give you a greater insight into the actual run-time behaviour of software in a multitasking environment. And it should really highlight one key aspect of multitasking designs: just because tasks are independent entities doesn't mean that their temporal performance is also independent of each other.

Chapter 3 Control of shared resources - mutual exclusion.

The objectives of this chapter are to:

- Explain the problems met when using shared resources in multitasking designs.
- Describe what mutual exclusion is.
- Show how mutual exclusion can be implemented using program flags.
- Explain the concepts and use of binary and counting semaphores.
- Describe what a mutex is and show how it improves on the semaphore.
- Highlight the weaknesses of both the semaphore and the mutex.
- Show how these weaknesses can be overcome by using a simple monitor construct.

3.1 The problem of using shared resources

In a single CPU system the processor is a shared resource. Now there aren't contention problems when sharing it between the various processes; everything is controlled by the scheduler. But this isn't true for the rest of the system. Different tasks may well want to use the same hardware or store area - simultaneously. Without controlling access to these common resources, contention problems soon arise. Consider, for instance, what can happen in the following situation. Here a control algorithm is executed at regular intervals, interrupt-driven by a timer process (figure 3.1).

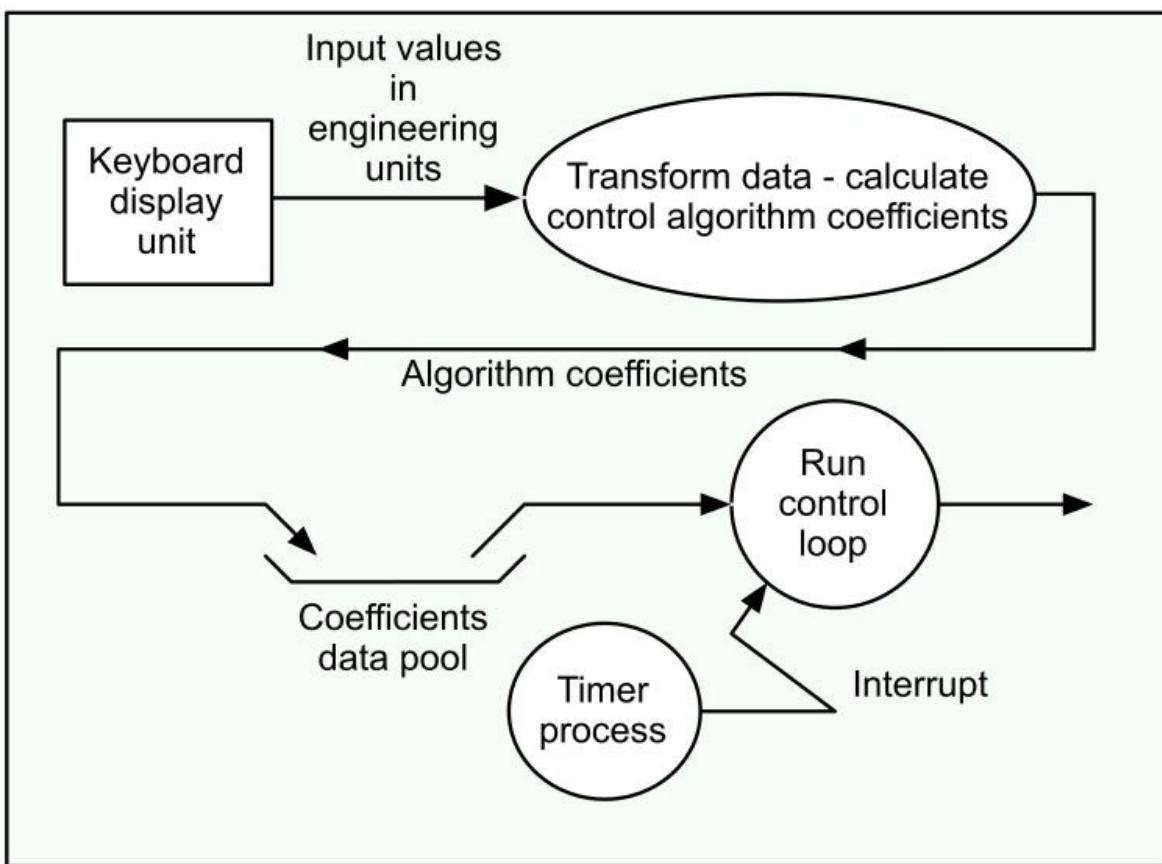


Figure 3.1 Shared data between processes

Part of the data used comes from the coefficient's shared read-write data store (data pool). The coefficient values are derived from engineering units input from a keyboard-display unit. Now, the problem here is a simple one. What would be the result if the control loop was activated while the coefficients were being updated? Up to eight bytes may be used for each coefficient; yet it might be that

only one or two can be changed at a time. Therefore a value might only be part-updated when a task switch takes place. If this happened the results could well be disastrous.

How can we tackle this problem? In very general terms the solution is clear. Make sure that a shared resource can be accessed by one, and only one, process at any one time. That is, implement a 'mutual exclusion' strategy. The difficulties come, however, when we try to institute specific methods.

3.2 Mutual exclusion using a single flag

Imagine that, to control access to a shared (or 'common') item, we put it in a special room, figure 3.2.

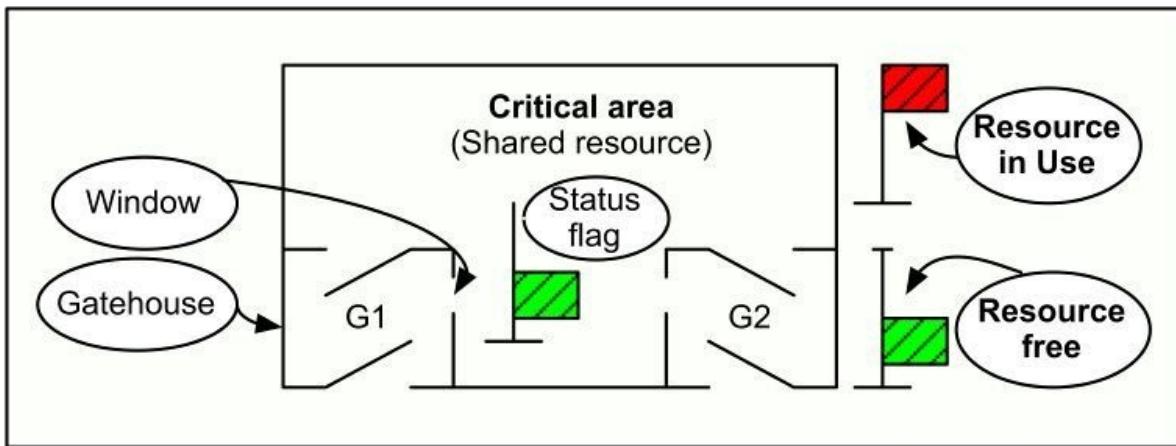


Figure 3.2 The single flag approach

A gatehouse is provided for each task wishing to use the resource; this is its means of access. An indicator flag is used to show whether or not a task is in the room (in the 'critical area'). The flag can only be seen, raised or lowered from within a gatehouse (any gatehouse, in fact).

Assume initially that the critical area is empty, the flag being down. User 1, who wishes to use the resource, enters his gatehouse. He first checks the flag position. Finding the flag down (indicating the resource is free) he then raises it and enters the critical area. At this point user 2 arrives on the scene, also wanting access to the shared item (figure 3.3). He enters his gatehouse and checks the flag. As it is up (resource in use), he waits here, constantly rechecking the flag status. Eventually user 1 leaves, his last job being to lower the flag, indicating 'resource free'. So when user 2 next checks the resource status he finds it free. Consequently he raises the flag and enters the critical area, now being its sole owner.

This all looks pretty good. Mutual exclusion has been successfully achieved using a fairly simple mechanism. Or has it? Consider the following scenario. The resource is free when user 1 enters gatehouse 1. He checks the flag. Finding it down, he turns to raise it. Just at this moment user 2 enters his gatehouse and checks the flag; He also finds it down. So, as far as he is concerned, his way is clear to enter the critical area. What he doesn't realise is that user 1 is also doing

the same thing. Clash!

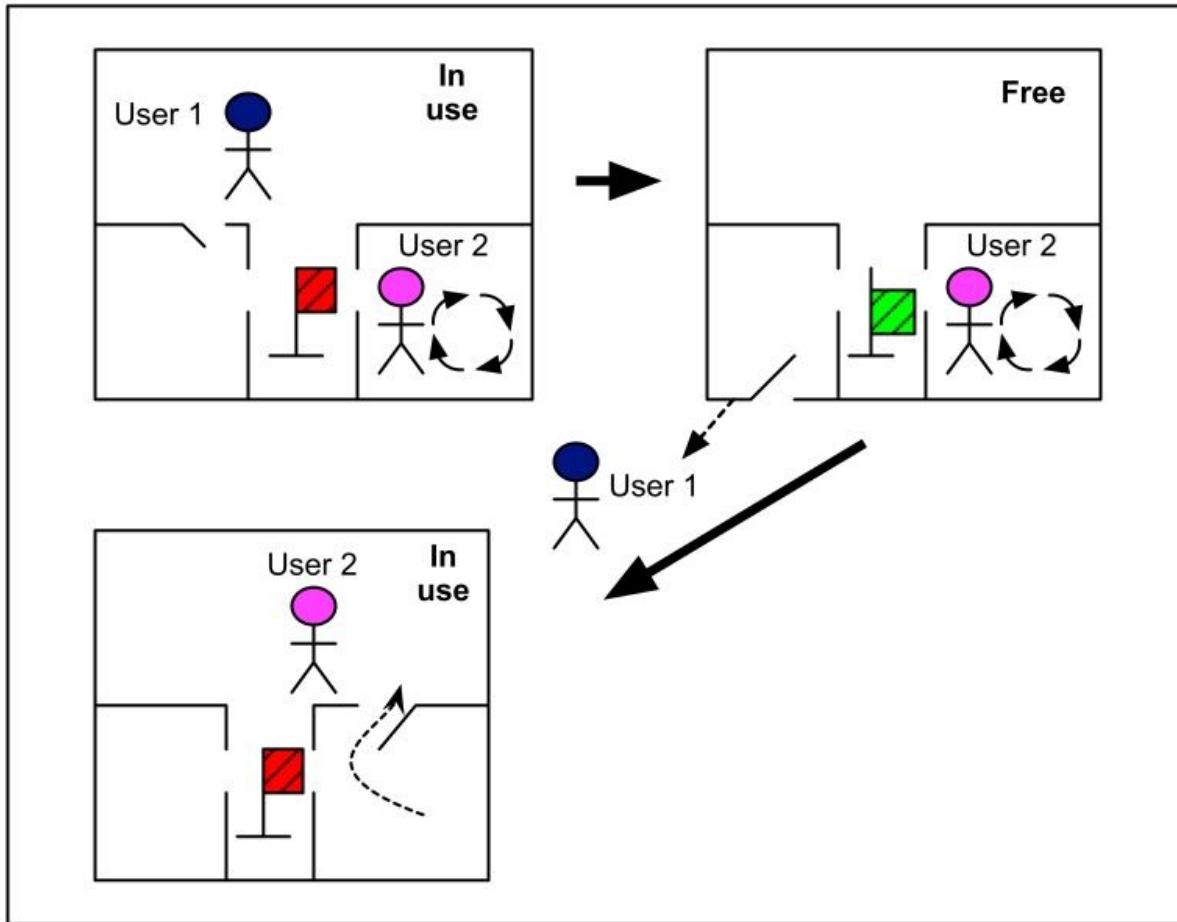


Figure 3.3 Mutual exclusion in action (single flag method)

Here protection breaks down because there is a time lapse between user 1 checking the flag and then changing its status. In computing terms, for a single processor system, this is equivalent to:

1. Load a status variable from memory into a processor register.
 2. Check its status.
 3. If the variable says 'free' then
Change its value to 'in use' and
Write this to memory
- else recheck.

During this sequence, user 1 (i.e. task 1) could very well be pre-empted. If that

happens after loading up the variable but before changing its status, then we may have problems. Suppose that the pre-empting task is user 2. It checks the flag status, finds the resource free, and enters the critical area. When task 1 resumes, it also finds that the resource is free, enters the critical area - and so the mutual exclusion mechanism has failed.

Fortunately, most modern micros are able to perform a bit/byte 'set-and-test' in a single instruction. This means that checking and setting is an indivisible or atomic operation, so security is assured. However this may not be the case in multiprocessor systems as these have true concurrency. This will be considered later.

The single flag technique is simple to implement, easy to use and, with proper design, safe. Unfortunately it isn't very efficient. As pointed out above, when a task finds the flag set against it goes into a checking loop. But this task can't change the flag status; thus it will remain in a 'busy-wait' mode for the duration of its time slice. The result is wasted processor time and reduced processor performance (i.e. lowered utilization). Such inefficiency is not acceptable in fast systems (especially hard-fast); an alternative mutual exclusion technique is needed.

It is clear that the task needs to give up the processor when it finds its way blocked. That is, it should suspend, thus allowing another task to instantly use the processor. Such an approach is called 'suspend-wait', and can be implemented in a number of ways. It could be done in some cases under program control using cooperative signalling. However a much better technique is to employ constructs specifically designed to support suspend-wait operations: semaphore, mutex and monitor.

A small point of terminology; a flag, when used in this way in multicore/multiprocessor systems, is also called a spinlock.

3.3 *The semaphore*

3.3.1 The binary semaphore

Essentially a semaphore is a program data item used to decide whether tasks can proceed or should suspend. There are two semaphore types, the 'binary' and the 'general' or 'counting' semaphore. Both work on the same principles, these being originally developed by Edsger Dijkstra, in 1965.

Let's deal first with the binary semaphore. This, fundamentally, is a task flow control mechanism that can be likened to a railway signal, figure 3.4. Trains, depending on the signal position, can either pass this point or else must stop. If they stop they remain at this point until the signal changes to 'go'.



Figure 3.4 The semaphore analogy - railway signals

In a similar way a semaphore either lets a task proceed (that is, continue to execute its code) or else suspends it. Once suspended, a task will remain in this state until it is readied by some program action.

In railway systems signals are safety mechanisms, used to prevent collisions, damage or death by controlling train movement. Practical rail networks have many signals, these being employed as needed. Likewise, any real multitasking design is likely to use a number of semaphores, each one being equivalent to a specific signal.

Semaphores are used for two very distinct purposes in concurrent software. As described here it functions as a mutual exclusion (contention elimination) device,

one semaphore being allocated to each shared resource. It is also used to synchronize task interactions, described later.

The concept of the semaphore when used for access control is shown in figure 3.5. The analogy is that of car park entry control, semaphores being equivalent to the controlling mechanism.

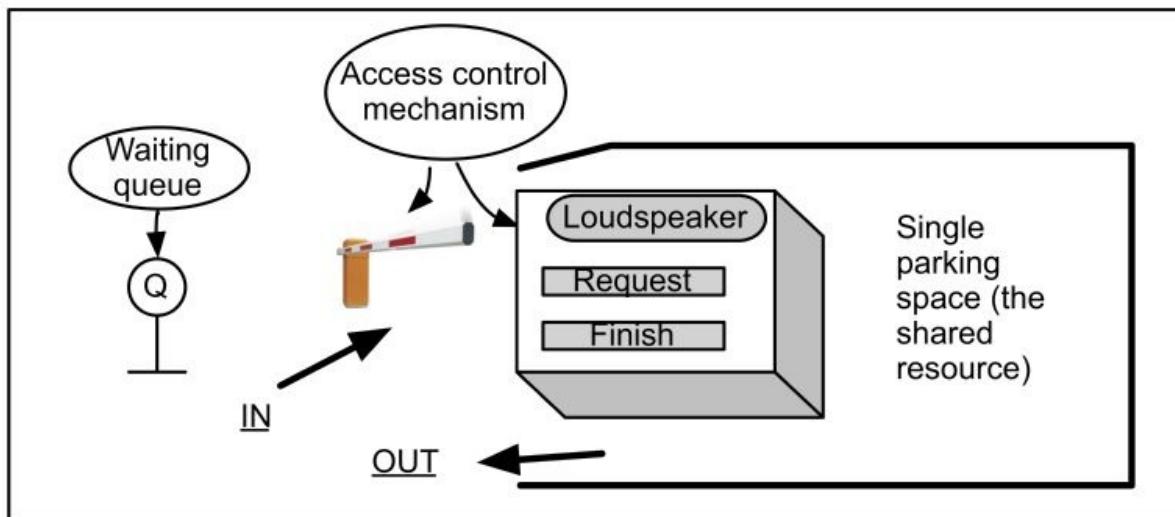


Figure 3.5 The binary semaphore and mutual exclusion - concept

Here the shared resource is a single parking space for a drop-off pick-up zone. What we want to do is to make sure that one car, and only one car, can enter the parking area. Hence users are told that before trying to park they must first check

that the space is free. To do this they use an access control interface which consists of a:

- 'Request' pushbutton. A user presses this to signal the car park attendant that entry is required to the car park.
- 'Finish' pushbutton. The user presses this on exit to inform the attendant that the car park is once again free.
- Loudspeaker. The car park attendant uses this to answer the customer's request.

In this analogy a binary semaphore is the software equivalent of this access control mechanism; and for 'user' read 'task'.

Assume initially that the car park - the resource - is free. The first operation is to provide the attendant with this status information (assume that the parking space cannot be seen from the control room). Its software counterpart is the initialization of the semaphore. Similarly, the 'attendant' functions are provided as part of the OS software.

When a user wants to use the resource he approaches the barrier and presses the Request button. In semaphore terminology this is defined to be a 'wait on the semaphore', the Wait operation. As the resource is free the attendant raises the barrier and answers 'pass'; the user then enters the protected area. Now the barrier is lowered.

At some point the user will leave and so vacate the parking space. On exit he signals this to the access control mechanism by pressing the 'finish' button. This is defined to be the Signal operation. The result is to update the attendant's status information to show that the car park is once more vacant.

Now examine the case where the resource is being used when another car arrives. In response to Wait the requesting user is directed to the waiting queue (this corresponds to task suspension). The current occupier, once finished, leaves the parking space, generating a Signal as he goes. As a result the control mechanism records that the resource is free. But now, subsequent events follow a different pattern; there isn't any updating of the attendant's status information. Instead the barrier is raised and a 'pass' message is issued to the waiting user (equivalent to one task waking up another one). This authorizes the user to enter the protected area; from this point on events proceed as described earlier.

One important point to consider is what happens when a higher-priority task (say task 3) arrives when a task (1) is already waiting in the queue. The outcome, in fact, depends on the queuing policy being used. In general two basic methods are

available, FIFO or priority pre-emption.

With FIFO queuing, task 3 lines up behind task 1. Thus task 1 can proceed as soon as conditions permit it (i.e. the space becomes available). Now while this is perfectly safe it has resulted in a lower-priority task delaying execution of a higher priority one. This could lead to significant problems (see later, priority inversion).

When priority pre-emption is used task 3 takes precedence and goes to the front of the queue. Thus it will be the first to be readied. But this has the effect of further delaying the execution of task 1, also leading to a potential performance problem (task starvation).

It is up to the individual designer to decide which method to use and where to use it. But in either case task behaviour can be modelled as shown in figure 3.6.

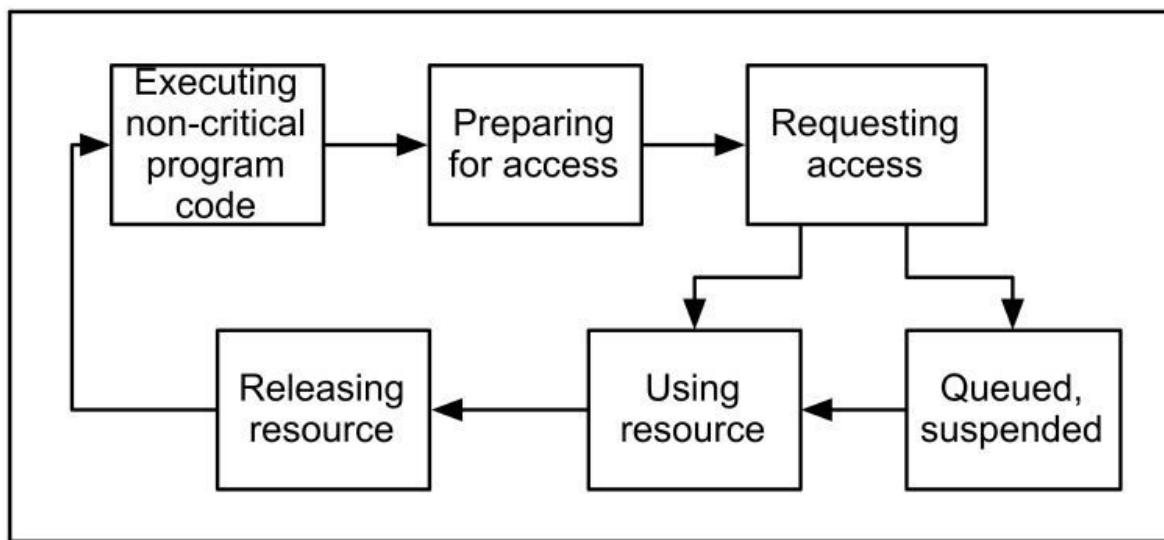


Figure 3.6 Task states when using semaphores

As stated earlier, an individual semaphore is created for each individual controlled resource. In programming terms this (the semaphore) is seen as nothing more than a named data item.

Let's look at a simple application, that of protecting the data pool 'Coefficients data' of figure 3.2. First we create a semaphore named CoefficientsSemaphore, the operations applicable to this being:

- Wait (CoefficientsSemaphore)
- Signal (CoefficientsSemaphore)

A binary semaphore has two values only, '0' or '1'. A '0' shows that the resource is in use, the '1' indicating that it is free. In its original form, the semaphore operations are (listing 3.1, 3.2):

```
/* Part of Procedure Wait on semaphore CoefficientsSemaphore */
if (CoefficientsSemaphore == 1)
{
    CoefficientsSemaphore = 0;
} /* end if */
else
{
    SuspendTask();
}
```

Listing 3.1

```
/* Part of Procedure Signal on semaphore CoefficientsSemaphore */

if (TaskWaiting)
{
    WakeupTask();
}
else
{
    CoefficientsSemaphore = 1;
} /* end if */
```

Listing 3.2

Within a program these would be used as and when needed (listing 3.3), as follows:

```
/* This is just an example fragment - mix of code and pseudocode*/
UnprotectedProgramStatements;
Wait(CoefficientsSemaphore);
UseSharedResource;
Signal(CoefficientsSemaphore);
UnprotectedProgramStatements;
```

Listing 3.3

Wait and Signal operations are defined to be 'primitive' types, that is, each one is indivisible. In other words, once the Wait or Signal process is started, the sequence of machine instructions cannot be interrupted. This is essential, otherwise we'll just end up with the problems of the single flag exclusion mechanism. Providing indivisibility isn't an easy task; it may pose real implementation difficulties. But they have to be overcome for the semaphore to work. Further, these operations must be protected by the operating system, not the programmer.

The binary semaphore may be implemented as a single byte or even a bit within a byte using a single 'set-and-test bit' instruction. This won't work, unfortunately, where the test and check involves a number of processor instructions. In this case we ensure the operations are atomic by disabling interrupts before carrying out semaphore actions. When we've finished with the resource interrupts are re-enabled.

Note that this technique won't usually work in multiprocessor systems. What is needed is a hardware lockout mechanism (see later chapter).

A historical point; Wait and Signal are also called P and V, being derived from the Dutch language. There is some disagreement about which words they actually refer to (prolaag and verhogen are favourites).

3.3.2 The general or counting semaphore

Suppose that our single shared resource is restructured so that it now consists of a number of items. Each item is identical, providing a specific service. For instance, it could be a set of local area network queues, used to store messages for onward transmission. Given this arrangement it is safe to let more than one task access the resource provided they don't use the same queue. To support this the semaphore construct is altered so that:

- It has a range of values (say 0 to 4); initially it is set to the maximum value (4).
- Each value corresponds to a specific instance of the provided resource, zero indicating all devices in use.
- When a user wants to access the store it first checks that the resource is

available (not a zero value), listing 3.4. If access is granted it decrements the semaphore value by 1 (one), and proceeds to use the resource facilities.

- When the user has finished, it increments the semaphore value by 1 and exits the store, listing 3.5.

```
/* Part of Procedure 'Wait on CAN queue' */

if (CanQueue >0)
{
    --CanQueue;
}
else
{
    SuspendTask();
} /* end if */
```

Listing 3.4

```
/* Part of Procedure 'Signal on CAN queue' */

if (TaskWaiting)
{
    WakeupTask();
}
else
{
    ++CanQueue;
} /* end if */
```

Listing 3.5

The value of CanQueue controls access to the resource and defines the item that can be used; it is never allowed to go negative.

The binary semaphore can be treated as a special case of the counting semaphore where the count value is one (1). This is demonstrated below (listing 3.6) in a practical example using the semaphore constructs of the ThreadX RTOS.

```

/*
Code example: RTOS -- ThreadX
    Wait equivalent:      tx_semaphore_get
    Signal equivalent:   tx_semaphore_put
    Semaphore data type: TX_SEMAPHORE
*/

/*
Create a counting semaphore with a count of 1.
Note that this must have file scope.
*/
TX_SEMAPHORE ADCsemaphore;
int           SemaphoreStatus;

/* initialization code typically in main */
SemaphoreStatus = tx_semaphore_create(&ADCsemaphore, "ADCsema", 1);

/*
Use the semaphore to control access to shared item.
This has function scope.
*/

***** Start protected code section *****/
    /* Wait on the semaphore */
    tx_semaphore_get(&ADCsemaphore);

    /* Use the protected resource */
    GetAnalogueInput(&RotorSpeed);

    /* Signal the semaphore */
    tx_semaphore_put(&ADCsemaphore);

***** Finish protected code section *****/

```

Listing 3.6

3.3.3 Semaphore limitations and problems

The semaphore has been widely used to enforce mutual exclusion policies. It is easy to understand, simple to use and straightforward to implement. Sadly, its limitations and related problems aren't always appreciated:

- A semaphore is not automatically associated with a specific protected item. Yet, in practice, it is essential to correctly pair these up.
- In the operations described so far there is no concept of 'seeing' the state of the semaphore. What the requester really does is to ask 'can I use the resource' (and if the answer is no, then the requesting task is automatically suspended).
- Wait and Signal operations form a pair. Regrettably the basic mechanism does not enforce this pairing. As a result a task could call either one in isolation, which would be accepted as valid source code. This may lead to very unusual run time behaviour.
- Nothing prevents Signal being called before Wait, again possibly a source of odd behaviour.
- A semaphore must be visible and in scope to all tasks that share the protected resource. This means that any task can 'release' the semaphore (by calling Signal), even if this is a programming error.
- Just because a resource has a semaphore associated with it does not guarantee protection. Safeguards can be bypassed if there is a 'back-door' route into the protected area (using a resource which is declared to be global to the program, for example).

There is also a further significant problem, more to do with its use rather than its construction. Most programs that use semaphores implement them as and when needed. Consequently they tend to be scattered around the code, often proving to be difficult to find. In a small design this is handleable, but the same can't be said of large ones. As a result the designer must keep track of all mutual exclusion activities, otherwise debugging can be 'challenging'. And, post design (in the maintenance phase), scattered semaphores can make life quite difficult indeed. Often the result of doing 'simple' program modifications is software that has very odd (and most unexpected) run-time behaviour.

3.4 The mutex

The mutex is very similar to a semaphore but is specifically intended to control access to shared resources, i.e. MUTual EXclusion (remember, the semaphore is essentially a flow control mechanism). To avoid confusion different names are used for the semaphore and mutex operations. Our choice is Lock (aka Wait) and Unlock (aka Signal).

A mutex differs from a semaphore in one critical way; the task that releases the mutex (Unlock) must be the one that locked it in the first place. Thus we can consider the locking task to own the mutex. A practical example of the use of the mutex is given in listing 3.7.

```
/*
Code example: RTOS standard -- Pthreads
Lock equivalent:      pthread_mutex_lock
Unlock equivalent:    pthread_mutex_unlock
Mutex data type:      pthread_mutex_t
*/

/* Create and initialize a mutex.
Note that this must have file scope.
*/
pthread_mutex_t ADCmutex;

/* initialization code typically in main */
pthread_mutex_init(&ADCmutex, NULL);

/*
Use the mutex to control access to shared item.
This has function scope.
*/
***** Start protected code section *****

/* Lock the mutex */
pthread_mutex_lock(&ADCmutex);

/* Use the protected resource */
GetAnalogueInput(&RotorSpeed);

/* Unlock the mutex */
pthread_mutex_unlock(&ADCmutex);

***** Finish protected code section *****
```

Listing 3.7

3.5 The simple monitor

It was pointed out earlier that there are many limitations and problems with the semaphore construct (and the mutex isn't much better). They really can't be considered to be robust programming constructs. What we want then is a replacement which, in program terms:

- Provides protection for critical regions of code.
- Encapsulates data together with operations applicable to this data.
- Is highly visible.
- Is easy to use.
- Is difficult to misuse.
- Simplifies the task of proving the correctness of a program.

The most important and widely-used construct meeting these criteria is the monitor (which owes its origins to the work of Dijkstra, Brinch Hansen and Hoare). The construct described here uses a simplified version of the original monitor; hence we've chosen the name simple monitor. Fundamentally it prevents tasks directly accessing a shared resource by:

- Encapsulating the resource (the critical code section) with its protecting semaphore or mutex within a program unit.
- Keeping all semaphore/mutex operations local to the encapsulating unit.
- Hiding these from the 'outside' world, i.e. making them private to the program unit.
- Preventing direct access to the semaphore/mutex operations and the critical code section.
- Providing means to indirectly use the shared resource.

These are shown conceptually in figure 3.7, an adapted version of the original semaphore access control technique. First, all component parts are encapsulated, the encapsulating unit having a single entry/exit point (equivalent to entry and exit lanes of a parking garage). Incoming traffic is separated from the outgoing stream, being routed to the access control mechanism. Observe that, in this arrangement, entry drivers can request entry only (equivalent to Wait). Otherwise the access behaviour is as described earlier. Note that all queuing is done within the encapsulating unit.

The Finish button (i.e. Signal) is located on the outgoing lane, and so can be operated only by an exiting driver.

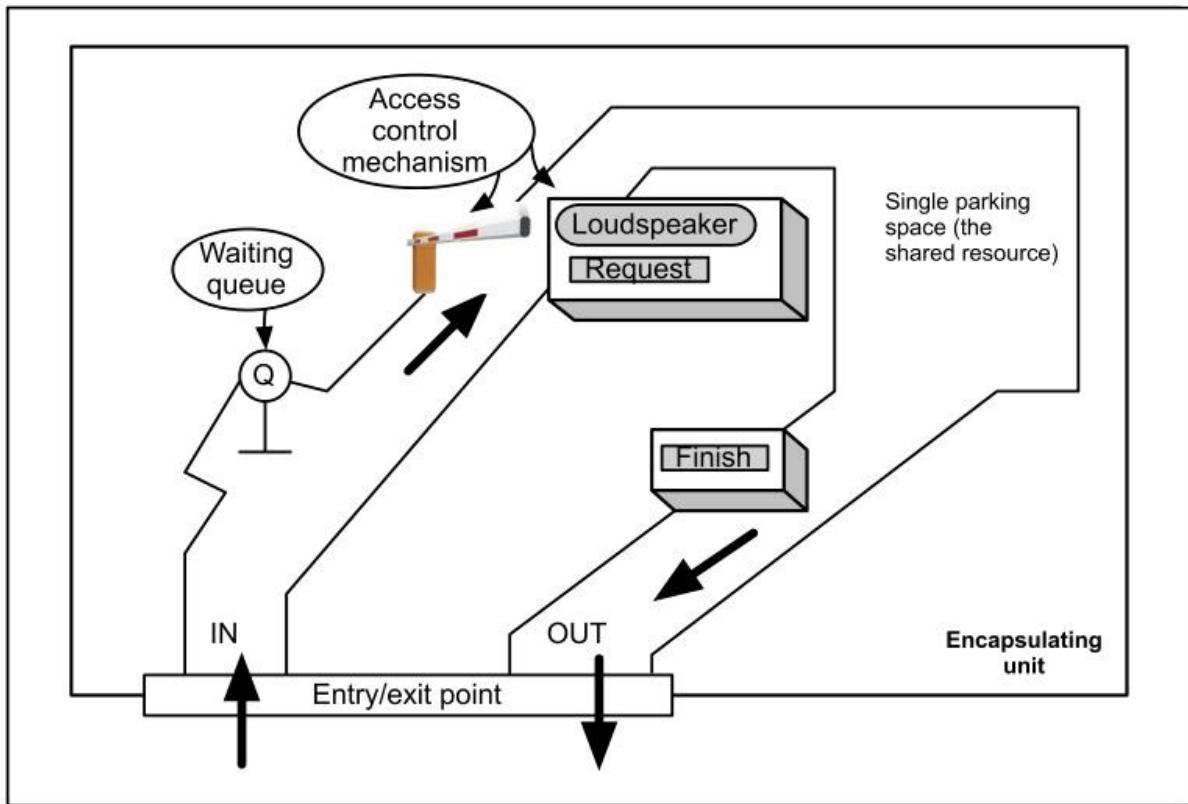


Figure 3.7 Conceptual view of the simple monitor

Some programming languages provide this or a similar construct (e.g. the protected object of Ada95). However, the simple monitor is not normally found in RTOSs. In this case you'll have to build it yourself, a key feature being the encapsulating unit. If you're programming with C++ the obvious choice is the class. This gives us all the required encapsulation, information hiding and public access mechanisms. With C we can mimic the class by putting the monitor software within a .c file and providing its public interface in the associated .h file.

An example of this in action is shown in listings 3.8, 3.9 and 3.10. Here the combination of the .c and .h files provides the overall encapsulation of, and public access to, the 'private' code of the function AnalogueInputMeasurement. This function encapsulates the protected resource with its protecting semaphore; it is impossible to get or put the ADCsemaphore outside the .c file. Also, the code arrangement guarantees that:

- get (Wait) and put (Signal) are paired and are in the correct order.
- The task that calls the function AnalogueInputMeasurement has ownership of the semaphore.
- When the function completes the resource must be available for use (that is, it cannot be left locked up).

```

/* This is in the .c file */
/* A binary semaphore ADCsemaphore has already been created,      */
/* is both visible and in scope to this file only, and is initialized. */

int AnalogueInputMeasurement (int ChannelNumber)
{
    int AnalogueValue = 0;

    /***** Start protected code section *****/
    /* Wait on the semaphore */
    tx_semaphore_get (&ADCsemaphore);

    /* Use the protected resource */
    AnalogueValue = Convert (ChannelNumber);

    /* Signal the semaphore */
    tx_semaphore_put (&ADCsemaphore);
    /***** Finish protected code section *****/

    return AnalogueValue;
} /* end function AnalogueInputMeasurement */

```

Listing 3.8

```

/* This is in the .h file, providing the public interface */

int AnalogueInputMeasurement (int ChannelNumber);

```

Listing 3.9

```

/* This is in the task (i.e. client) code */

const int RotorSpeedChannel = 0;
const int RotorPositionChannel = 1;
int RotorSpeed = 0;
int RotorPosition = 0;

void main (void)
{
.....
    RotorSpeed = AnalogueInputMeasurement (RotorSpeedChannel);
    RotorPosition = AnalogueInputMeasurement (RotorPositionChannel);
.....
} /* end main */

```

Listing 3.10

A mutex could equally well have been used instead of a semaphore. However it doesn't add any value to the design as the overall behaviour in both cases is identical.

In summary, the simple monitor is:

- Used to control access to resources.
- Much more powerful than the semaphore.
- Is easy to use.
- Is difficult to misuse.

Note that in reality the calls to the monitor would actually be embedded in the code of the individual tasks.

3.6 Mutual exclusion - a final comment

The primary purpose of mutual exclusion mechanisms is to control access to shared resources. This includes hardware, shared data and system software (such as the RTOS itself), figure 3.8. Many solutions to resource contention have been proposed; few are used in practice. For small programs, or those with little parallelism, low-level direct methods are quite suitable as long as care is taken. This also applies to designs that don't use formal operating systems (typical of most small embedded functions). In real-time systems the binary semaphore tends to be more widely used than the general (counting) semaphore. There is nothing fundamental about this. It's just that, for the monitoring and control of external devices, the binary semaphore does the job quite well. Even so, semaphores and mutexes have features that can make them unsafe, especially where:

- Programs are large.
- The software is structured as a number of co-operating parallel tasks.

In such instances the simple monitor is a better choice.

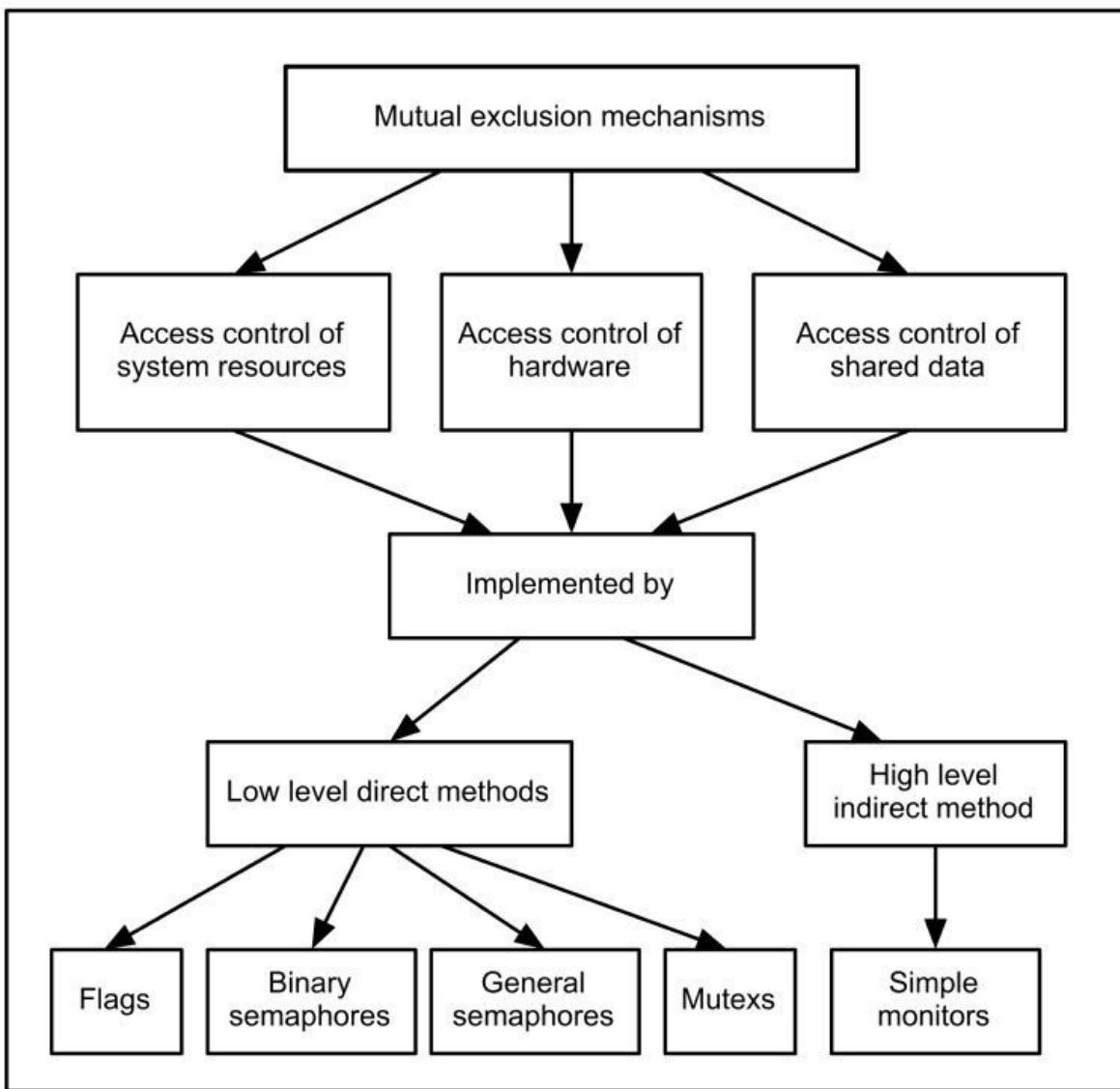


Figure 3.8 Practical mutual exclusion mechanisms

Review

You should now:

- Fully appreciate why tasks cannot have uncontrolled use of shared resources.
- Know what mutual exclusion is and what it does.
- Know the difference between busy-wait and suspend-wait mutual exclusion methods.
- Recognize the advantages of the suspend-wait technique.
- See how single flags can be used to implement busy-wait mutual exclusion.
- Know why certain operations must be atomic.
- Understand the concept and use of the binary and counting semaphores.
- See how the mutex improves on the semaphore.
- Know why semaphores and mutexes are not robust program constructs.
- Understand what a simple monitor is and how it overcomes the weaknesses of the semaphore and mutex.
- Understand the code-level structure and use of the simple monitor.

It's now time to pick up the practical work again, specifically that covered in exercises 5 to 10.

Chapter 4 Shared resources and contention issues

The objectives of this chapter are to:

- Explain why mutual exclusion operations may result in serious run-time problems.
- Show what deadlock is and explain how deadlock-free systems can be produced.
- Describe techniques for preventing deadlocks, highlighting those suitable for fast systems.
- Describe what priority inversion is and why it degrades system performance.
- Explain how priority inversion can be eliminated by dynamically changing task priorities.
- Describe the operations of the basic priority inheritance protocol and the immediate priority ceiling protocol.

4.1 Resource contention - the deadlock problem in detail

In this section we'll look at deadlocks in some detail, first by reviewing why and how they occur. Once this is understood then it is only a small step to work out how to deal with the problem. A number of solutions are possible, some complex, some simple. You will see, however, that only a few are suitable for use in real-time (and especially fast and/or critical) systems.

Let us begin by considering the structure of a typical small control system, figure 4.1.

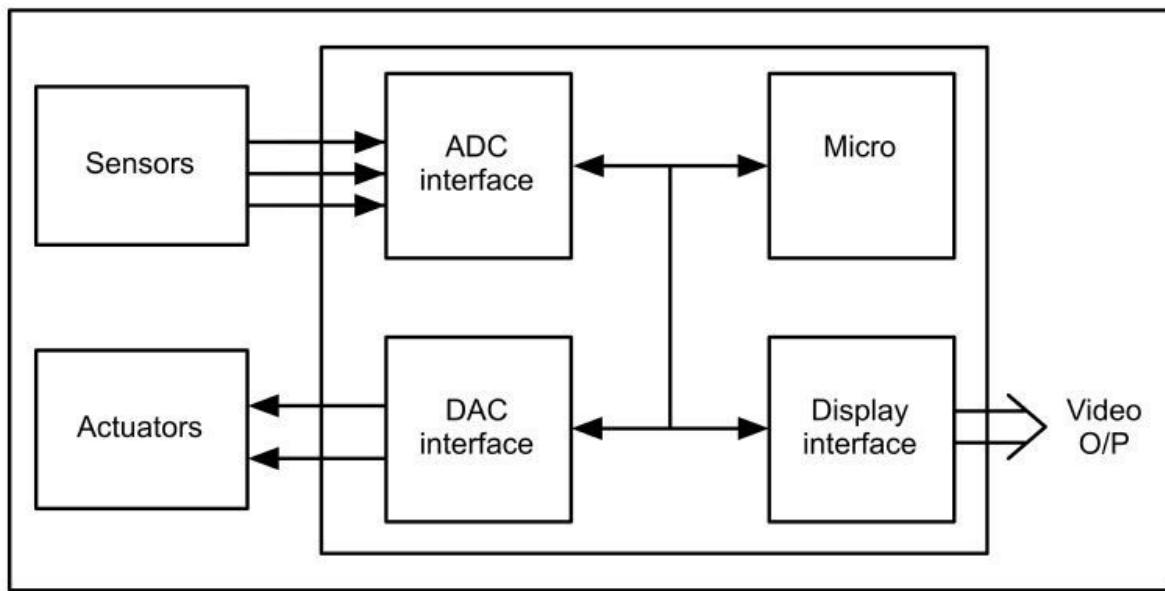


Figure 4.1 Small control system - overall structure

Here we have:

- A set of sensor signals fed in via a single multi-channel ADC interface.
- A set of actuator control signals fed out via a single multi-channel DAC interface.
- All display information generated by a video display interface.

The software in this system is designed to provide multitasking operation. In its simplest form, the relationship between the system devices and the software

tasks is shown in figure 4.2. Here there are three tasks: Control, System Identification (SI) and Alarm.

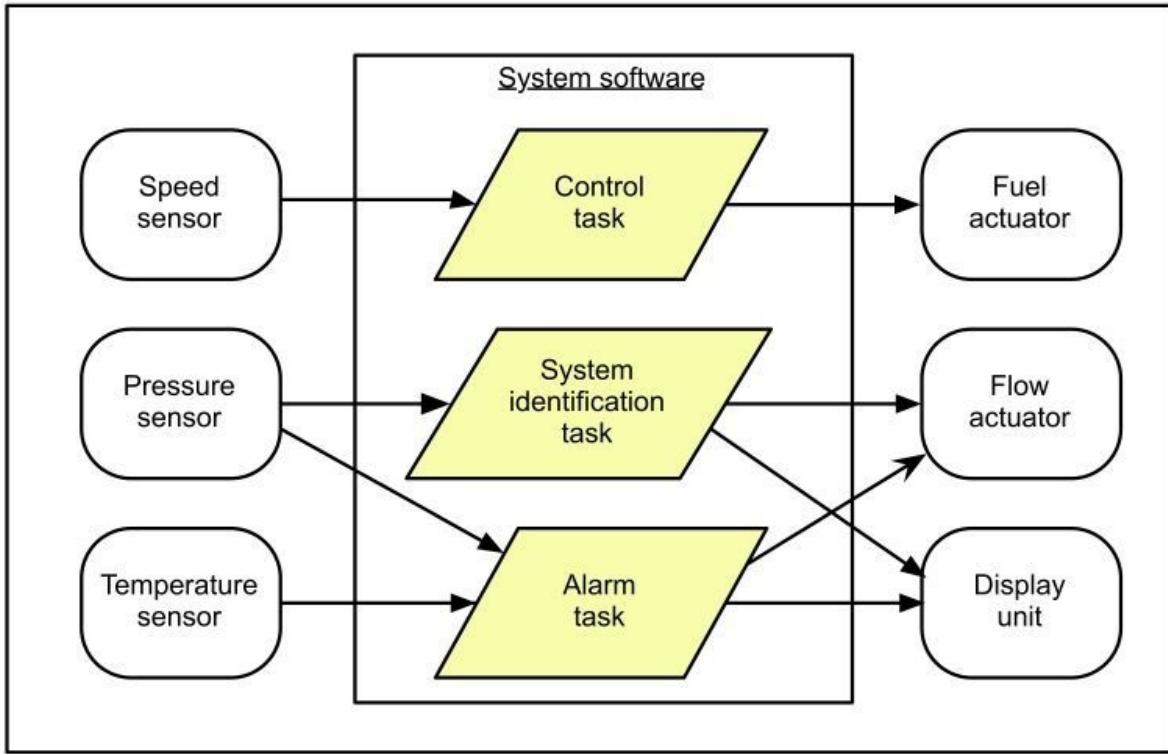


Figure 4.2 Small control system - basic software task structure

The control task is a commonplace closed loop one. It reads in speed sensor data, computes a control signal and sends this out to the fuel actuator. The role of the system identification task is to mathematically identify some part of the physical system. This it does by generating an output signal to stimulate the system via the flow actuator, 'simultaneously' reading the response via a pressure sensor. The alarm task is self-explanatory. From this diagram it can be seen that the control task is functionally independent of the other two. It is tempting also to think that these are independent from the software point of view. However, a more complete diagram of the software structure, figure 4.3, shows this isn't the case.

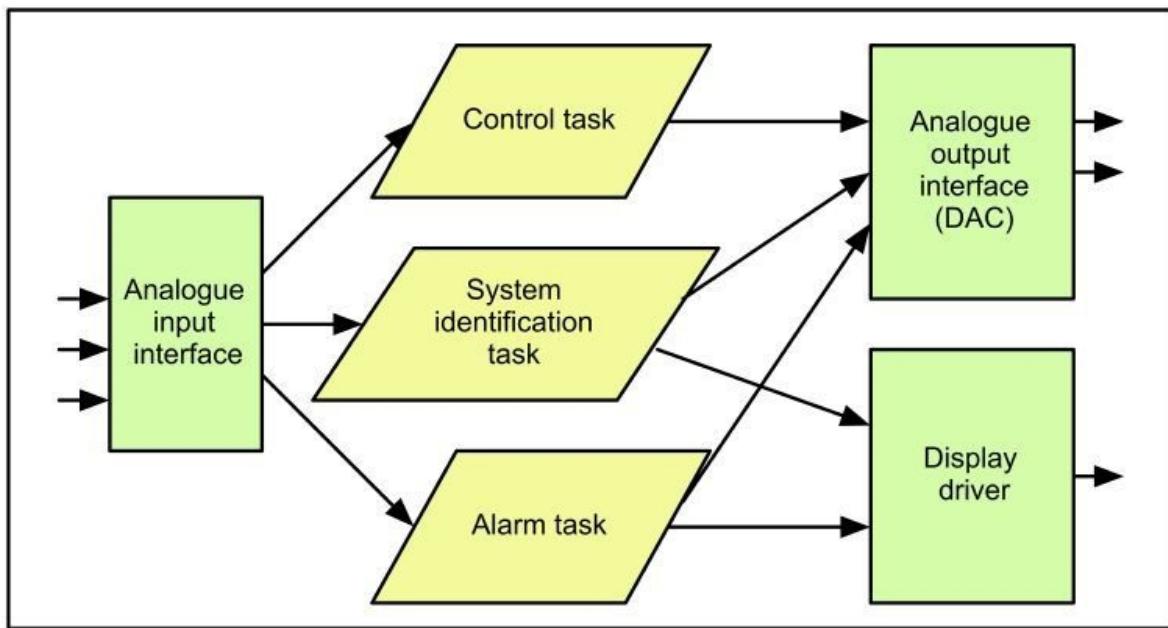


Figure 4.3 Small control system - complete task level software structure

Before running through its operation, however, let us introduce a new tasking model feature: the passive software machine (denoted by a rectangle). This is not a schedulable item; it is used primarily to house shared software facilities. In effect these items become available to all tasks, which call on them as and when required. From a practical point of view they could be implemented as a Java package, a C++ class, a C file, etc. Figure 4.3 contains three passive objects, Analogue Input Interface, Analogue Output Interface and Display Driver.

Now let us review the operation of the Control and SI tasks. The Control task, when activated, first selects the speed input channel of the analogue-to-digital converter (ADC). It then starts conversion, reads the digitised input, computes the output control signal and then releases the ADC. After this it selects the fuel actuator output channel of the digital-to-analogue converter (DAC), updating it with the control signal value. Once this is done the DAC is released.

When the SI task is started it first selects the flow actuator output channel, generates a stimulus signal and sends it to the DAC. It then selects the pressure input channel, starts conversion and, when completed, reads in and stores the input value. After N reads it computes the system identification polynomial, then releases the DAC and the ADC.

Note well; functionally independent tasks may well be coupled together by software functions or resources (as here with the passive objects). For safe

operation mutual exclusion is normally provided within the access mechanisms of shared resources. Such resources are said to be 'non pre-emptible'.

Using non pre-emption, however, has a number of consequences, some quite catastrophic. Let us start by looking at a normal error-free situation, figure 4.4.

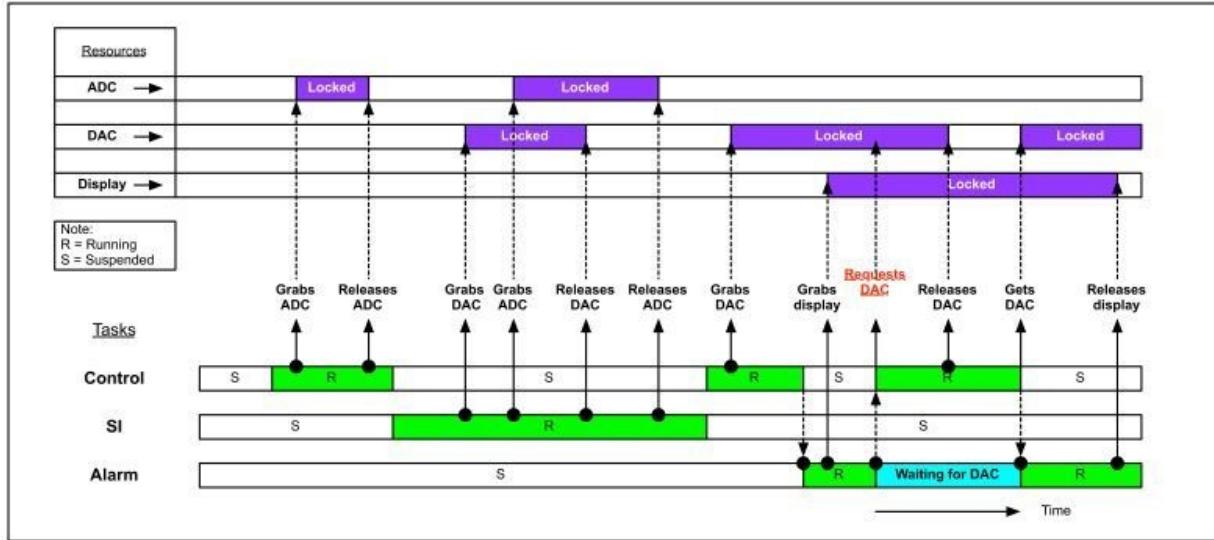


Figure 4.4 Task interaction - error-free situation

This demonstrates the acquisition and release of shared non pre-emptible resources. It also shows what happens when a task (here the Alarm task) tries to access a resource (the DAC) already held by a suspended task (Control task). Given the right set of circumstances the system could run for long periods without faults appearing. We could, given the natural optimism of engineers, believe that all is well with the design. But look at the situation depicted in figure 4.5.

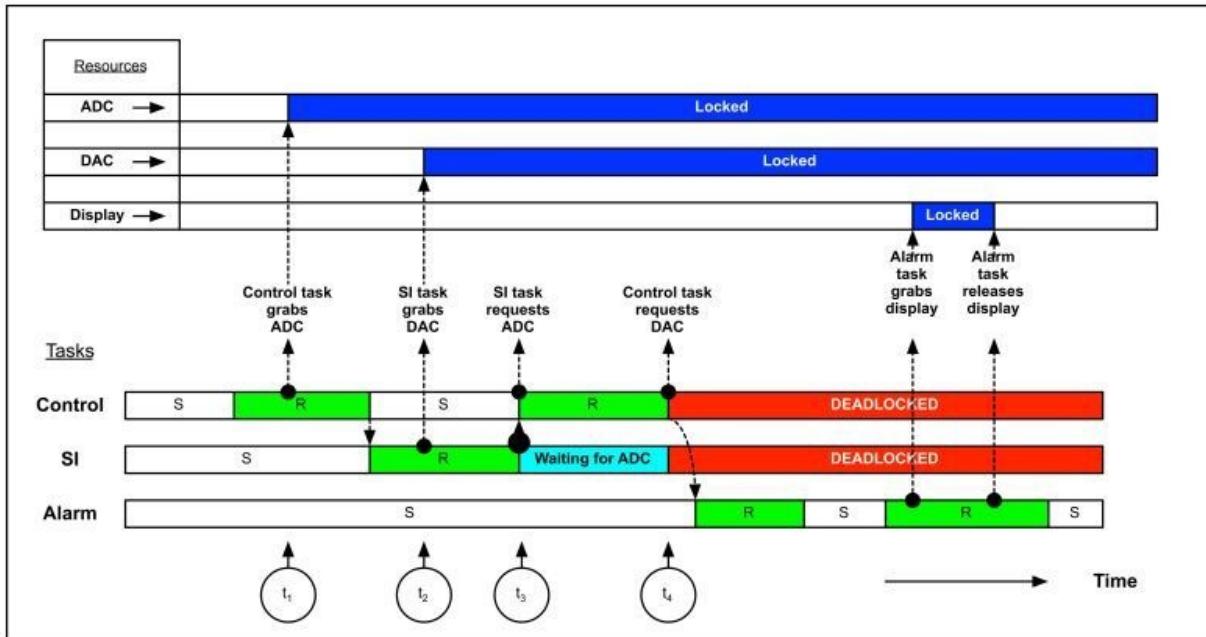


Figure 4.5 Task interaction - deadlock situation

The sequence of events is that:

- At t1 the Control task acquires the ADC.
- Shortly afterwards there is a task switch; the SI task begins to run.
- At t2 the SI task acquires the DAC.
- At t3 the SI task tries to acquire the ADC. As this is held by the Control task, the request fails. The SI task is now suspended and the Control task reawakened.
- At t4 the Control task tries to acquire the DAC. As this is held by the SI task, the request fails. The Control task is now suspended.
- These two tasks are now deadlocked in suspension - neither can proceed.

Note that the other tasks in the system could function perfectly well provided they don't want to use either the ADC or the DAC. This, in fact, illustrates one more important aspect of multitasking, especially in complex, large systems. It may be some time before we realise that a deadlock situation exists; its presence is masked by the amount of ongoing activity.

The sequence of events given in figure 4.5 can be shown in a different way, figure 4.6, demonstrating clearly how deadlock has been produced. Here a task may hold onto a resource while it waits to acquire a second one (logically enough called a 'hold and wait' situation). Note how resource sharing results in a

simple form of circular dependency. It is quite easy to see here how the dependency circle gets formed.

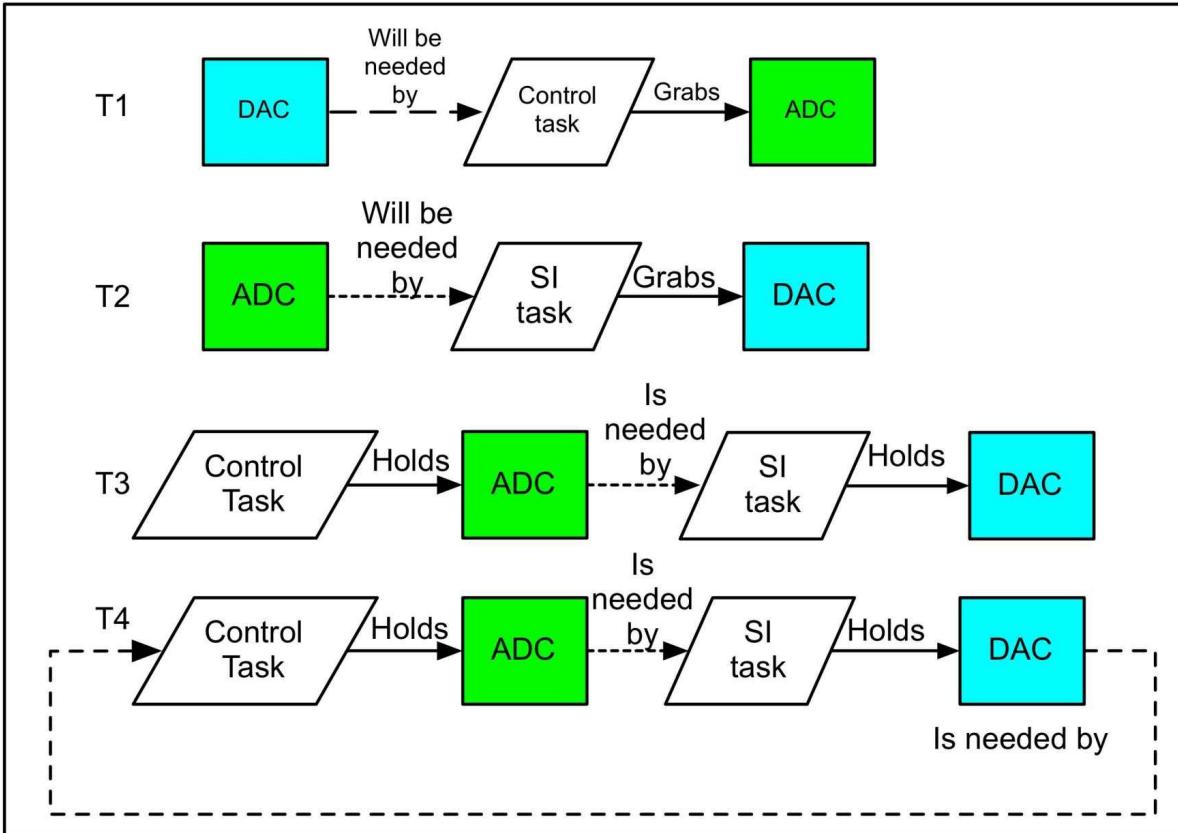


Figure 4.6 Simple circular dependency due to resource hold and wait

In larger systems dependency circles can arise in a slightly different fashion, figure 4.7. Figure 4.7(a) shows a three-task system where each task uses one, and only one, resource. As depicted here there is no resource sharing, hence deadlock can't occur. The situation of figure 4.7(b) is different. Here task 1 uses resource 1 (R1) only but task 2 uses R1 and R2. Similarly task 3 uses R2 and R3. We can highlight the shared resources by redrawing figure 4.7(b), see figure 4.7(c). From this it is clear that deadlock can't occur at this time as tasks share, at worst, only one resource.

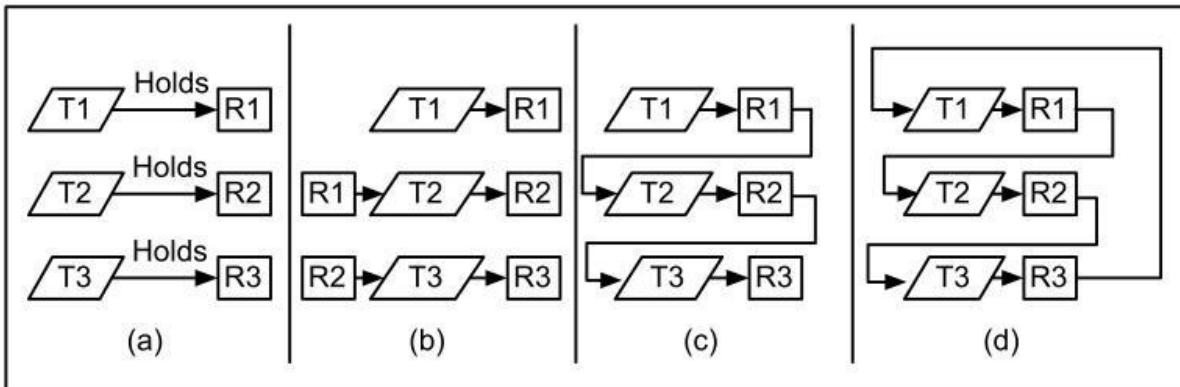


Figure 4.7 Showing circular dependency developing

The result of this dependency is reduced performance, but at least there is no deadlock.

Now consider that (for some reason) T1 needs to use resource R3. The resulting dependency relationship is shown in figure 4.7(d), forming a circular chain of tasks and resources. It is clear that given the right set of circumstances, deadlock will occur

A dependency circle may occur only in rare situations; moreover the dependency may not be obvious from the design diagrams. We stand a good chance of spotting the problem in designs having a small number of statically created tasks (i.e. those that exist permanently while the software is running). This should tell you that it makes good sense to:

- Limit the number of tasks in a multitasking system and
- Avoid changing task numbers at run time (dynamic task creation/deletion).

The key question here is 'what conditions are necessary before deadlock can occur?'. These are listed in figure 4.8.

CONDITION	NAME
A task has exclusive use of resources	Mutual exclusion
A task can hold on to a resource(s) whilst waiting for another resource	Hold and wait
A circular dependency of tasks and resources is set up	Circular waiting
A task never releases a resource until it is completely finished with it	No resource pre-emption

Figure 4.8 Deadlock - necessary preconditions

But note that these by themselves are not sufficient for deadlock to arise.

Summarising: Deadlock can occur only if all four conditions are satisfied simultaneously. However, even in the best of designs we cannot guarantee that, over the life of a system, deadlocks won't occur (poorly-implemented modifications may destroy the quality of the original work). In some situations deadlocks cannot be easily or simply broken; the processor may enter a permanent deadlock state. Hence, without a recovery mechanism, we can lose control of part or all of the processor system. Therefore, for critical systems, some form of error-recovery action must be implemented. Which recovery strategies are applied will depend on the application - the most commonly used measures are discussed below

4.2 Producing deadlock-free systems

There are two basic ways to handle the deadlock problem, figure 4.9. First, make sure that it just isn't possible to get into a deadlock situation. Use prevention or avoidance methods. Second, accept that deadlocks may occur but take steps to overcome the problem.

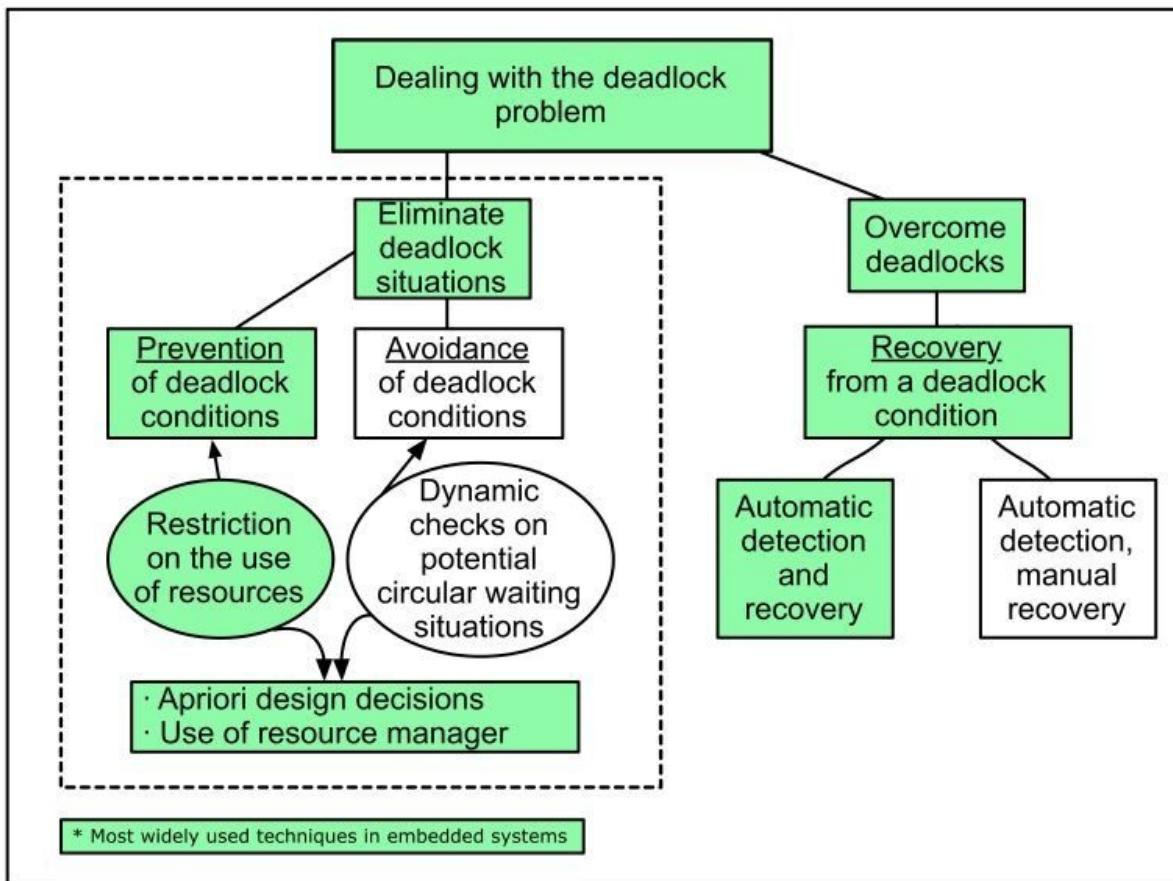


Figure 4.9 Strategies for dealing with deadlocks

To prevent deadlocks, we must ensure that at least one of the conditions listed in figure 4.8 cannot arise. This requires that we make a priori design decisions concerning the use of resources. For complex systems we may have to provide a software resource manager.

Avoidance is based primarily on controlling what happens at run-time. The key is to guarantee that the run-time behaviour doesn't lead to circular dependencies. Central to this is knowledge of:

- Resource/task relationships.

- Resources claimed and in use.
- Resources allocated to tasks but not yet claimed ('preclaimed').
- Resources available for use.

Some form of resource manager will be needed to assess system status and take appropriate action. Such operations must, of course, be performed dynamically as the code executes. For fast systems with frequent task switches the associated run-time overhead is likely to be substantial (and probably unacceptable). As a result deadlock avoidance is rarely used in real-time systems.

Overcoming deadlocks with the assistance of manual recovery techniques may be applicable to desktop computing operations. It is not, though, a serious contender for use in fast and/or embedded applications. Needed here are fully automatic detection and recovery mechanisms. For hard-fast systems this will normally involve watchdog timers.

In general deadlock recovery should be seen as a last resort action, a 'get out of jail' card. Thus to handle deadlocks we usually rely on prevention as the primary mechanism, with recovery as a last line of defence.

Now let's look specifically at deadlock prevention. We can apply a number of policies, figure 4.10:

- Allow simultaneous resource sharing.
- Permit resource pre-emption.
- Control resource allocation to tasks.

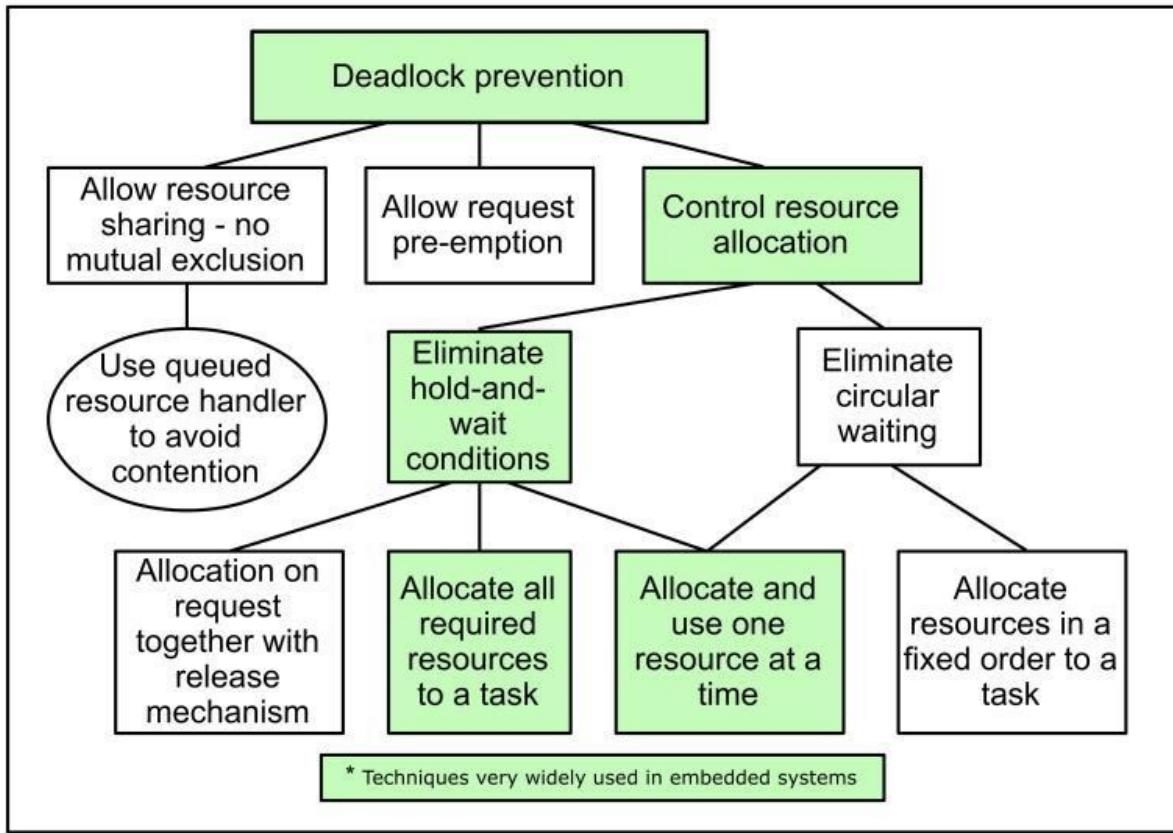


Figure 4.10 Prevention of deadlocks

These may be applied individually or in concert. Let us consider each in turn.

4.3 Preventing deadlocks

4.3.1 Simultaneously sharing resources

With simultaneous sharing, tasks are allowed to access resources whenever they want to; there is no mutual exclusion. This, whilst simplicity itself, is generally unacceptable in any form of critical system; the dangers of unwanted task interference speak for themselves. There is, though, another way of dealing with the access problem: make tasks queue up in order to use the resource. This technique is frequently used with, for example, shared printers, plotters, network interfaces and the like. It is normally implemented by means of a queued resource handler. From the tasks point of view such sharing is - or appears to be - simultaneous (though from the job perspective it is entirely sequential).

Implicit in the approach is that time is not especially pressing.

4.3.2 Allowing request pre-emption

With resource pre-emption a resource held (locked) by a suspended task can be retrieved for use by the running task. Clearly this approach, like simultaneous sharing, can lead to dangerous situations and must be used carefully. However, there are situations where it can work effectively and safely. This is useful, for instance, when looking at the contents of shared data stores or checking the status of I/O devices. The following scenario demonstrates the technique in action. Task A makes a read access to a data store and locks it. Some time later it suspends, being replaced by task B. During execution B finds that it wishes to read the contents of the data store. Even though this resource is locked the OS permits the read to be carried out. In effect the resource is temporarily reallocated. However, should B try to write to the data store, the rules are changed (write access to a locked resource has the potential for generating chaos; thus it is disallowed). Task B finds the resource locked and so suspends on a wait condition.

From this description we can see that a resource manager is needed to make the technique work. Consequently the method imposes a complexity and time overhead, and this may cause problems in small and/or fast systems.

4.3.3 Controlling resource allocation

By controlling the allocation of resources it is possible to eliminate two problems: hold-and-wait and circular waiting. A number of techniques can be used, as follows:

(a) Single resource allocation.

With this method a task can acquire and use only one resource at a time. As a result both hold-and-wait and circular waiting problems are eliminated. This is a very simple, pragmatic and secure technique, especially as all decisions are made in the source code. Its drawback is now the difficulty of accurately predicting run times when a task uses a number of shared resources. For example, when the Control task executes, its operational cycle is:

- Acquire ADC resource, process input signal, release ADC
- Compute algorithm.
- Acquire DAC resource, write to output, release DAC.

Unfortunately, if the control task cannot acquire a resource when it needs it, then unpredictable delays can occur. The result is variability in the actual execution time of tasks from run to run. In extreme cases this may cause serious problems in the system as a whole (it's not just a software issue).

To minimize this problem resource holding times should be as short as possible (in other words, get in, carry out the job and get out as quickly as possible).

(b) Allocation of all required resources.

Here tasks grab all needed resources simultaneously; there is no waiting for other resources. This guarantees that hold-and-wait and circular waiting conditions can't arise. The method is simple, safe and effective but it too has a drawback, that of task blocking (i.e. the blocking of other tasks that share the resources). The result is that the overall performance of the system may be significantly degraded.

(c) Allocation on request.

Here a task requests the use of resources as it executes, gradually acquiring the required items. However, should it ask for a resource that happens to be locked, it suspends; at the same time it releases all resources currently in its possession. When next activated it tries to re-acquire the required set of resources.

This, while it eliminates wait-and-hold, is a complex procedure having quite unpredictable timing behaviour. As such it is very unlikely to be used in real-time systems, especially in fast and/or critical applications.

(d) Fixed order allocation.

The underlying principle of the fixed order allocation strategy is that:

- A task, when activated, requests each resource individually when it requires it (a 'claim').
- If the resource is free it is allocated to the task.
- There is a defined claim order for resources.
- If a second resource is required, a second claim is issued (and so on for further resources). If at any time the first resource is completely finished with, it may be released.

This method is designed to eliminate circular waiting.

It can be very useful - especially in large systems - to show the resource requirements of tasks in a table form. One example is that of figure 4.11. From this it can be seen that task A needs the Display and the UART, task B the ADC and the DAC, etc.

<u>Resource</u>	<u>Task</u>	Task A	Task B	Task C
4. ADC			Needed	Needed
3. DAC			Needed	Needed
2. Display		Needed		Needed
1. UART		Needed		

Figure 4.11 Example task-resource relationship

Let us see how this works in practice, figure 4.12.

Task Resource	Task A			Task B			Task C		
	N	C	A	N	C	A	N	C	A
4. ADC				Y	Y	Y	Y		
3. DAC				Y	Y	Y	Y		
2. Display	Y	Y	Y				Y		
1. UART	Y	Y	Y						
Key: N = Needed C = Claimed			A = Allocated						

Figure 4.12 Example of claim by order in action - 1

As shown here the resources have been numbered 1 to 4. A task must start its claim operation beginning with the lowest number in its own set. Thus task A, when executing, will first claim (C) the UART; if it is free the resource is allocated (A). Likewise, Task B's first claim is for the DAC, whilst task C's first request is for the Display.

The reason for ordering resources - an essential aspect of this method - becomes clear when a number of tasks compete for resources. Consider the situation depicted in figure 4.12 where tasks A and B have both claimed, and been allocated, resources. But in this case as they don't share resources there is no contention issue. Thus they can always execute concurrently without problem. Task C, however, is a different matter as it does share resources with the other two. So let's see how the claim by order method works to eliminate problems, figure 4.13.

Assume that task B is currently the running one and has acquired both the DAC and the ADC. At this point a task swap is made to C, which, at some later stage acquires the Display. It then requests use of the DAC. However, as this is locked by task B the request fails. The result is that task C must give up all resources it is holding; it then suspends. When next activated it once more tries to acquire the required set of resources.

Task Resource	Task A			Task B			Task C		
	N	C	A	N	C	A	N	C	A
4. ADC				Y	Y	Y	Y		
3. DAC				Y	Y	Y	Y	Y	No
2. Display	Y						Y	Y	Y
1. UART	Y								

Key: N = Needed C = Claimed A = Allocated

Figure 4.13 Example of claim by order in action - 2

From this it can be seen that it is impossible for the two following conditions to occur simultaneously:

- Task B holding the ADC whilst waiting to acquire the DAC.
- Task C holding the DAC whilst waiting for the ADC.

Ergo, circular waiting has been prevented.

Some important points can be gleaned from the foregoing scenarios. First, the information contained in the task/resource table shows the tasks that can always execute concurrently. Second, it shows where resource contention may occur, resulting in task blocking. Now, these have important performance (timing) implications. When a task cannot acquire a resource it is forced to suspend and to give up any resources already allocated. Fine. But, given a multitasking shared-resource environment, can we predict when it will be able to acquire all its resources? This is difficult enough where the table is static (doesn't change with time). However, if tasks are created dynamically then things are much more complex; timing predictions may be filed under 'fiction'.

Whilst fixed resource allocation may be effective in soft and/or slow systems, it isn't really suitable for hard-fast applications.

4.4 Priority inversion and task blocking

4.4.1 The priority inversion problem

Deadlock prevention is an essential factor in the design of real-time systems. Unfortunately, even when secure exclusion techniques are included, this may not be the end of our difficulties. In solving one problem - resource contention - we may introduce another one, that of priority inversion.

The basics of priority inversion can be explained easily by looking at the behaviour of a simple two-task (A and B) system. Suppose task A is using a locked resource when the scheduler decides to do a task swap. Assume also that the new task B wishes to use the resource held by A. On checking the access mechanism, it finds the resource unavailable, and will thus suspend. Mutual exclusion is working as planned. But what if B's priority is greater than A? This makes no difference; B still gets blocked. The result is that the low priority task A blocks the higher priority one; B cannot proceed until A allows it to. The system behaves as if the priorities have reversed, a priority inversion. However, this behaviour is exactly what one would expect when mutual exclusion is used; there is nothing abnormal about it.

In a two-task system the performance deterioration is unlikely to be a great problem. But look at the following situation, figure 4.14. Here is a four-task system, comprised (in order of priority) of tasks A, B, C, and D. The system also includes two shared resources, W and X.

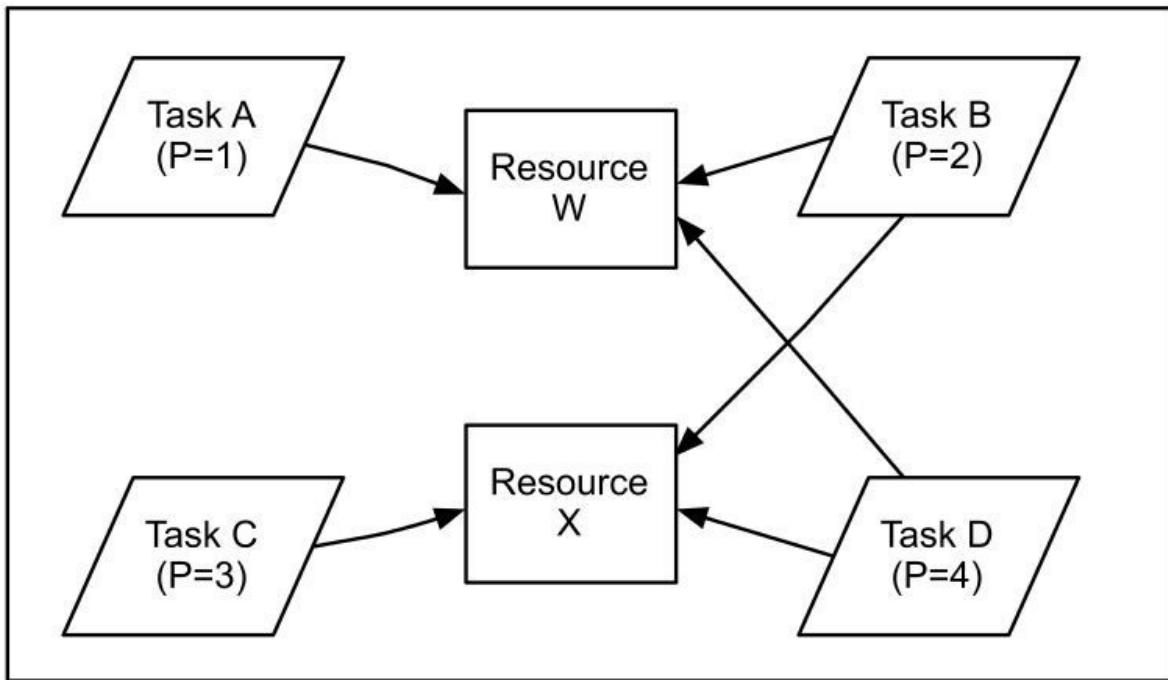


Figure 4.14 Example tasking structure

Please note; the following assumptions are made to simplify the explanation of system behaviour:

- Context switching (rescheduling) takes place only at tick time.
- Tasks can suspend at any time.
- Tasks can be readied at any time.

Let's look at an example run-time scenario, figure 4.15. As shown here, at time t_0 task D is executing; all other tasks are suspended. Before the next tick occurs (at time t_1) D locks resource W. Note also that (by coincidence) all other tasks have been readied. At t_1 task A pre-empts D; D goes to a ready (waiting to run) state. Shortly afterwards A tries to use resource W but finds it locked; it therefore suspends. When t_2 is reached task B is made active, runs to completion, and then suspends. This is repeated for C at t_3 ; then at t_4 D is set executing once more. Only when it releases the lock can A replace it (time t_5).

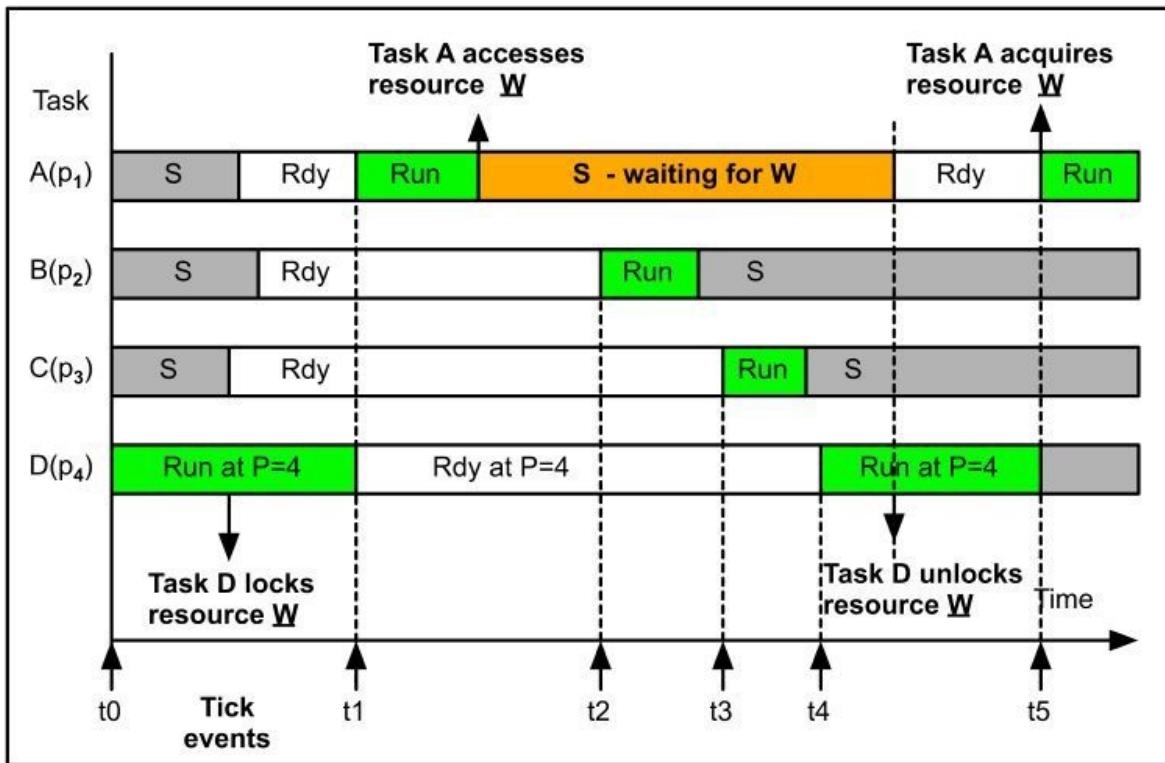


Figure 4.15 The priority inversion problem

In this design A was given the highest priority because it is an important task. Yet it has been forced to wait for all other tasks to execute because of the mutual exclusion locks. Clearly this sort of performance cannot be accepted. How, though, can we prevent this 'chained' priority inversion occurring?

The problem can be tackled in two ways, both involving a temporary increase in task priorities. In the first the priority of a runnable task may be raised to a value determined by other tasks, a priority inheritance technique. In the second, priorities are first assigned to shared resources; then the priority of a running task may be raised to a value set by the resource priority. This is called a priority ceiling technique.

4.4.2 Basic priority inheritance protocol

When using the basic priority inheritance protocol (or simply priority inheritance protocol) a running task can inherit the priority of a suspended task. An example of this in action is given in figure 4.16. Compare this with the scenario of figure 4.15.

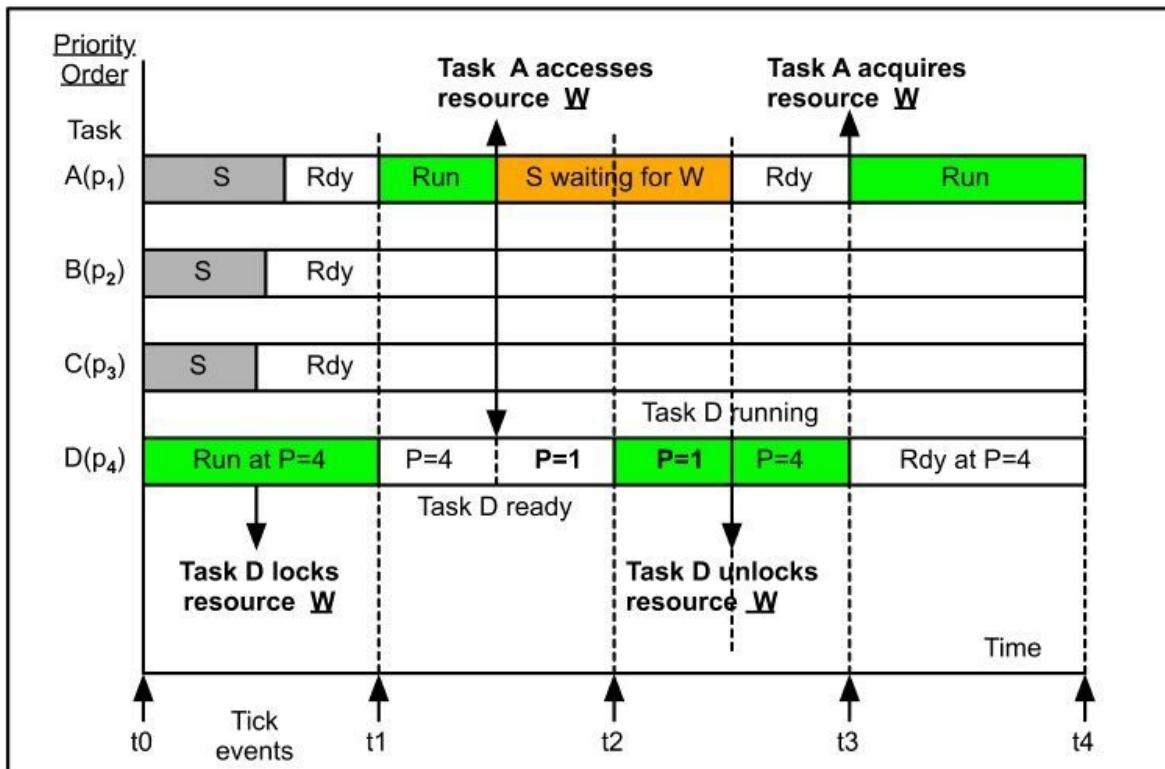


Figure 4.16 Priority inheritance protocol in use - 1

Operation proceeds as before until task A is suspended when it tries to access the locked resource W. At this point the priority of D is raised to equal that of A (i.e. 1). Thus, at reschedule time t2 task D is once more set executing. Slightly later it unlocks resource W (which readies task A), with the result that its priority is returned to normal (4). Consequently, at tick time t3, it is pre-empted by task A, which now begins executing. Thus task A was suspended only while D was using the shared resource.

A second scenario is given in figure 4.17. You should be able to follow this through by yourself (please do so, it is important to understand this topic). One very important point is demonstrated in this diagram; the (perhaps) unpredictable effect of task interactions on system performance. If task 4 hadn't locked resource X then it is possible that task C would have finished executing by time t2; as shown it is still running after time t4.

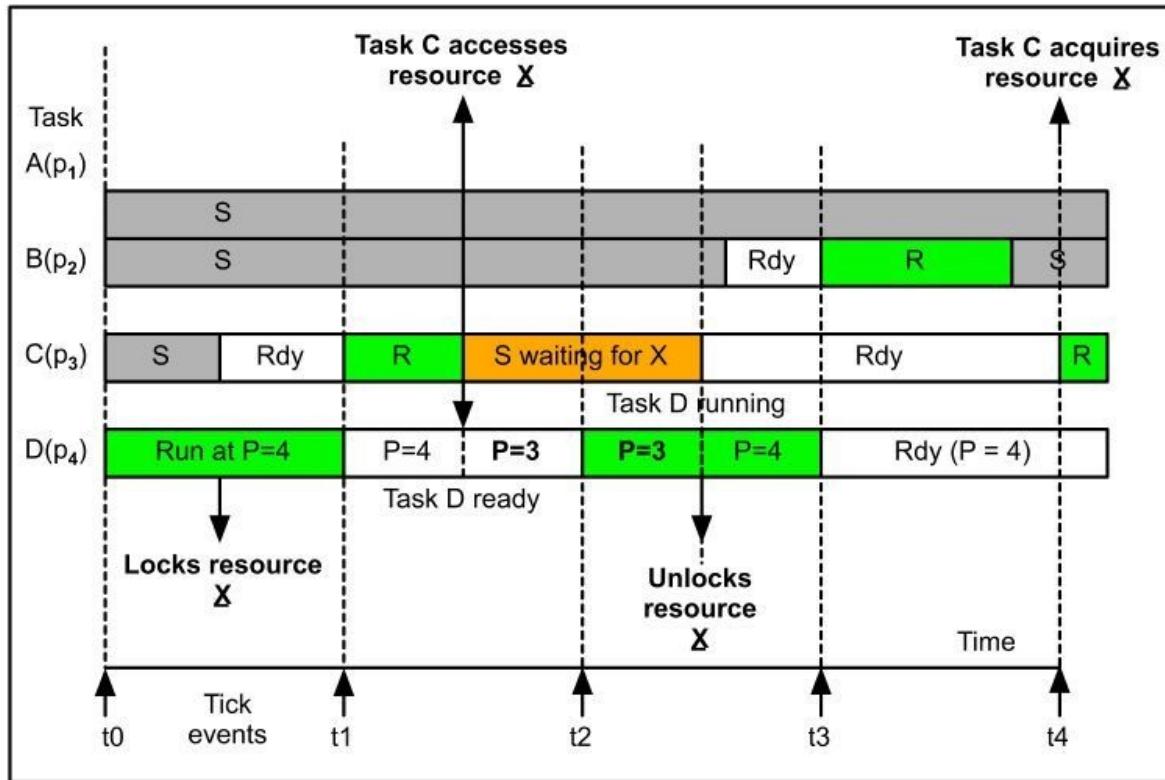


Figure 4.17 Priority inheritance protocol in Use - 2

4.4.3 Immediate priority ceiling protocol

As an alternative to priority inheritance we can use the priority ceiling protocol. The technique described here is a variant of the original protocol, called the immediate priority ceiling protocol (or, for brevity, the priority ceiling protocol). This is simpler to implement and is, in practice, more widely used.

The basis of this method is that each resource has a defined priority setting - set to that of the highest priority task that uses the resource. Thus resource W has a ceiling priority of 1 while that of X is 2. When a task locks a resource it immediately raises its priority to that of the resource priority setting. On unlocking, the task priority goes back to its original value. Thus when a task has acquired a resource it cannot possibly be pre-empted by tasks that share this resource. Note though, it can be pre-empted by any task having a priority higher than that of the ceiling. By definition such tasks do not share the resource being contended for; hence there isn't a priority inversion problem. These points are demonstrated in figure 4.18.

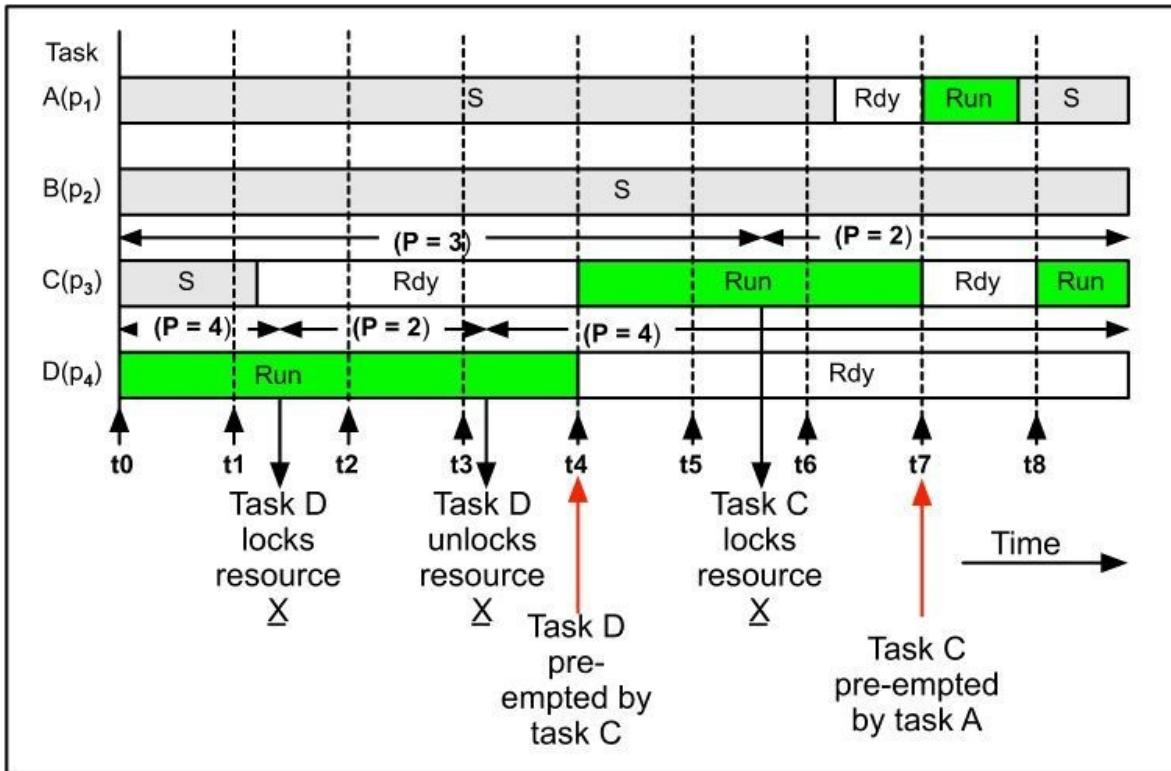


Figure 4.18 Priority ceiling protocol in use

To start with, at tick time t_0 , task D is running (with a priority of 4), all other tasks being suspended. A short time after t_1 task C becomes ready, at priority 3. Soon after that task D locks the resource X, at which point its priority is raised to 2, the resource ceiling value. At the reschedule time t_2 task C cannot pre-empt task D; task D continues executing.

The next event of interest occurs when task D unlocks the resource, its priority then reverting to its preset value of 4. Hence at the next reschedule point, t_4 , task C is now able to (and does) pre-empt task D. It initially runs at priority 3, but when it later locks resource X this rises to 2.

Shortly after t_6 task A is readied, at priority 1. Observe from figure 4.14 that this task doesn't share any resources with task C. Hence there is no mutual exclusion interaction between the tasks vis-à-vis the locked resource X. As a result, at time t_7 , task A pre-empts task C and then runs to completion. Task C remains at priority 2, and resumes at time t_8 .

There is another advantage offered by the priority ceiling protocol; it prevents deadlock (which is not the case for the priority inheritance protocol). I leave it to you, dear reader, to verify that this statement is correct.

These techniques have been used in a number of operating systems and language task models (for example, the protected object of Ada incorporates the priority ceiling protocol). It should, though, be clear that there is an overhead incurred. The system must:

- Keep track of all task suspensions (a task suspension list).
- Keep track of all locked/unlocked resources.
- Dynamically change the priority of tasks when they lock the resource(s).
- Reset task priorities when a resource is unlocked.

All this takes time, which may well pose problems for hard-fast systems.

4.5 Deadlock prevention and performance issues

Deadlock prevention is an essential factor in the design of real-time systems. Avoiding priority inversion - though not essential - is highly desirable. For soft and/or slow applications a number of techniques are available which combine security with good performance. In hard-fast systems, by contrast, our choices are very limited. The simplest approach is to inhibit context switching whilst tasks are using shared resources. A simple way to do this is to disable interrupts. Essentially this is a form of priority inheritance, conceptually raising the running task to the highest possible level. The method is safe, effective and easy to implement. As a result it is widely used. But it is vital that the time spent with interrupts disabled is kept to a minimum.

It can be seen how task interaction may produce significant variation in the time performance of multitasking systems. Such indeterminate behaviour may result in systems failing to meet their specifications. Worse still, they may work correctly most of the time but fail at the worst possible moments. Once again the message here is to limit the number of tasks in a multitasking system.

Review

You should now:

- Know why protecting shared resources may lead to run-time problems.
- Understand what deadlock is and why it occurs.
- Know the necessary preconditions for deadlock to take place.
- Know how recovery can be made from a deadlock situation.
- Understand what techniques can be used to prevent deadlock.
- Understand the meaning of hold-and-wait and circular waiting.
- Realize why circular waiting may be very difficult (or even impossible) to predict where dynamic task creation is used.
- Realize that if a task chooses to suspend (i.e. self-suspend) it should generally release any resources that it has locked.
- Know the different methods used to control resource allocation and what they achieve.
- Understand how priority inversion arises and what problems it can cause.
- Grasp the concept and use of the priority inheritance protocol.
- Understand the concept and use of the priority ceiling protocol.
- Have worked out why deadlock can't occur in a system that uses the priority ceiling mechanism.

Time for more practical work; complete exercises 10 and 11.

Chapter 5 Intertask communication

The objectives of this chapter are to:

- Show why tasks generally communicate and interact.
- Describe how tasks communicate with each other.
- Describe synchronizing and coordinating (non-synchronizing) task interactions.
- Show why, where and when task coordination is to be preferred to task synchronization.
- Introduce coordination flags, event flags and event group flags.
- Explain the concepts and use of the unilateral and the bilateral rendezvous.
- Describe how to implement task-to-task data transfer using pools and queues.
- Show how the mailbox supports data transfer with task synchronization.

5.1 Introduction

5.1.1 Overall aspects of intertask communication

We've earlier seen that tasks may, from a software perspective, be independent. This, however, is the exception rather than the norm; tasks usually interact with each other. And it also turns out that there are three different forms of communication to support such interactions, figure 5.1.

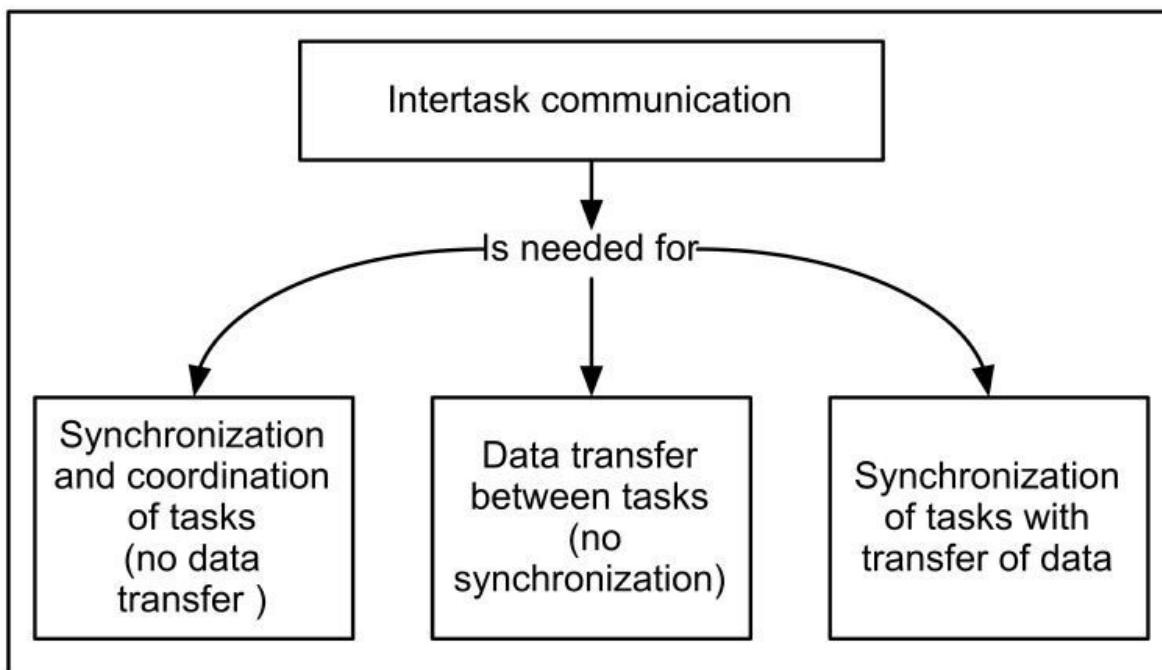


Figure 5.1 Intertask communication features

First, tasks may need to communicate in order synchronize and/or co-ordinate their activities without exchanging data. Synchronization and coordination requirements generally occur where tasks are linked by events (or event-sequences), not data. Such events include time-related factors such as time delays, elapsed time and calendar time. Consider, for example, the requirement that a display task should update status information in response to commands from a keyboard handler task. Here there is no transfer of data from the handler to the display task, merely event signalling.

Second, tasks may have to exchange data but without needing to synchronize operations. In the control system example crew information is controlled by a display output task, the data being obtained from other tasks. Now there is no

reason why these should all work in synchronism. They can very well proceed as asynchronous functions, merely transferring data as required.

Third, tasks may have to exchange data, but at carefully synchronised times. For example, the output from the measurement task is also the input to the computation task - a data transfer requirement. But it is also important that the computation task acts only on the latest information. Thus it works in step with the measurement process - task synchronization.

Separate mechanisms have been developed for each of the three functions to provide safe and efficient operation. Their details are given in the following sections.

5.1.2 Coordination versus synchronization

First, let us by clear what is meant by the terms coordination and synchronization. These are defined by Chambers to be:

Coordinate:

To integrate and adjust (a number of different parts or processes) so as to relate smoothly one to another.

Synchronize:

Cause to happen, move or operate in exact time with (something else or each other)

At times there is a somewhat fuzzy line between task synchronization and task coordination. However, the key difference is the phrase 'exact time'. Coordination ignores timing exactness; fundamentally it sets out to make sure that tasks run:

- In the correct order and/or
- When specific conditions are met.

For example, we might be required to implement the following specification:

'The compressors may not be started until all interlocks are clear and all alarms are validated'

Let us suppose that we design the software to include an interlock task, an alarm task and a compressor task. To comply with the specification the compressor task must delay start-up until the other tasks have completed specific work. But it doesn't matter:

- About the completion order of the interlock and alarm tasks.
- When the compressor task checks to see if it is ok to initiate the start-up sequence.
- That once the interlock and alarm tasks have provided the required event information (interlocks clear, alarms validated) they continue to do other things
- If the compressor task carries on doing other operations while waiting for the required conditions to be met.

Thus there is no inherent need for either senders or receivers to enter a waiting or suspended mode while coordinating activities.

This is quite different from operations that require synchronization of their activities. For example, in figure 5.2 we have an example of automated materials handling that uses two robots; a pallet transport robot and a materials loader/unloader (palletizer) robot. The transport robot's function is to move the pallets about the factory, while the palletizer function is to unload and/or load items onto the pallets (palletizing). All operations are controlled by software, in particular a transport task and a palletizing task.



Figure 5.2 Automated materials handling

(See videos of typical palletizer operations on youtube, ‘Lambert Material Handling’).

Before material can be transferred to/from the pallet both robots must be in their correct position. What we have to do is:

- Position both robots correctly (at the synchronization or *rendezvous* point).
- Perform the required loading/unloading actions.
- Resume individual robot operations.

But as these are independent units we can never predict which one will first be ready for palletizing to begin. Hence, if the transport robot is the first one in position, it must wait until the palletizer robot is ready. Likewise, if the palletizer robot is first, it must wait until the transport robot is ready. What this means for the code design is that we must:

- Identify, in the code of each task, exactly where synchronization is to take place and
- Insert synchronizing mechanisms at these points.

Different constructs are used to support software coordination and synchronization, these being condition flags, event flags and signals (figure 5.3). Flags can be set, cleared or read, while for signals the operations are wait, send and check (the terminology here has been chosen for preciseness, clarity and historical usage in embedded systems). Please note; in many RTOSs the predefined flag construct is the event flag.

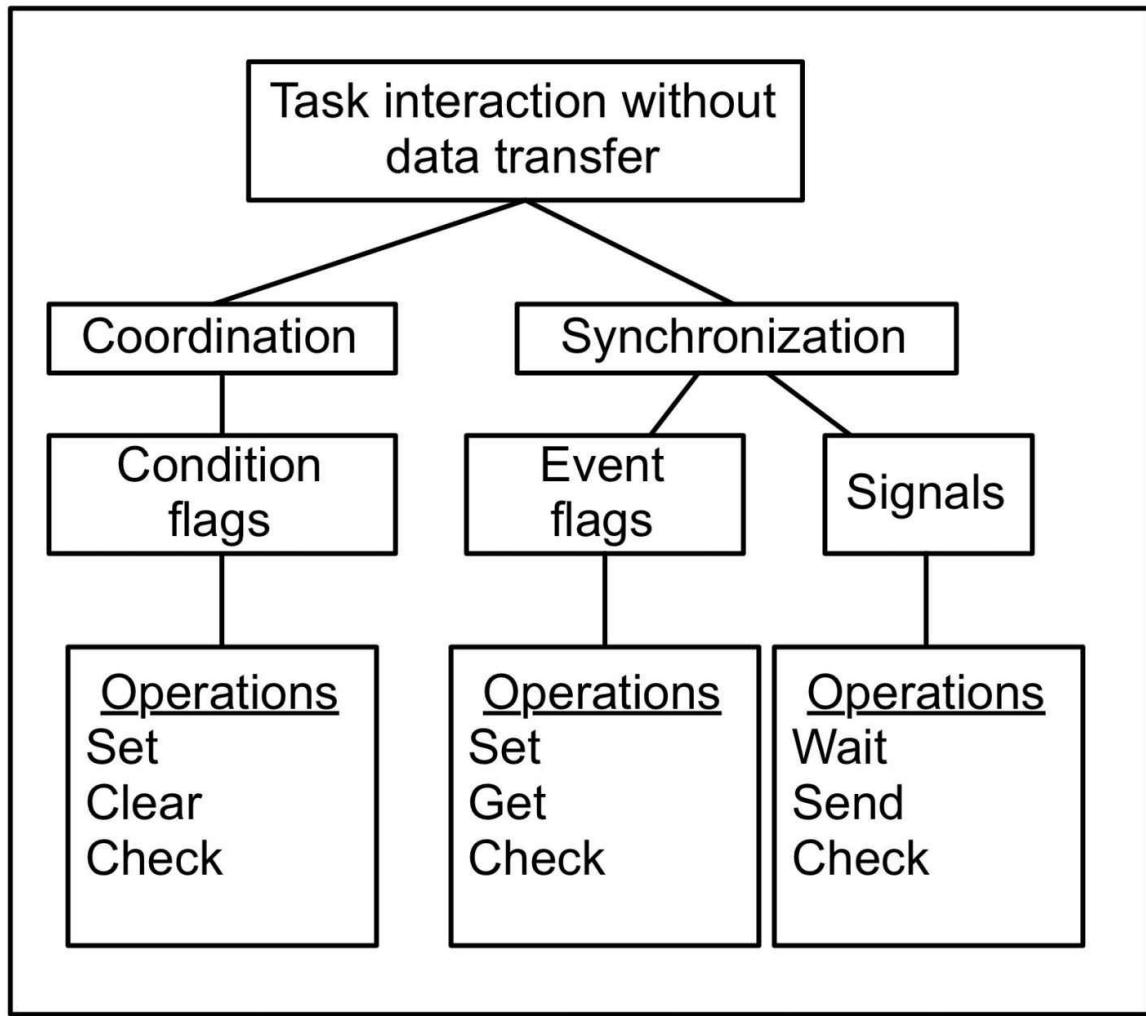


Figure 5.3 Coordination and synchronization constructs

5.2 Task interaction without data transfer

5.2.1 Task coordination mechanisms

(a) The simple condition flag.

We have already covered various applications of flags; the basic ideas are straightforward enough. Earlier it was shown how can may be used as a busy-wait mutual exclusion mechanism. Here, though, their function is to allow tasks to coordinate their activities; in this role they are called condition flags. Please note that here the term condition is used to avoid confusing these with event flags. In practice condition flags are simply called flags (figure 5.4).

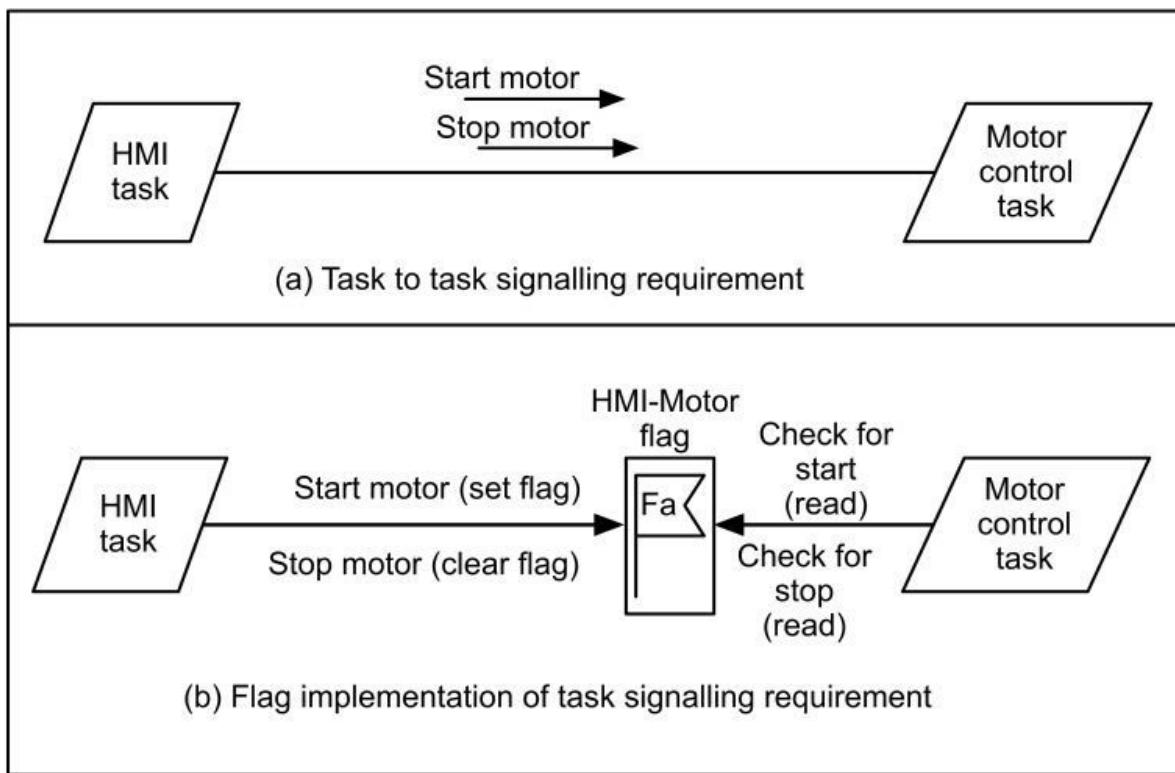


Figure 5.4 Simple use of a flag for coordination

Consider the simple signalling requirement shown in figure 5.4(a). Here an HMI task sends start and stop commands to a motor control task. These commands are generated in response to inputs from an operator's keypad (not shown). Both tasks need to run continuously for the system to function correctly. Now, the simplest way to implement this requirement is to use a global variable. This is also a poor approach; the problems of globals are well known. A fundamental

rule of task-based design is that all intertask communications are routed via the appropriate comms components. Figure 5.4(b), which is self-explanatory, shows how a flag can be used in this application.

At the code level, a condition flag is a binary (two-valued) item that is best implemented as a Boolean. If that data type isn't available then a word, byte, bit or enumerated type can be used (RTOS-defined data types don't usually include an ordinary flag as this is an elementary data type).

The design given here, while it satisfies the requirements, isn't especially robust. One weakness is that we have to know at the code level that set means Start and clear means Stop. As a result it is:

- Easy to confuse the two and get it wrong
- More difficult to pick up mistakes when reviewing the code.

It is far better to avoid the words set and clear but instead use a form of self-documenting code, as for example (Listing 5.1):

```
typedef enum {Start, Stop}      StartStopFlag;  
StartStopFlag HMImotorFlag = Stop;
```

Listing 5.1

It is also recommended that all comms components be encapsulated in a communications class and/or communications file.

For many applications this solution is perfectly acceptable. However its weakness is that a single mistake (e.g. using Start instead of Stop) could produce serious problems in the real world. What we have here is a case of one valid command being turned into a second valid command; there is no redundancy of information. A more secure design is that of figure 5.5, where each individual command has a corresponding flag.

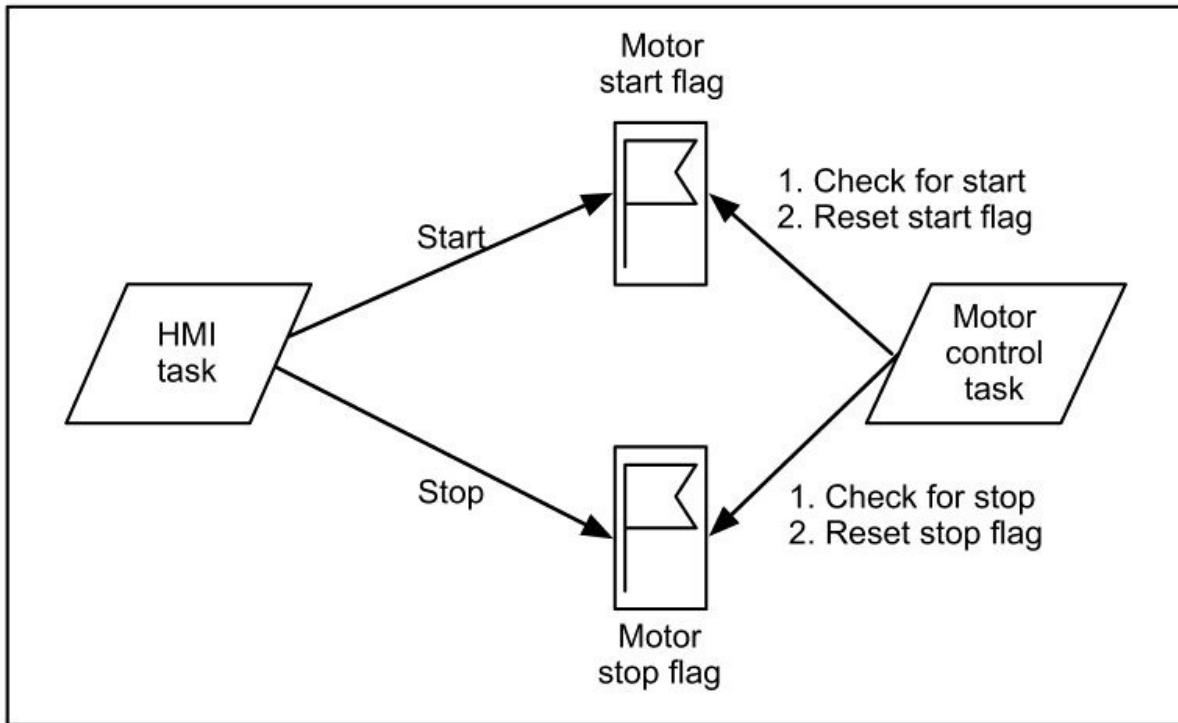


Figure 5.5 Improved use of flags for coordination

The rules used here are that:

1. The HMI sending task checks to see that a flag is reset before changing its state.
2. The Motor control (receiving) task checks the state of both flags before acting on a command. After a successful check the commanding flag is then reset.

The essential code constructs are shown in listing 5.2 (this, of course, is not the only rule set that can be used).

```

typedef enum {StartSet, StartReset} StartFlag;
typedef enum {StopSet, StopReset} StopFlag;

StartFlag MotorStartFlag = StartReset;
StopFlag MotorStopFlag = StopReset;

```

Listing 5.2

(b) Condition flag groups

Let us now look at the flag group (figure 5.6) where we group a set of flags together into a single unit. Quite frequently the flag group is a single word, each flag being a bit within the word.

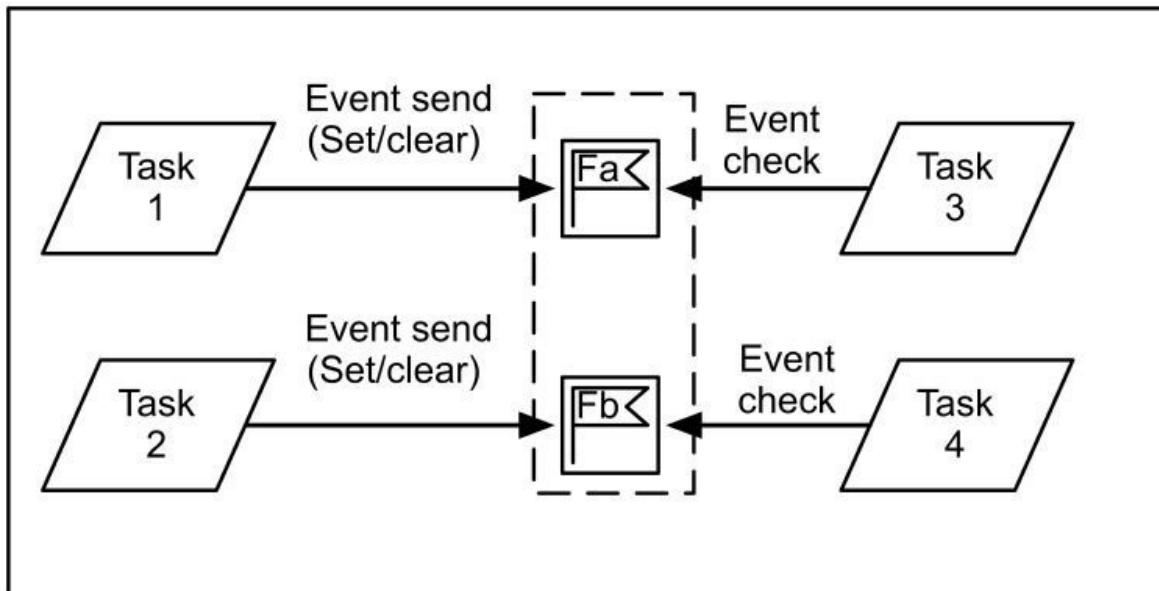


Figure 5.6 Task coordination - condition flag group

As a result:

- Each bit can be changed individually or
- The whole group can be modified using a single write command or
- Sets of bits can be changed using bit masking techniques.

As shown in figure 5.6 a single flag group can replace a number of individual flags. However, a word of caution; it is best if you don't base your normal flags on such structures. One simple programming mistake can wreak havoc in your system. We'll leave you to work out why.

The flag group is especially useful in two particular situations:

- Where a task is waiting on a set of events (figure 5.7) and
- Where a task broadcasts an event to a number of other tasks (figure 5.8).

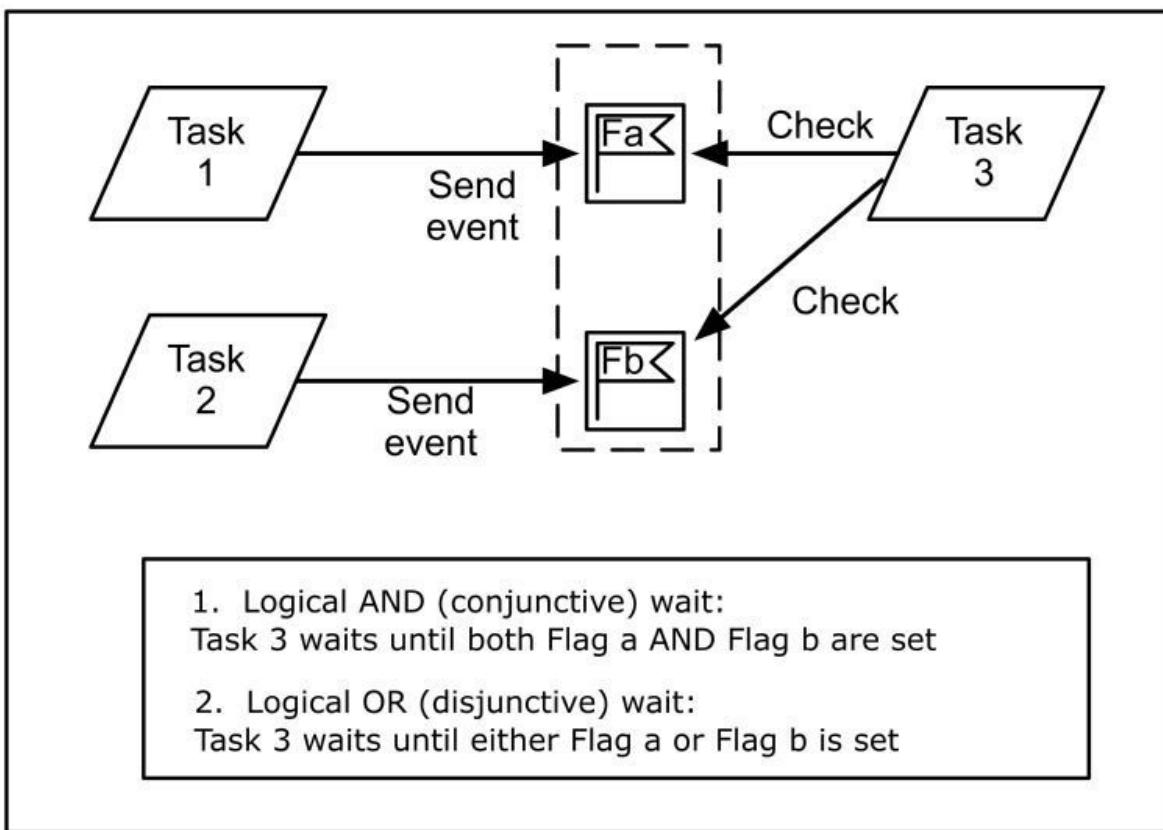


Figure 5.7 Condition flag group - waiting on a set of events

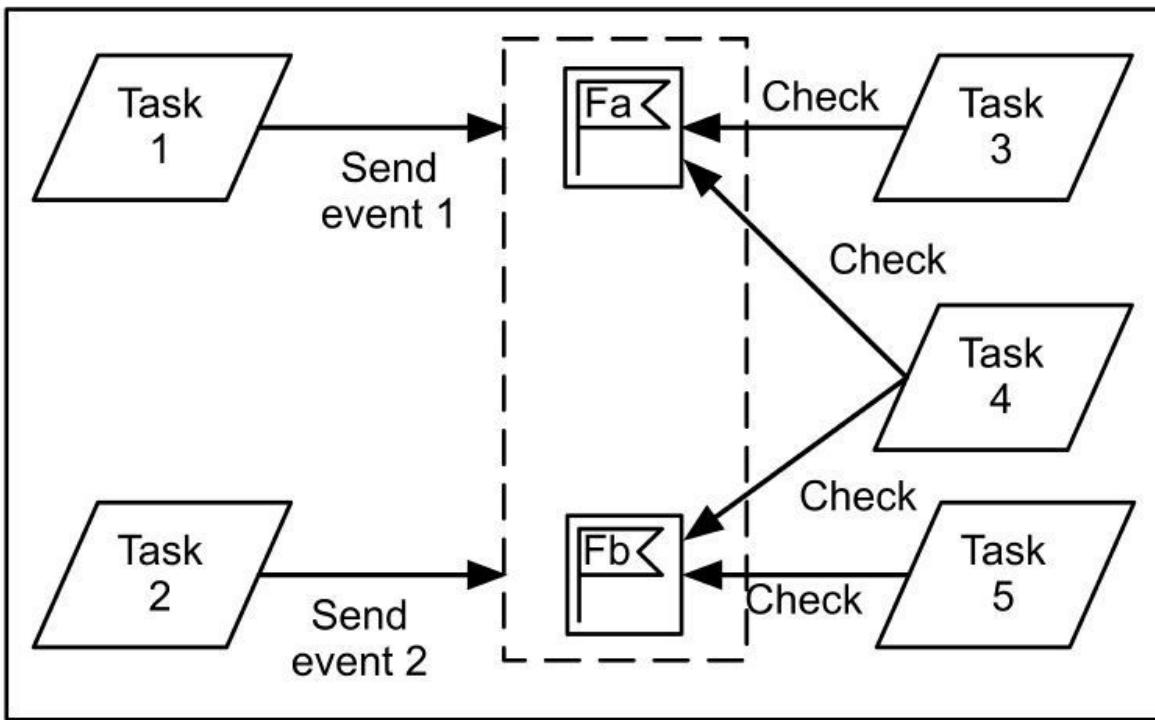


Figure 5.8 Condition flag group - broadcast function

As shown in figure 5.7, the flag group makes it easy to implement combinational logic operations, viz:

- Logical AND: the motor may be started only when the interlocks AND the alarms are clear.
- Logical OR: the motor must be stopped if overspeed OR high temperature is detected.
- Complex decisions: The tank is to be emptied automatically when fermentation is complete AND the lye temperature has fallen below 30°C OR if the operator manually selects EmptyTank.

Figure 5.8 shows the broadcast feature in action. Here flag Fa allows task 1 to broadcast to tasks 3 and 4, whereas flag Fb supports task 2 broadcasting to tasks 4 and 5.

5.2.2 Task synchronization using event flags - the unilateral rendezvous

The unilateral ('one-way') rendezvous is a limited form of synchronization but can be very effective in some situations. Consider a requirement in a fuel tank

protection system that, as soon as a flame is detected, gas suppressant bottles must be fired. To prevent explosion the response must be very fast, typically less than 10 milliseconds from detection to completion. Our implementation would probably have the flame detector generate an interrupt that immediately invokes the fire suppressant task. Thus the software interaction involves two tasks; the flame detection interrupt service routine (ISR) and the fire suppressant task. The ISR is a sender task, the other one being the receiver. Note that this receiver task is an aperiodic one. *Normally it is in a suspended state, waiting to be woken up by (waiting to rendezvous with) the sender task.* By contrast, the sending task does not wait for synchronization with the receiver; it merely signals that a synchronizing condition has been met. Such unilateral synchronization is depicted in figure 5.9. Here event flags are used to support task-to-task interaction, the basic rules being:

- ISR1(2) is a sender task. Task1(2) is its corresponding receiver.
- The event flag is initialized to the cleared state (flag value = 0).
- When a task calls Get on a cleared flag it is suspended.
- When a task calls Get on a set flag (flag value = 1) it clears the flag and continues executing.
- When a task calls Set on a cleared flag, it sets that flag and continues executing. If a task is waiting suspended on that flag it is woken up (readied).
- When a task calls Set on a set flag it just continues executing.

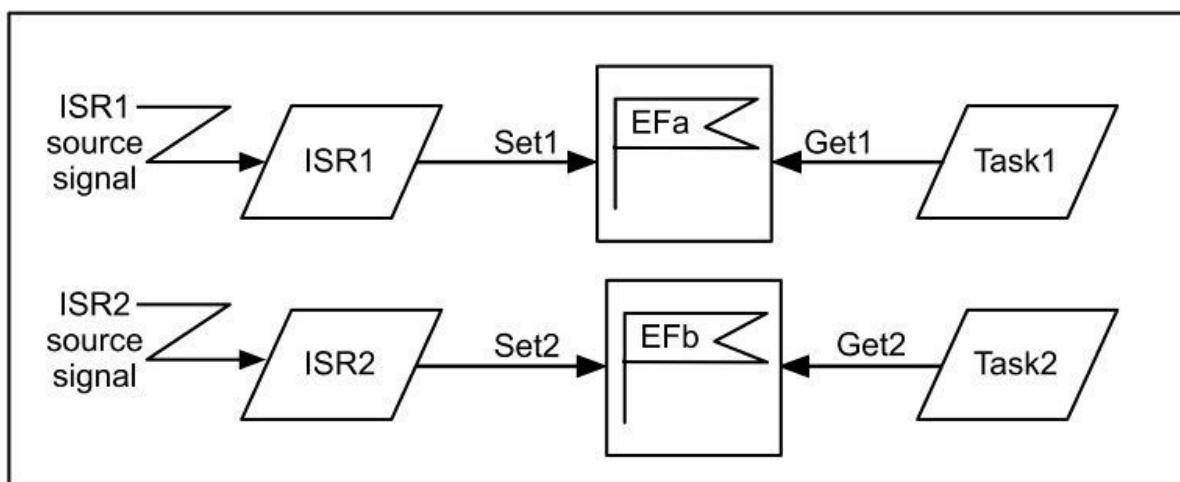


Figure 5.9 Event flags and unilateral synchronization

So here in figure 5.9, for example:

- If Task1 calls 'Get1' before ISR1 has generated 'Set1', it is suspended. When 'Set1' is subsequently sent to the flag (EFa), Task1 is readied.
- If 'Set1' is generated before Task1 calls 'Get1' (i.e. the ISR task reaches the synchronization point first), ISR1 does not stop but continues execution. However the Set call leaves flag EFa in the set state. As a result, when task 1 calls Get, it first clears the flag and then carries on executing. But please note: unilateral synchronization is normally used to enable a sender task to wake up a receiver: a case of the ‘deferred server’.

Let's look at a simple practical example based on the use of ThreadX (figure 5.10), the event flag group being a single 32-bit word.

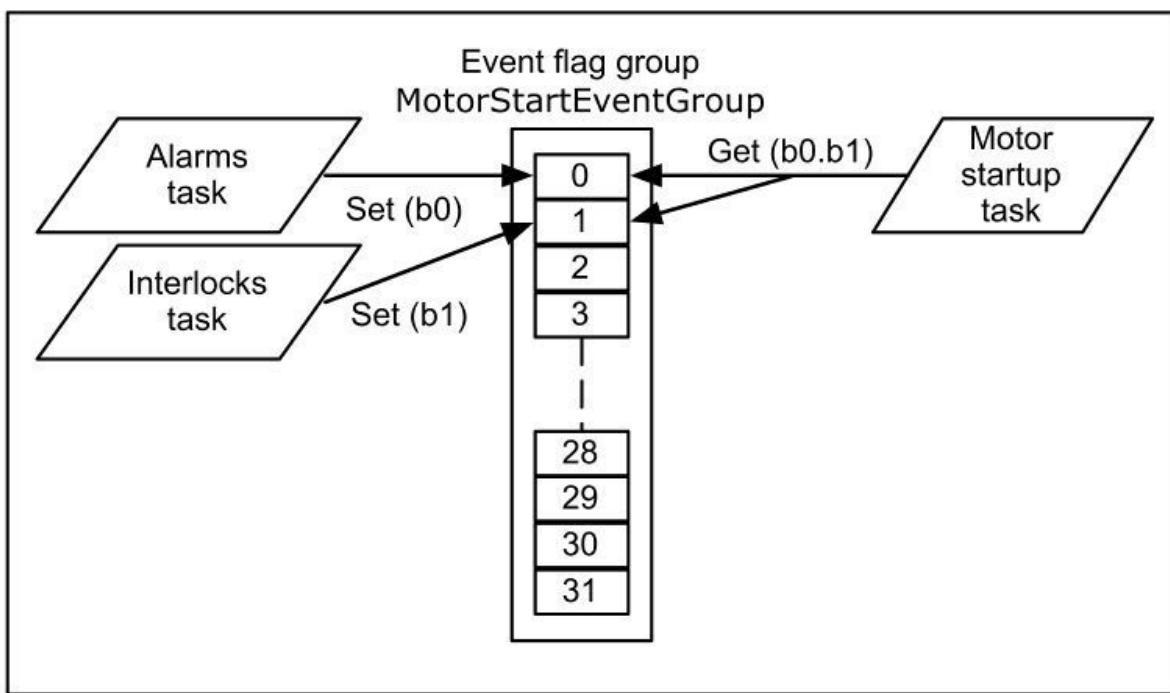


Figure 5.10 Event flag group - waiting on sets of events

Its purpose is to implement the specification that 'the motor may be started only when the interlocks AND the alarms are clear' using a unilateral rendezvous. When the interlocks are clear, the Interlocks task sets bit 1. Likewise, the Alarms task sets bit 0 when the alarms are clear. The receiver task, Motor startup, cannot proceed past the synchronization point unless both bits are set (i.e. word value = hex 00000003, for brevity written as 03(H) or 0x03).

Listing 5.3 shows the essential declarations and the creation of the event flag

group it. Listings 5.4 and 5.5 show how the individual bits of the flag group are set, whilst listing 5.6 demonstrates the action of the receiver task. The associated comments should make it clear what the code units do.

```
/* ===== Declarations ===== */
TX_EVENT_FLAGS_GROUP MotorStartEventGroup;
uint GroupCreationStatus;
uint SetServiceStatus;
uint GetServiceStatus;

IN THE CODE:
/* ===== Create the event flag group===== */
*/
/*
   All bits are automatically initialized to zero, i.e. cleared
*/
GroupCreationStatus = tx_event_flags_create (&MotorStartEventGroup,
                                             "MotorStartEventGroup");
```

Listing 5.3

```
/* ===== The Set action in the Alarms Task ===== */
/*
   This logically ORs the flag value with hex 01 to set bit 0
*/
SetServiceStatus = tx_event_flags_set (&MotorStartEventGroup, 0x01, TX_OR);
```

Listing 5.4

```
/* ===== The Set action in the Interlocks Task =====*/
/*
   This logically ORs the flag value with hex 02 to set bit 1
*/
SetServiceStatus = tx_event_flags_set (&MotorStartEventGroup, 0x02, TX_OR)
```

Listing 5.5

```

/* =====The Get action in the Motor Startup Task ===== */
/*
This is the synchronization point.
The task waits suspended indefinitely until bits 0 and 1 are set (i.e. the flag value is
hex 03 if all other bits are ignored)
When this happens the event flags are cleared and the task proceeds.
*/
GetServiceStatus = tx_event_flags_get (&MotorStartEventGroup, 0x03,
                                         TX_AND_CLEAR, TX_WAIT_FOREVER);

```

Listing 5.6

5.2.3 Task synchronization using signals - the bilateral rendezvous

To synchronize the materials handling robot activities shown in figure 5.2, two-way or 'bilateral' interaction (the bilateral rendezvous) is needed. Such synchronization is achieved by using signals, these being 'Wait', 'Send' and 'Check' (figure 5.11).

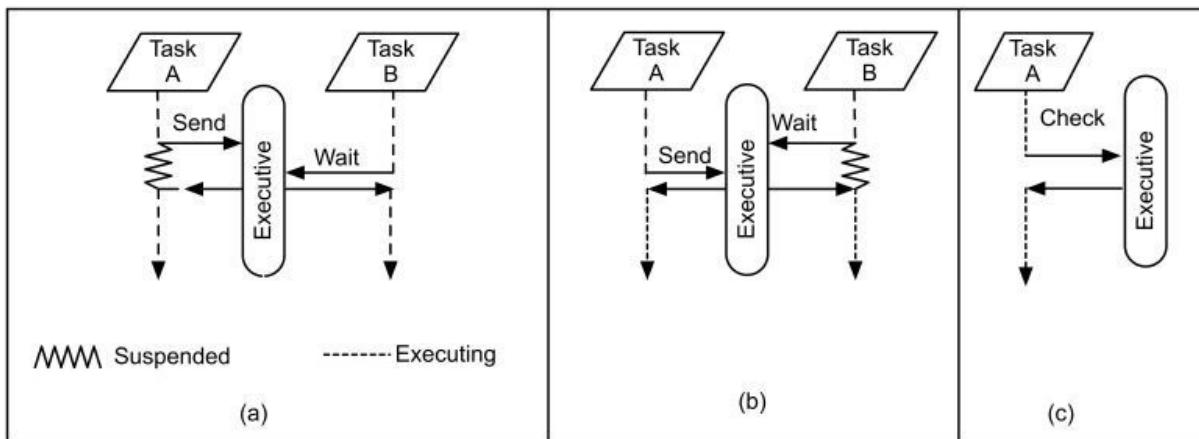


Figure 5.11 Task synchronization using signals

Signalling activities are the responsibility of the executive; to the user such operations (described below) are transparent.

First consider the 'Send' action, figure 5.11(a). Here task A executes its program and reaches a point where it sends a signal (effectively to the executive). At that instant no tasks are waiting to receive this signal; consequently task A is

suspended. Some time later task B generates a wait request for the signal sent by A. It (task B) picks up the signal and carries on executing. The wait request also restarts Task A.

What happens if B generates the wait before A has sent the signal, figure 5.11(b)? The result; task B is suspended until A sends the signal. At this point task A wakes task B and then continues executing.

Flexibility can be added to the signal construct by allowing tasks to decide if they wish to get involved in synchronization. The Check operation, figure 5.11(c), surveys the status of the signal but does not itself halt task execution. Such decisions are left to the checking task, a technique that can be used very effectively for polling operations.

In practice, signals are usually realized using functions, as follows (listing 5.7):

```
/* Sends a signal to the synchronizing signal named SignalName.  
Suspends if no tasks are waiting for the signal */  
void Send(SyncSignal SignalName);  
  
/* Waits for a signal. If the signal is not present when the request is  
generated, the task suspends. Otherwise the sender is reactivated and the  
system rescheduled */  
void Wait(SyncSignal SignalName);  
  
typedef enum { false, true } bool;  
  
/* Checks to see if a task is waiting to send a signal. Returns true if a  
signal is present */  
bool Check(SyncSignal SignalName);
```

Listing 5.7

Now for a number of important points:

1. First, there isn't a one-to-one link between tasks; task pairing is not specified in these constructs.
2. Second, tasks are considered to be both senders and receivers.
3. Third, signals aren't associated with specific tasks. All that is required that one task has the Wait whilst the other has the corresponding Send.
4. Fourth, they exhibit the insecurities of the semaphore.

5. Fifth, signals look remarkably like binary semaphores, something which causes great confusion. In fact their implementations are very similar. The fundamental difference is in how they are used, not their construction. Semaphores are generally used as mutual exclusion mechanisms, signals for synchronization.
6. Sixth, few RTOSs provide the bilateral synchronizing construct.

Semaphores may be used to create signals, but there isn't a simple one-to-one relationship. One design method employs two semaphores (one for each direction of signalling , figure 5.12) to build a single signal.

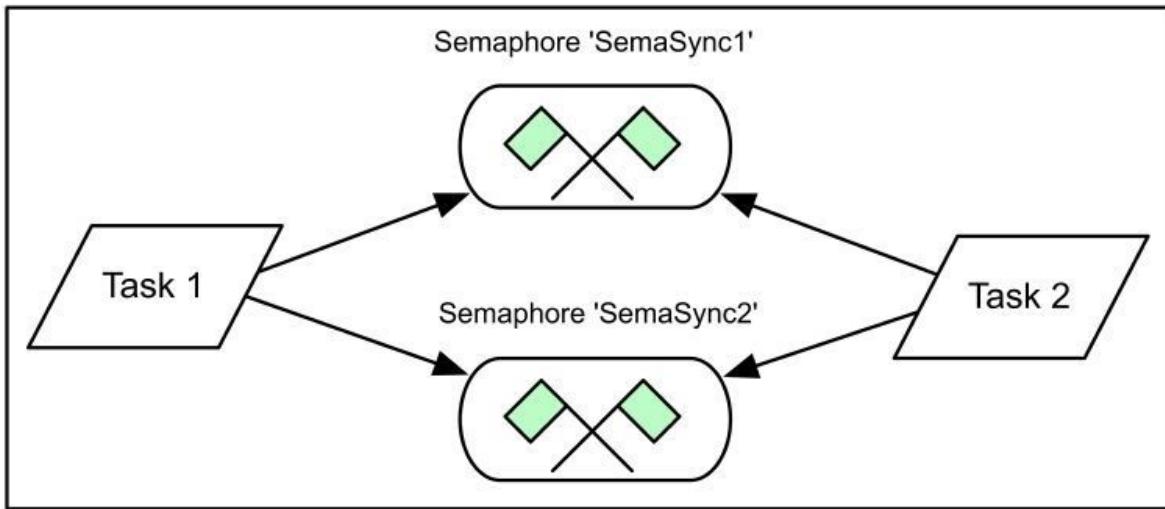


Figure 5.12 Using semaphores to mimic a signal.

The outline (pseudo) code used to implement this is shown in listing 5.8.

```
Task 1 code:
while (1)
{
Code statements;
/* The synchronization point */
Signal(SemaSync2);
Wait(SemaSync1)
Code statements;
} /* end forever loop */
```

```
Task 2 code:
while (1)
{
Code statements;
/* The synchronization point */
Signal(SemaSync1);
Wait(SemaSync2);
Code statements;
} /* end forever loop */
```

Listing 5.8

A fuller example, based on Pthreads, is given in listing 5.9. Here the equivalent to the signal Send is the Pthreads construct post.

```

/*      Simple pthread example - synchronization using semaphores      */
/* ----- */

MAIN PROGRAM UNIT
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
void * Task1(void *);
void * Task2(void *);

#define NUM_THREADS 2

pthread_t tid[NUM_THREADS];      /* array of thread IDs */
sem_t     SemaSync1, SemaSync2; /* the semaphores */

main( int argc, char *argv[] )
{
    int i;
    sem_init(&SemaSync1, 0, 0);
    sem_init(&SemaSync2, 0, 0);
    pthread_create(&tid[0], NULL, Task1, NULL);
    pthread_create(&tid[1], NULL, Task2, NULL);
    for ( i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
    /* other code statements */
} /* end main */

/* ----- TASK 1 CODE----- */
void * Task1(void * parm)
{
    /* code before synchronization */
    /* now reached the synchronization point */
    sem_post(&SemaSync2);
    sem_wait(&SemaSync1);
    /* code after synchronization */
}
/* -----end TASK 1 CODE----- */

/* ----- TASK 2 CODE----- */
TASK 2 CODE
void * Task2(void * parm)
{
    /* code before synchronization */
    /* now reached the synchronization point */
    sem_post(&SemaSync1);
    sem_wait(&SemaSync2);
    /* code after synchronization */
}
/* ----- end TASK 1 CODE----- */

```

Listing 5.9

This last example shows semaphore operations scattered across tasks. If, in practice, you choose to use this approach, then you'll end up with an accident waiting to happen. It is much better practice to use monitor-type techniques, giving conceptually the structure of figure 5.13.

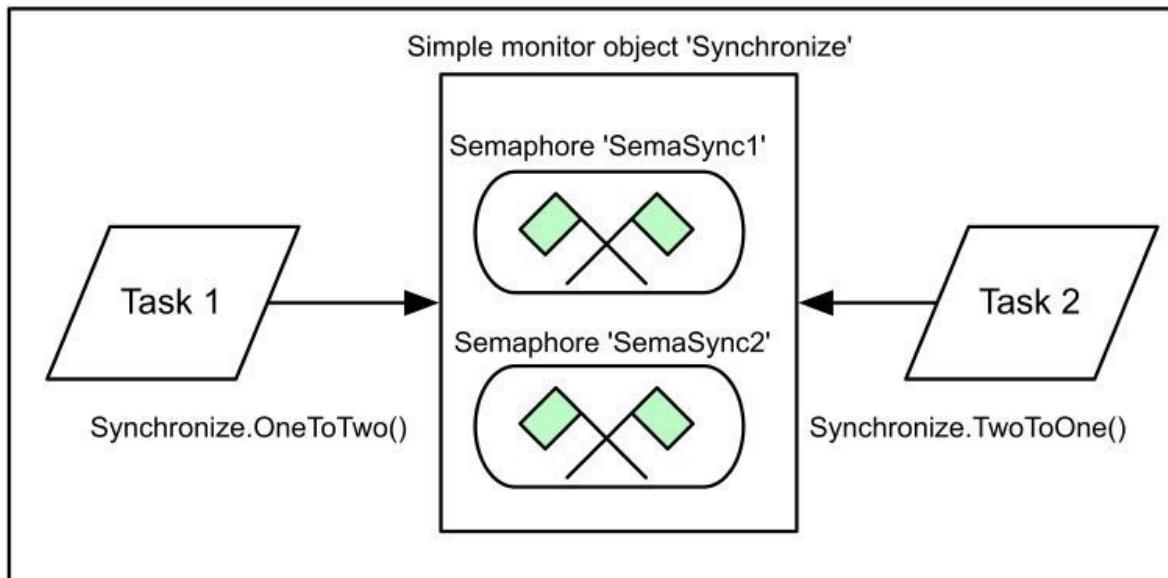


Figure 5.13 A more secure semaphore-based signal construct

5.3 Data transfer without task synchronization or coordination.

5.3.1 Overview

There are many occasions when asynchronous tasks exchange information without any need for synchronization or coordination. This requirement can be implemented by using a straightforward data store incorporating mutual exclusion features.

In practice two data storage mechanisms are used, 'Pools' and 'Queues', figure 5.14(a). Queues are also known as channels, buffers or pipes.

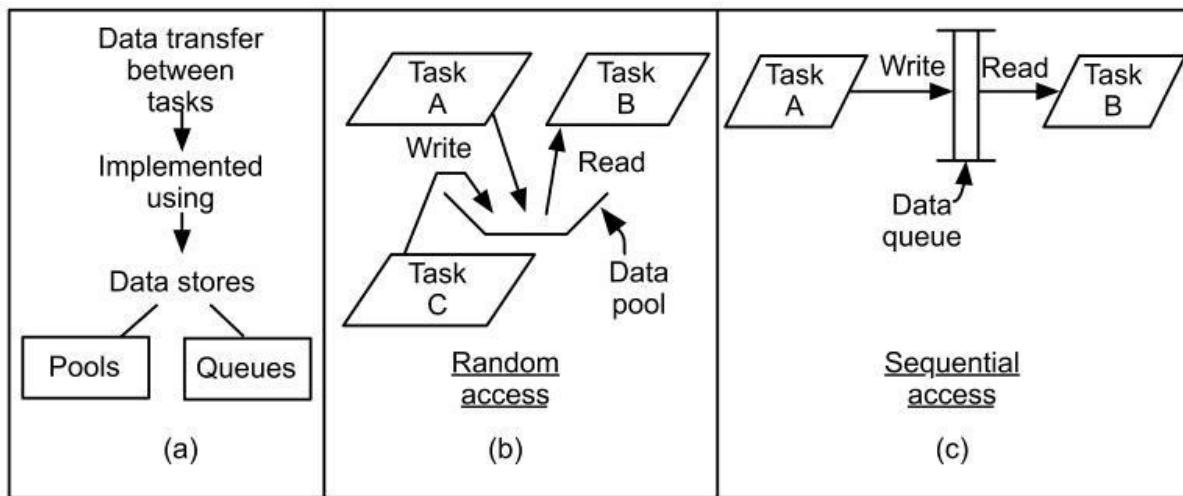


Figure 5.14 Transferring data between tasks

5.3.2 Pools

A pool, figure 5.14(b), is a read-write random access data store. Typically it is used to hold items which are common to a number of processes, such as coefficient values, system tables, alarm settings, etc. The notation used here shows tasks A and C depositing data into the pool, with B reading information out. This is not a destructive read-out, i.e. information within the pool is unchanged by the read action. The pools consist of sections of read-write memory, usually RAM (or for read-mostly operations, flash memory). Pools can be readily created using records or structures (type struct in C or C++), hence they aren't usually provided as RTOS-specific types. When you build a pool it

should, like all task comms components, be robust and secure. The store itself should be encapsulated as a private item and incorporate mutual exclusion protection (figure 5.15). Access to the pool data should be possible only via public functions provided in an access interface.

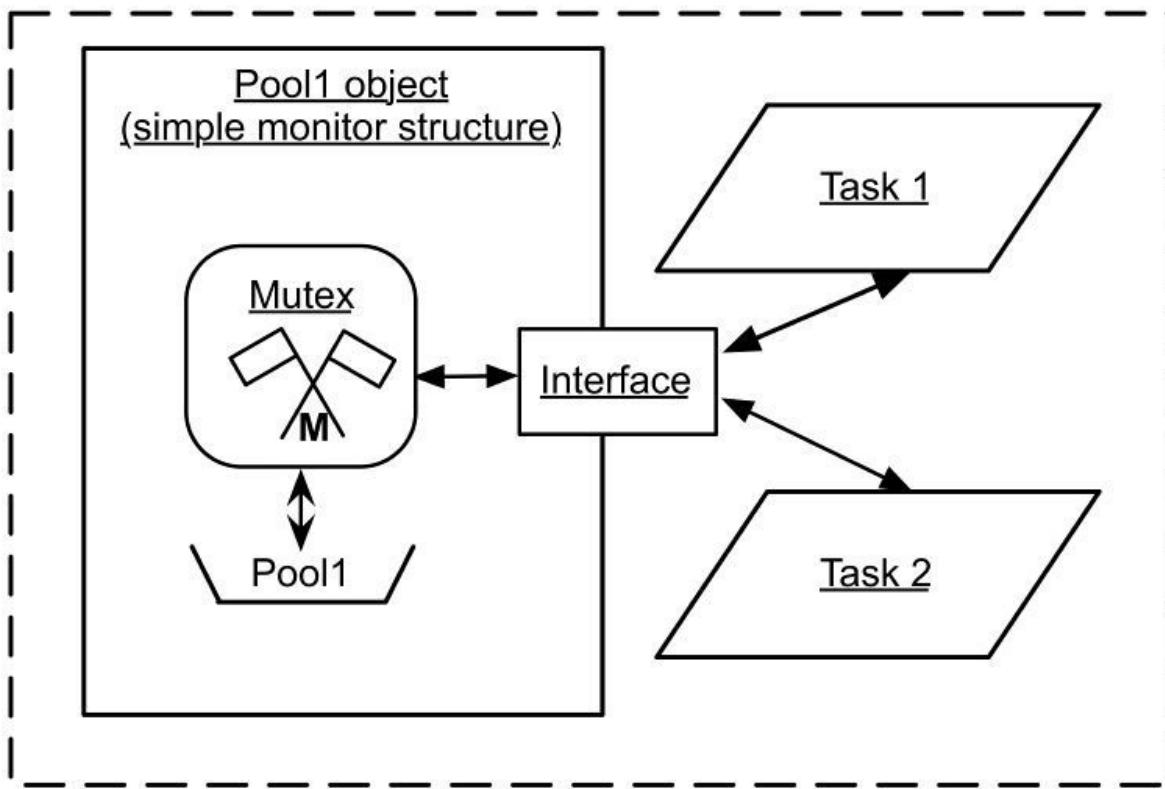


Figure 5.15 A protected pool

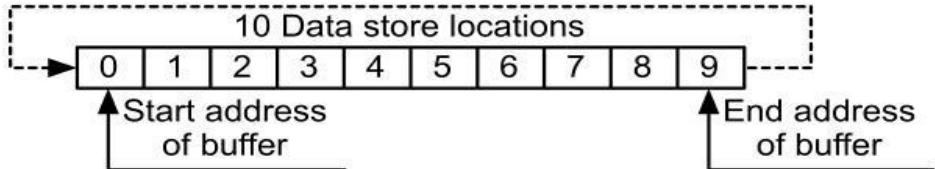
In practical systems it makes sense to use numerous pools, as and when desired. This restricts access to information, so avoiding the problems of global data.

5.3.3 Queues

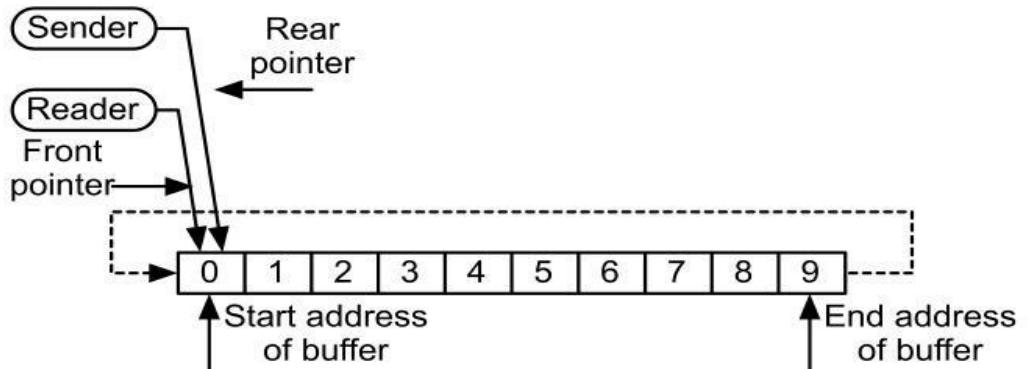
Queues are used as communication pipes between processes, normally on a one-to-one basis (figure 5.14c). Here, task A deposits information into the queue, task B extracting it in first-in first-out style. The queue is usually made large enough to carry a number of data words, not just a single data item. As such it acts as a buffer or temporary storage device, providing elasticity in the pipe. Its advantage is that insertion and extraction functions can proceed asynchronously

(as long as the pipe does not fill up). It is implemented in RAM. The information passed between processes may be the data itself; in other cases it could be a pointer to the data. Pointers are normally used for handling large amounts of data where RAM store is limited. Two techniques are used to implement queues, a linked list type structure and the circular buffer.

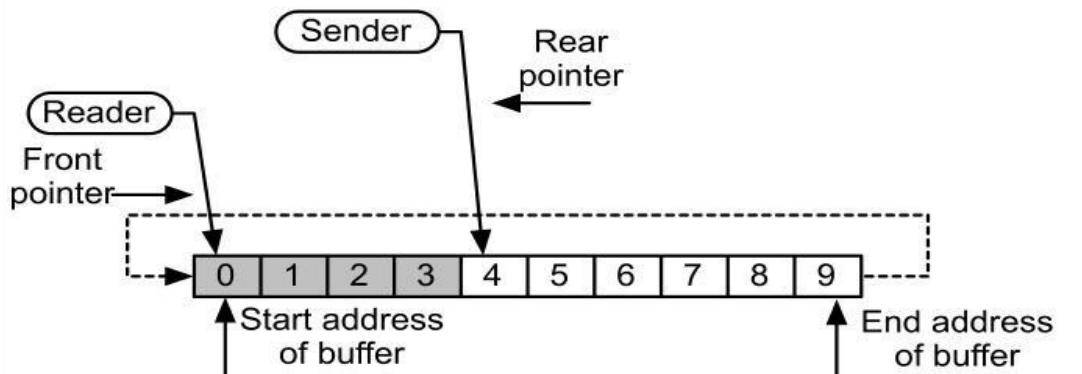
The linked list structure has been discussed under scheduling; no further detailed description is needed. A very useful feature is that its size is not necessarily fixed, but can be expanded or contracted as desired. Further, very large queues can be built, limited only by the available memory space. Yet for embedded systems these are not particular benefits. First, if RAM is limited, it is impossible to construct large queues at all. Second, large FIFO queues handling multiple messages can have a long transport delay from data-in to data-out. The resulting performance may be too slow for many real-time applications. As a result the preferred queue (channel) structure for embedded work is the circular buffer, figure 5.16.



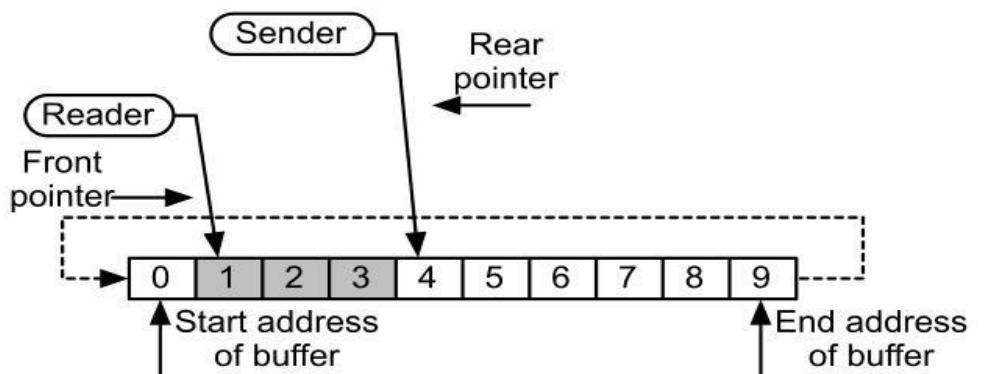
(a) Channel structure - the circular buffer



(b) Circular buffer - initial state



(c) Circular buffer - four data items inserted



(d) Circular buffer - one data item removed

Figure 5.16 The circular buffer - cyclic storage for the queue

A circular buffer is normally assembled using a fixed amount of memory space, figure 5.16(a), being designed to hold a number of data units. Buffer size is defined at creation time (here, for example, being 10 data units), but is fixed thereafter. What makes the queue circular is that data unit 0 is the successor of data unit 9; addressing is done using modulo 9 counting (just as the 12-hour clock uses modulo 12 counting).

During writing and reading operations data could be shifted through the channel. However, in general, this would impose an unacceptable time overhead; an alternative approach is used. Fig 5.16(b) shows how pointers are used to identify the start and finish locations of the stored data ('Reader' and 'Sender'). By using these we do not have to shift data through the buffer. Inserted data units always stay in the same memory locations; only the pointers change value - as demonstrated in Figs 5.16 (c) and (d). These pointers can also be used to define the 'queue full' and 'queue empty' conditions - they become equal.

Under normal circumstances tasks A and B proceed asynchronously, inserting and removing data from the queue as required. Task suspension occurs only under two conditions, queue full and queue empty. Should the queue fill up and task A tries to load another data unit, then A is suspended. Alternatively, if the queue empties and task B tries to remove a data unit, then B is suspended. In many implementations suspension does not occur; instead an error exception is raised.

There is an important difference between the pool and the queue. In the first, reading of data does not affect the contents. In the second, though, queue data is 'consumed' when read, i.e. a 'destructive' operation (this is really a conceptual view of things; in reality the reader pointer has merely moved on to the next location).

An outline of the use of the channel is given in listings 5.10 to 5.13, based on Lindentree wrapper APIs.

Basic APIs

1. *Create a queue.*

```
FOS_CreateQueue(QLength, QItemSize);
```

2. *Get message from queue.*

```
FOS_GetFromQueue(QName, AddOfQData, QwaitingTime);
```

3. *Send message to queue.*

```
FOS_SendToQueue(QName, AddOfQData, QwaitingTime);
```

Listing 5.10

Create a global queue connecting sender task A to receiver task B

```
/* Using RTOS-provided data types */
```

```
    FOS_QName    GlobalQA2B;
    FOSQLength   QA2BLength = 1;
    FOSItemSize  QA2BItemSize = 4;
```

```
    GlobalQA2B = FOS_CreateQueue (QA2BLength, QA2BItemSize);
```

Listing 5.11

Send to queue - in task A

```
/* Using RTOS-provided data types */
```

```
    long DataForQueueA2B;
    const FOSQwaitTime NoWaiting = 0;
    FOSQloadStatus QLoadState;
```

```
    QLoadState = FOS_SendToQueue (GlobalQA2B, &DataForQueueA2B, 0);
```

Listing 5.12

```
Get from queue - in task B

/* Using RTOS-provided data types */

long DataFromQueueA2B;
const FOS_QwaitTime NoWaiting = 0;
FOS_QreadStatus QreadState;

QreadState = FOS_GetFromQueue (GlobalQA2B, & DataFromQueueA2B, NoWaiting);
```

Listing 5.13

5.4 Task synchronization with data transfer

As shown earlier, situations arise where tasks not only wait for events, but also use data associated with those events. To support this we need both a synchronization mechanism and a data storage area. The structure used is the 'mailbox', this unit incorporating signals for synchronization and storage for data, figure 5.17.

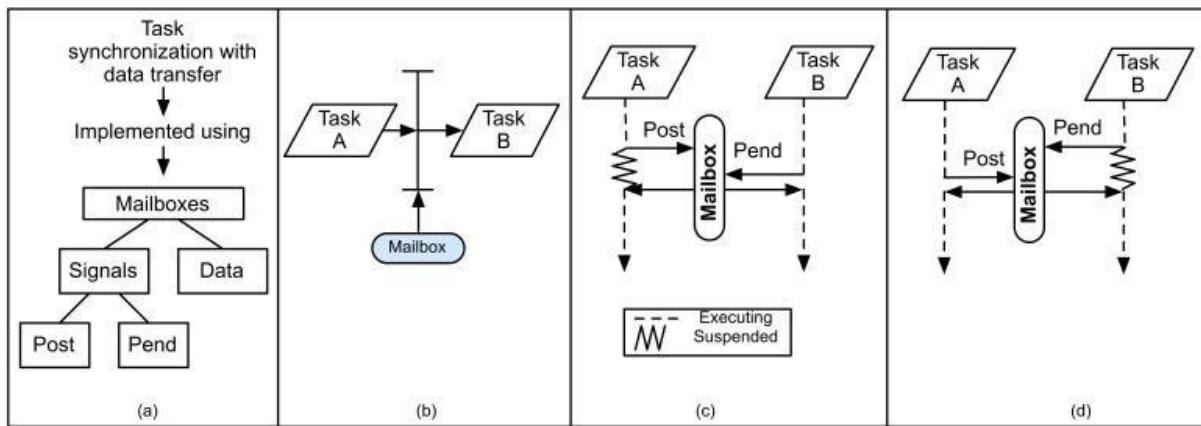


Figure 5.17 Data transfer with task synchronization

When a task wishes to send information to another one it 'posts' the data to the mailbox. Correspondingly, when a task looks for data in the mailbox it 'pends' on it. In reality, Post and Pend are signals. Moreover, the data itself is not normally passed through the mailbox; a data pointer is used. Even so, no matter how large the data contents are, the data is treated as a single unit. Thus, conceptually, we have a single-store item. Task synchronization is achieved by suspending or halting tasks until the required conditions are met, Fig 5.17 (c),(d). Any task posting to a mailbox that hasn't got a pending task gets suspended. It resumes when the receiver pends for the information. Conversely, should pending take place first, the task suspends until the post operation occurs.

Frequently the mailbox is used as a many-to-many communication pipe. This is much less secure than a one-to-one structure, and may be undesirable in critical applications.

An outline of the use of the mailbox is given in listings 5.14 to 5.17, based on MicroC/OS-II APIs.

Basic APIs

1. Create an empty mailbox.

```
OSMboxCreate ((void *) 0);
```

2. Deposit a message in a mailbox.

```
OSMboxPost (MBoxName, AddOfMessage);
```

3. Collect a message from a mailbox.

```
OSMboxPend (MBoxName, TimeOutValue, ErrorCode);
```

Listing 5.14

Create a global mailbox connecting sender task A to receiver task B

```
OS_EVENT      *GlobalMBoxA2B;
```

```
GlobalMBoxA2B = OSMboxCreate ((void *) 0);
```

Listing 5.15

Send to mailbox - in task A

```
INT8U  DataForMBoxA2B[50];  
INT8U  err;
```

```
err = OSMboxPost (GlobalMBoxA2B, (void *)  
&DataForMBoxA2B[0]);
```

Listing 5.16

```
Get from mailbox - in task B

void * DataFromMBoxA2B;
INT8U Wait200 = 200;
INT8U err;

DataFromMBoxA2B = OSMboxPend(GlobalMBoxA2B, Wait200, &err);
```

Listing 5.17

Review

You should now:

- Appreciate why tasks generally communicate and interact with each other.
- Know that such relationships fall into three categories;
 - i. Interaction without data transfers.
 - ii. Asynchronous data transfers between tasks, and
 - iii. Synchronized data transfers.
- Know what coordination flags and event flags are used for and how these relate to group flags.
- Appreciate the reason for and use of group flags.
- See how event flags support unilateral synchronization (rendezvous).
- Understand what the signal is and how it implements bilateral synchronization.
- Know how to construct a signal from semaphores or mutexes.
- Understand the reasons for having two data transfer mechanisms, the pool and the queue.
- Appreciate how pools and queues are built.
- Appreciate the differences, advantages and disadvantages of the two methods.
- Know how a circular buffer works and see why it is used to implement a data transfer channel.
- Understand the function of a mailbox.
- Know how a mailbox works and why we use it.

At this point you should complete the remaining set of exercises.

Chapter 6 Memory usage and management

The objectives of this chapter are to:

- Give a brief overview of the storage of digital information in embedded systems.
- Describe why both volatile and non-volatile storage (memory) devices are used.
- Compare the features of important embedded devices and show how these are organized in real-time embedded applications.
- Explain the difference between the conceptual and physical views of memory structures.
- Describe why memory access protection is needed in robust systems and how this is achieved.
- Explain why dynamic memory allocation is used and what problems it can lead to.
- Describe the memory features of solid-state drives.

6.1 Storing digital information in embedded systems

6.1.1 Introduction

This particular section might well be called 'things you don't really need to know but it can be very useful to do so'. So, if you thirst for knowledge, read on.

First, a statement of the obvious; all information within the processor system is held in digital electronic format (even if it originates from magnetic disk storage). Second, some information must always be retained by the system, even when power is removed; the rest we can afford to lose without degrading behaviour or performance. Put simply: if information is retained when the system is powered off, it is non-volatile. Conversely, if it's lost, then it's volatile. And note, this says nothing about what the information represents or why we have it in the first place. We'll come back to these points shortly. For the moment we need to identify the electronic technologies that enable us to store such information. Also, bear in mind that we don't need to know anything of the 'how' of these technologies; that's not relevant here.

Various technologies and devices have been developed to store information in embedded systems, figure 6.1. It can be seen that these consist of two major groups, the non-volatile and the volatile data stores.

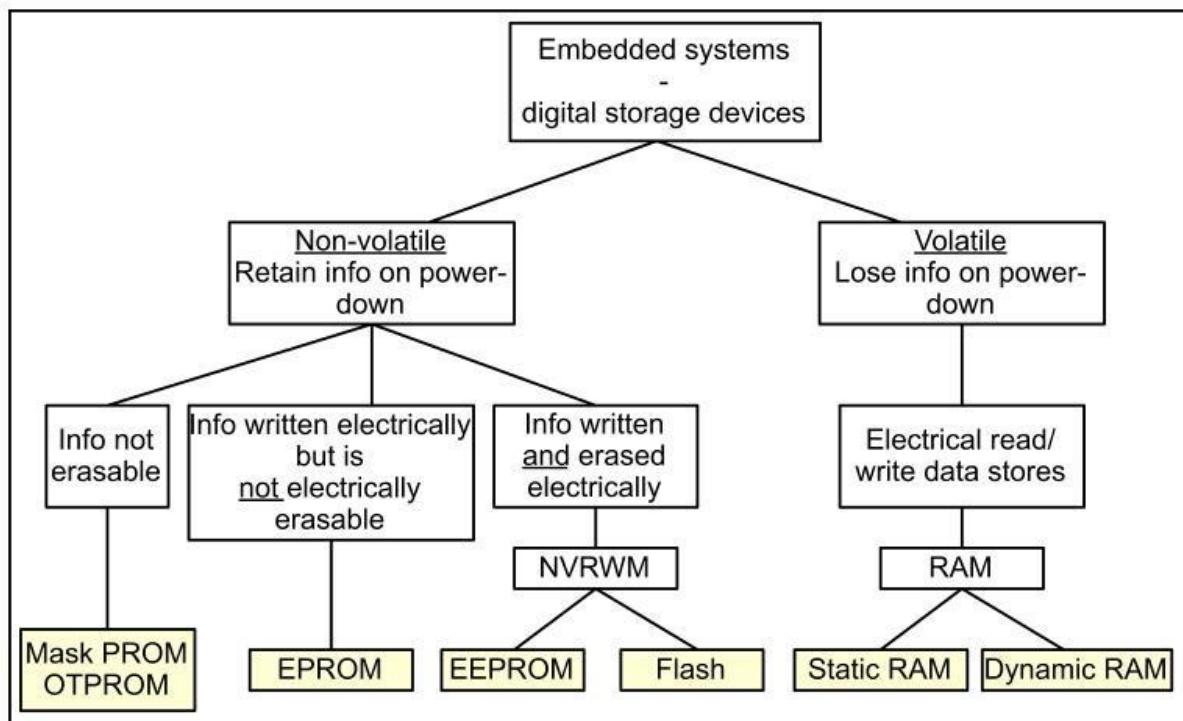


Figure 6.1 Memory devices in embedded systems

6.1.2 Non-volatile data stores

There are three categories of non-volatile store, depending on how information can be reprogrammed. The first situation is where we intend that the information, once 'loaded' (programmed), should never be changed. It is considered to be non-erasable (strictly speaking, 'erasing' means that we set all the bits in a digital store to the same value). The second is where we do intend to change the information, but not very often. The third is where we also intend to change the information but to do it simply, in-situ, under software control. However, we don't plan to make a very large number of changes during the lifetime of the storage device (a good name for this is a 'read-mostly' operation).

(a) Non-erasable storage.

There are two types of non-erasable storage devices, the mask programmable read-only memory (mask PROM) and the one-time programmable read-only memory (OTPROM). Mask PROMS, which are programmed during manufacture, are the cheapest of all provided very large numbers are used. OTPROMs are supplied in the erased state and are then programmed electrically by the user. The programmed data cannot subsequently be changed, and is retained indefinitely by the device.

(b) UV erasable storage - EPROM.

The next device to consider is the UV-erasable electrically programmable read-only memory (EPROM). Here the device contents are erased using ultra-violet light, whilst writing new information is done electrically. To change the stored information the EPROM is first erased, then re-programmed with the new material.

(c) Non-volatile read-write memories - NVRWM.

The third group are the non-volatile read-write memories, NVRWM, consisting of two technologies. The first is the electrically erasable programmable ROM (EEPROM), the second being Flash memory. With these, erasure and (re)programming is done electrically; any other differences aren't of particular interest to us at this stage. Flash memory is widely used in single-chip microcontrollers, so much so that the word Flash tends to be used as a generic term (aka Hoover for vacuum cleaner). Other devices in this category include the ferro-electric random access memory (FRAM) and the magnetoresistive RAM

(MRAM). At this time they aren't yet widely used in embedded systems.

6.1.3 Volatile data stores

In general volatile read-write data stores are called random access memories (RAMs) even if, at times, this isn't quite true. They come in two forms, static RAM (SRAM) and dynamic RAM (DRAM). As said earlier, how they work isn't relevant; the important issues here are those of cost, speed, bit-densities, power consumption and reliability of operation.

6.1.4 Memory devices - a brief Flash-RAM comparison

Here we will consider only the most widely used devices: Flash, SRAM and DRAM.

It might seem, at first sight, that Flash memory is sufficient for all our needs. After all, it is both non-volatile and is also a read-write device. Unfortunately for us it has two features that impact on its use as a general store device: speed and wear-out.

Read times are fast, in general being comparable with general purpose SRAM. Write operations, by contrast, are very much slower (a factor of 10 in some cases). The difference comes about mainly because 'writing' to Flash actually involves erasure followed by the programming of new data. If most of the memory operations are reads, then the performance of Flash-based and RAM-based designs is comparable. But once a significant number of writes is involved this is no longer true; RAM significantly outperforms Flash.

The second point mentioned earlier, device wear-out, is caused by erasure/reprogramming operation. Modern devices are guaranteed to operate correctly for (typically) somewhere between 10^5 and 10^6 operations. This has greatest impact on the use of Flash disks (RAM-disks) as a replacement for hard disks, discussed later.

It also happens that EPROMS also wear out, but in practice this isn't a problem.

6.1.5 Memory devices - a brief SRAM-DRAM comparison

A very general comparison of SRAM-DRAM features is given below in table 6.1.

Device	Memory density	Cost per bit	Speed	Power consumption	Reliability
SRAM	Lowest	Highest	Fastest	Lowest	Best
DRAM	Highest	Lowest	Slowest	Greatest	Poorest

Table 6.1 A comparison of SRAM and DRAM key features

This comparison is, by its nature, very general, so treat it as such. For instance, very high speed DRAM is faster than ordinary SRAM. But taking like for like (i.e. fast SRAM versus fast DRAM) the speed relationship shown in table 6.1 holds, this typically being in the order of 5:1.

Now let's turn to reliability. A point of clarification: here it means the probability of data bits getting corrupted, not device failure itself. With SRAM each data bit is defined by the state of a set of latched transistors. With DRAM each data bit is defined by the charge state of an on-chip capacitor. And owing to inherent charge leakage, DRAMs have to be regularly refreshed. The weakness of this storage method is that bit flips (caused usually by electromagnetic interference (EMI)) are much more likely to occur in DRAMs.

If bit flipping can lead to serious failures extra precautions must be taken with the use of the devices. This usually involves storing extra check bits with the data bits and verifying correctness using error-checking codes. For mundane applications a single check bit is added to each byte, errors being flagged up by error detecting codes (no attempt is made to correct errors). However, for highly critical applications (e.g. deep space vehicles) a number of check bits are appended to each data word to allow error detection and correction to be carried out.

6.1.6 Embedded systems - memory device organization

Underlying all decisions concerning the selection and organization of memory devices in embedded systems is one basic question; what information must be held at all times? Remember, when embedded systems are switched on we expect them to automatically start up and then work correctly. But this will happen only if the program code and all essential data are present, even during power-down conditions. Which means that such code and essential data (both constants and variables) must be held in non-volatile storage.

The code issues are clear-cut but what of the data, in particular the variable data (i.e. that which changes during system operation). Precisely what should be saved when power goes off? Well, let's look at some examples to answer this question.

First, consider a personal weather station whose measurements include time, temperature, pressure, etc. In this application, when the unit is powered off, is there any need to retain the current data? Very unlikely; the data is not especially important and hence can be stored in RAM.

Second, take a case of a car radio that allows the user to allocate radio stations to specific selector buttons. This is not a once-and-for-all allocation; the user can change settings at will. However, each time the radio is switched back on the settings should be as they were at switch-off time. Clearly it is essential to retain such settings on power-down, which means they must be saved in non-volatile storage (Flash).

A third example: consider where we record the state of charge of a submarine propulsion battery. This is a situation where measurements and calculations have to be made at quite short intervals (e.g. a few seconds) and for extended periods of continuous operation (months, perhaps). And it's essential that we don't lose this information if processor power is lost. Once again there is a need for non-volatile storage of data. But, the storage requirements are quite different to those of the previous example.

For the radio, the number of setting changes made during the lifetime of the vehicle will be relatively few. As a result the number of data re-writes will also be few. In this case it is perfectly safe to store the setting data in Flash memory. By contrast the battery monitoring system will be constantly updating information; extensive re-writing will take place. And the problem here is that if data is directly stored in Flash, then wear-out could well occur (simple calculations will show if this is likely). The solution to this is to use a combination of RAM and Flash; log the on-going data to RAM and then transfer it to Flash when power-down is detected.

At this point we have identified what should be held where; now let us now look at the organization of memory devices in practical systems (please note: these examples are representative, not comprehensive).

(a) Single-chip microcontrollers for general-purpose use.

Microcontrollers, by definition, have all memory on-chip. Most modern devices

come with various combinations of Flash and static RAM.

(b) Single-chip microcontrollers for high-speed applications.

These also have on-chip Flash and static RAM together with fast external SRAM. During processor initialization the stored information is loaded into the external RAM; after this all program execution is RAM-based.

(c) Microcontrollers for complex memory-intensive applications.

DRAM comes into its own where applications need a great deal of memory. But where high performance is also required, the relatively slow speed of dynamic RAM is a drawback. To alleviate this, an additional storage area of fast SRAM, a cache, is also used, figure 6.2.

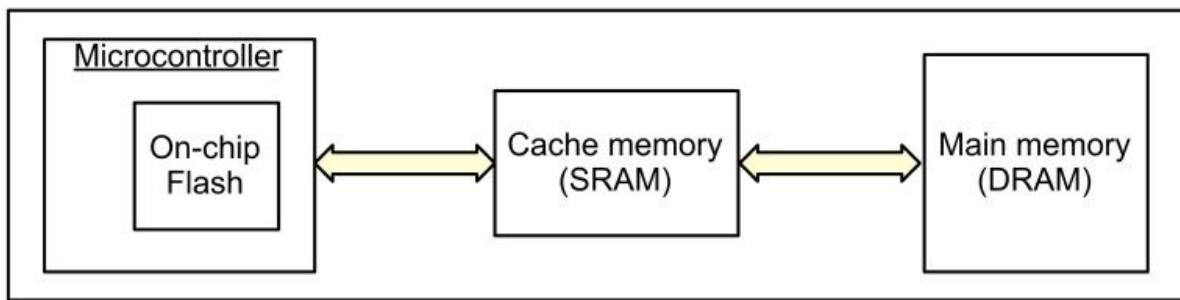


Figure 6.2 Cache memory usage

The idea, quite simply, is to speed things up. The basis for this is that programs often access the same memory information time and time again. A copy of such information (data and/or instructions) is loaded into the cache, for use by the processor. Thus processor-memory interactions take place at a rate determined by the high-speed SRAM, not the slower DRAM, devices. This is especially effective when the program itself is housed in main memory or when data is obtained from a disk store. With many micros the cache is integrated on-chip, referred to as a L1 cache.

Please realize that this is a superficial look at a complex subject. However, it is sufficient to give a good insight into knowing how to use your memory in an embedded system.

6.2 Memory aspects - conceptual and physical.

Let us start by looking at how memory devices relate to the basic elements of a real-time operating system, figure 6.3.

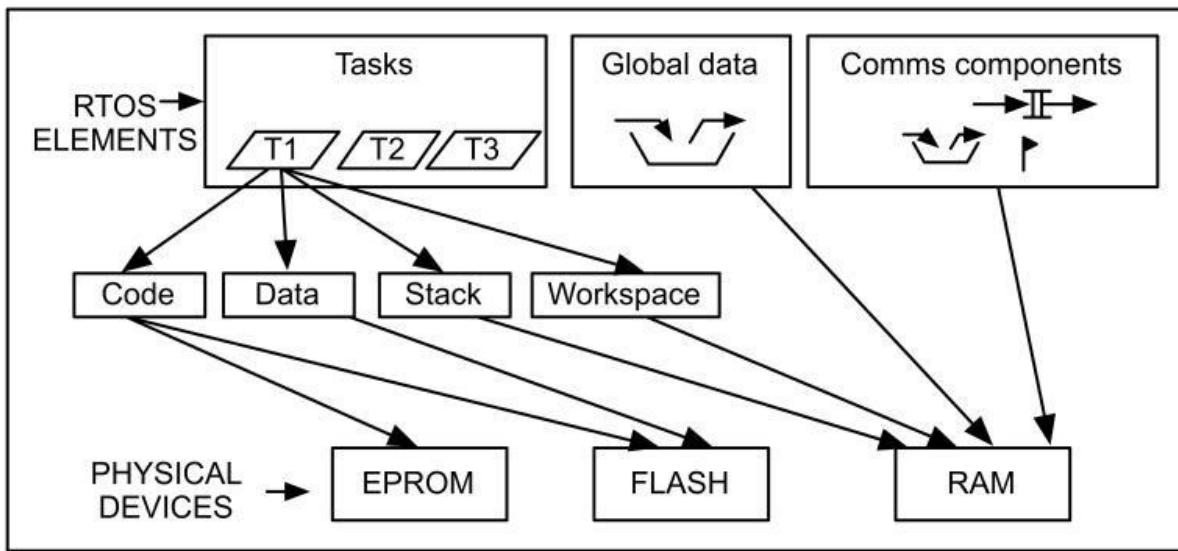


Figure 6.3 Typical memory usage in a multitasking system

The major items are tasks, communication components and global data. Strictly speaking you could argue that global data is a communications component; it does provide a means for sections of software to 'speak' to each other. However, it is not normally viewed as being a comms component per se (note: in this context 'comms' means 'task comms'). Even so, the most appropriate model for global data is a pool.

The elements making up a task are its code, data, stack and general working area, the 'heap'. Code, as mentioned before, is normally held in EPROM or Flash memory. Each task may have its own individual stack; alternately a single runtime stack may be shared amongst all tasks. The advantage of the shared stack method is that it needs less RAM space than the individual stack method. This is particularly useful in 'resource constrained systems', e.g. those with only small amounts of RAM. However, the performance of shared stack systems is inferior, especially under high task switching (context switching) rates.

Now consider the conceptual processor-oriented view of a multitasking system given in figure 6.4.

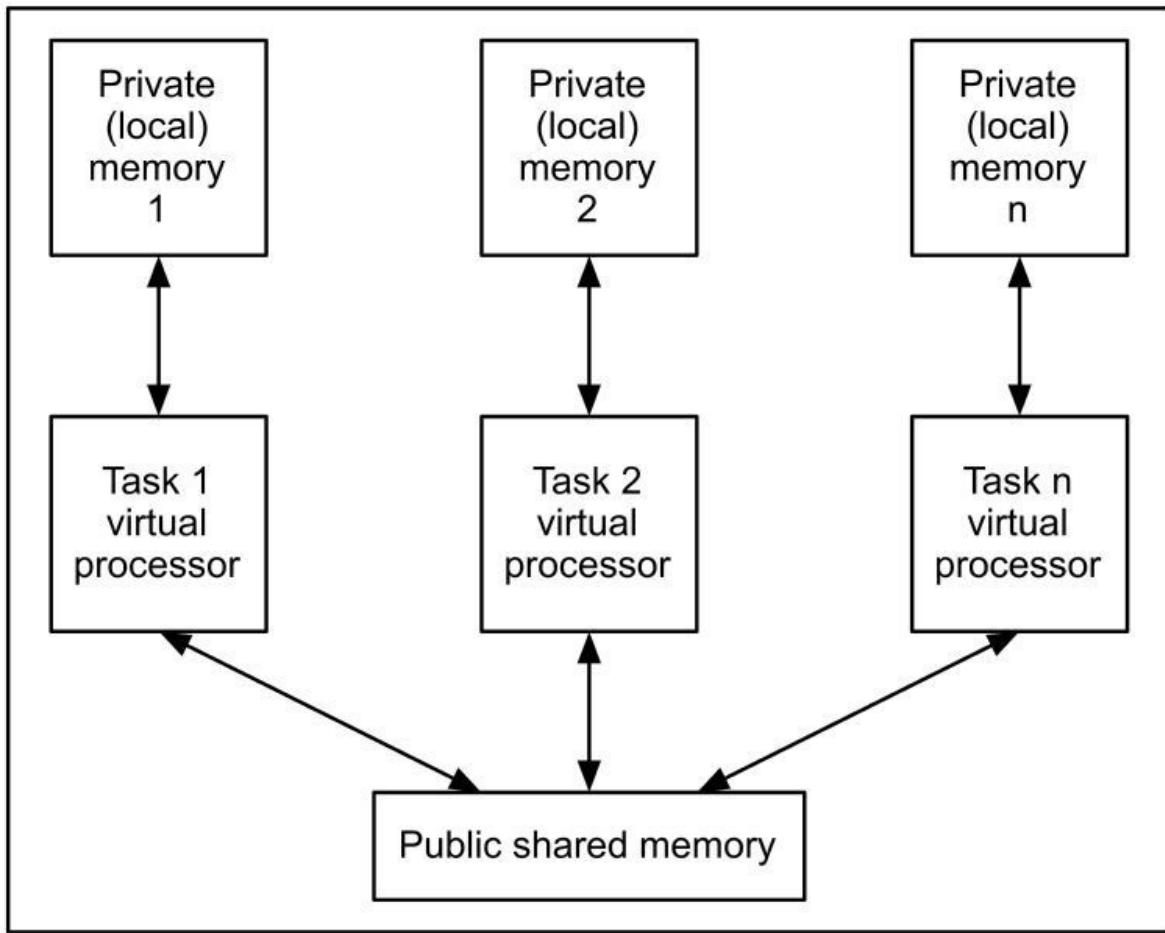


Figure 6.4 Conceptual view of a multitasking structure

In this model we pretend that each task has its own processor, the task 'virtual' processor. In its simplest form a virtual processor consists of the 'register set' in the task process descriptor. It becomes an actual processor when a task is switched into context, interacting with both private and shared memory areas of ROM and RAM. Private memory houses information specific to a task, including both data and code items. Other tasks have no need to access this material. Shared memory is generally used for global data, comms components and shared code.

It is important to be able to reconcile these two views in the context of memory usage and devices, figure 6.5.

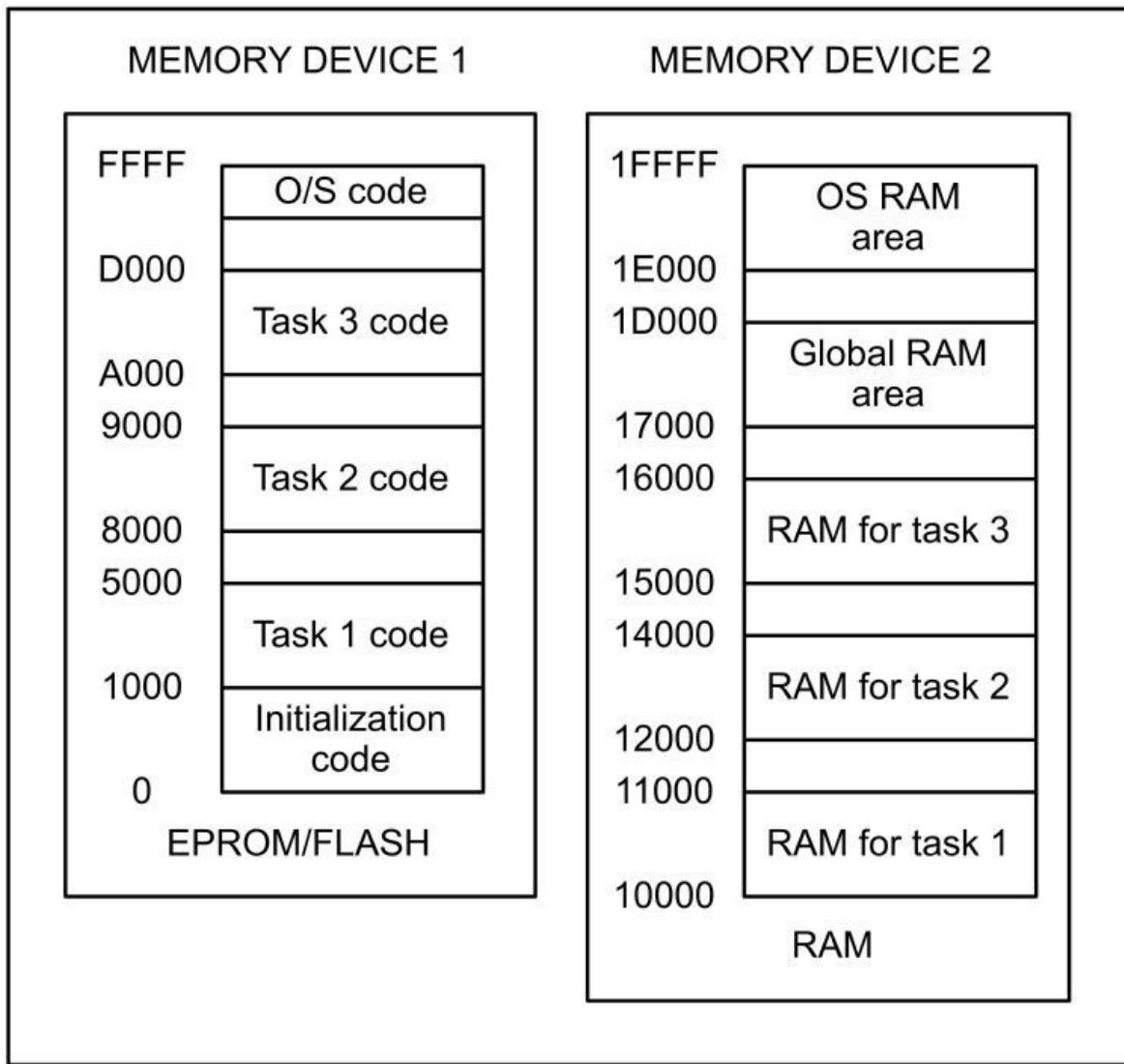


Figure 6.5 Example of physical memory structure- small system

This memory structure is typical of those found in smaller systems. It consists of two 64 kbyte memory chips, one EPROM (or Flash) and one RAM. Observe that these devices are mapped into specific positions in the processor memory address space. Each memory address is unique - device addresses cannot overlap. The RTOS elements shown earlier are mapped (located) at specific locations within the address space of each device. These comments also apply to single-chip microcontrollers; it's just that the Flash and RAM memories are housed on-chip and not on external devices.

6.3 Eliminating inter-task interference

6.3.1 Controlling memory accesses - a simple approach

A number of significant points can be deduced from the detail of figure 6.5. The first is possibly one of the most important. With this physical structure, the distinction between private and shared (public) memory is merely conceptual. There is absolutely no reason why task 1, for instance, cannot access the private memory of tasks 2 and 3. Perhaps even worse it could interfere with the operating system itself. This points up the need for some form of protection mechanism. As a result a number of processors provide a 'firewall' to allow (at the very least) separation between the operating system and the application tasks (e.g. protected modes of operation). Unfortunately this still leaves us with the danger that tasks may interfere with each other. One simple technique to counter this is shown in figure 6.6. When a task is dispatched, the memory address bounds are loaded into two registers. Each memory address produced by the processor is compared with these bounds; any violation leads to the generation of an error signal (usually an interrupt).

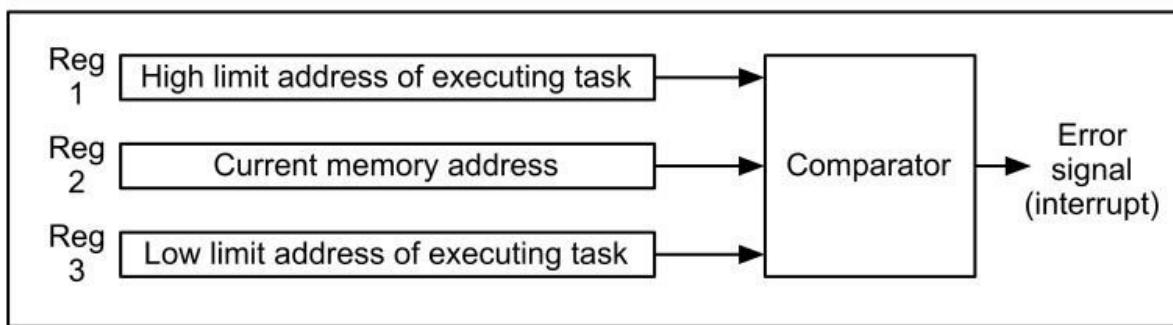


Figure 6.6 Preventing task interference - a simple protection scheme

While the idea isn't complex, it does have some drawbacks. The first (and fairly obvious) one is that it takes time to load the registers and to carry out comparisons. There is also the difficulty of getting the bounds information in the first place (this will depend on the memory allocation method being used). Fortunately we can minimize the timing problems by using hardware to provide protection.

A minor practical point: address information is usually produced using a low-limit address value (the base address) together with the size of the memory being used (the address range).

6.3.2 Controlling memory accesses with a memory protection unit

A hardware device specifically designed to prevent tasks making invalid memory accesses is the memory protection unit (MPU), figure 6.7. It is programmed to hold task address information, specifically the address limits.

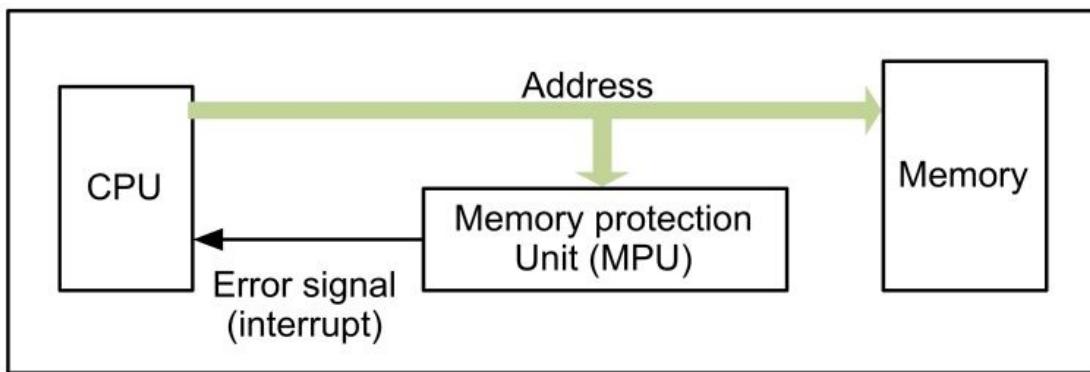


Figure 6.7 Preventing task interference - hardware (MPU) protection

The MPU monitors the address information flowing between the CPU and the processor memory; any violation of address bounds causes an error signal (exception) to be produced.

The MPU can be looked at from two points of view: processor level and task level, figure 6.8.

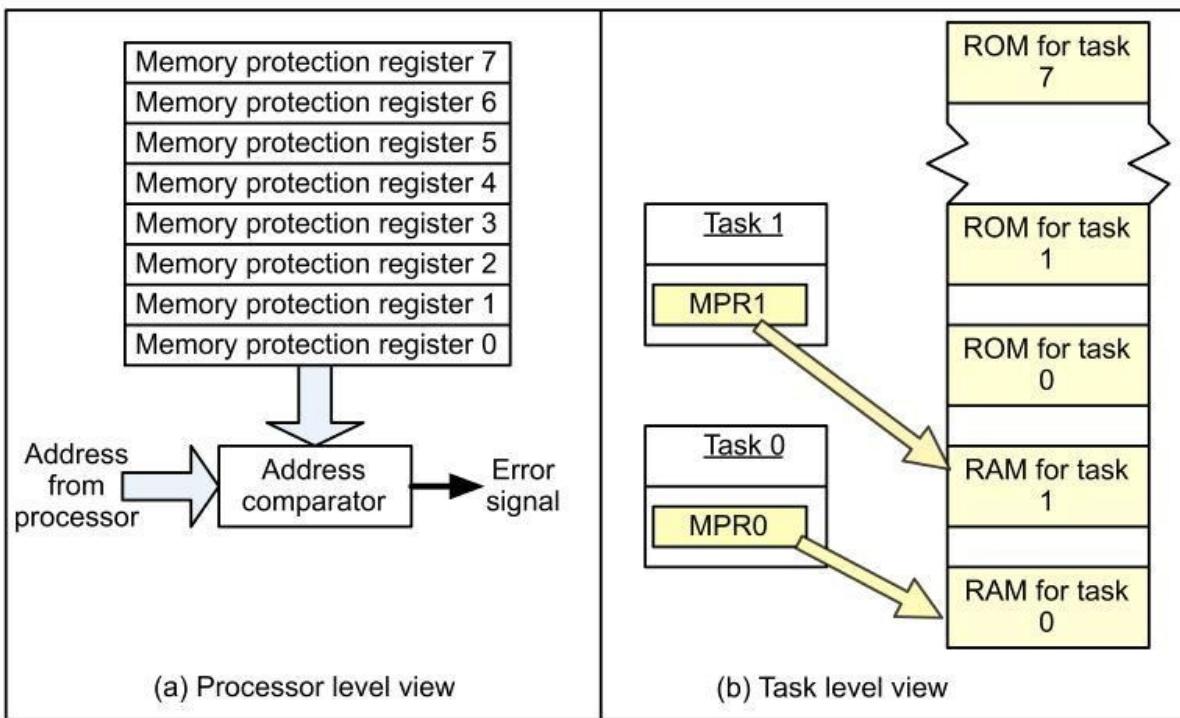


Figure 6.8 MPUs and their protection registers

Central to the protection scheme is a set of memory protection registers (MPRs). The example here has eight registers, each one holding the memory address bounds of a task (base address plus associated range size). At task creation time each protection register is configured with the appropriate memory information. During task execution each address generated by the processor is compared with those held by its MPR; any attempt to access memory outside of the predefined limits raises an exception. In this example protection is provided for RAM areas only; some MPUs also have code protection registers.

It is recommended that, where possible, hardware designs intended to support multitasking operation should include MPU facilities (many processors now provide an MPU as part of the on-chip circuitry, as for example the ARM1156). However, it isn't by itself capable of working out address bounds; these still have to be obtained from program (or compiler) information.

6.3.3 Controlling memory accesses with a memory management unit

Most embedded systems are designed to be used in very specific applications, e.g. robot controllers, building management, ship steering, etc. However there are exceptions to this, systems intended to support multiple software

applications. Examples include multimedia devices, gaming machines, infotainment systems and the like. Usually the software (code and data) for each application is held on a backing store, being loaded into main memory when required. Now this leads to a problem concerning the memory addresses used when the programs are compiled.

Applications like these frequently run on micros that have different memory sizes and configurations. Information about the target memory structure, the physical memory, isn't known at compile time. What we can pretend, though, is that the target has memory which matches its address space (note: a 32-bit processor can address 2³² store locations). Programmers can then decide what addresses they wish to use for any particular application. These are called the 'logical' addresses, and are the ones seen by the processor. However, these logical values cannot be applied directly to the real memory; they must first be mapped into the memory address space, the 'physical' addresses. And that is the primary role of the memory management unit (MMU), figure 6.9.

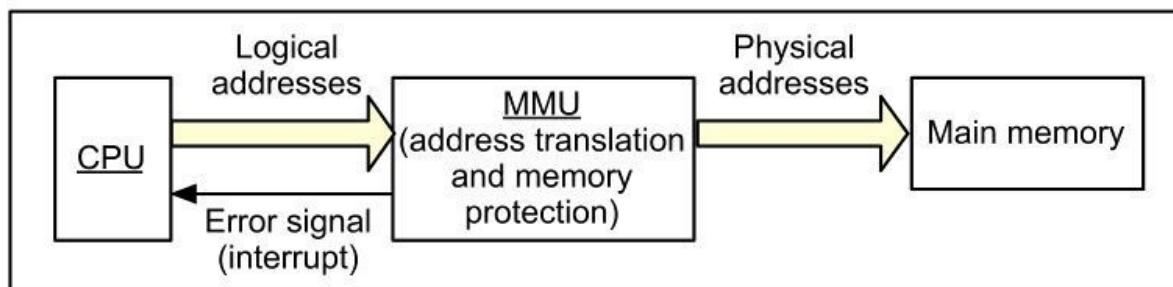


Figure 6.9 The role of the MMU

The MMU contains a set of translation registers that are used to convert the logical addresses (those generated by the CPU) to the physical ones (those seen by the memory). These registers are re-loaded with translation data when the new application software is loaded into main memory.

A secondary role of the MMU is that of memory protection. It executes this function in exactly the way as that of the MPU described earlier.

It is important to realize that all operations carried out on and by the MMU (and also the MPU) are 'invisible' to the programmer. Memory bounds, address translation, access rights, etc. are all dealt with by the operating system. Such information is generated during the compilation process, being derived from task creation, linkage and location information.

6.4 Dynamic memory allocation and its problems

6.4.1 Memory allocation and fragmentation

The problems of dynamic memory allocation apply mainly to RAM (although similar problems may arise with the use of Flash memory). Moreover, these issues may also apply - though less critically - to designs that do not use multitasking.

As pointed out earlier, tasks require RAM space for a variety of reasons. The most elementary problem is how to provide this space. A simple, effective and clean technique is to define - as part of the task creation - the amount and location of RAM provided for each task. This allocation can be explicit; that is, the programmer defines the details in source code. Alternatively such decisions can be left to the compiler. Either way, the allocation is a static one; once made it remains unchanged during the lifetime of the task. Thus at the beginning of a program in a small system, the RAM usage could be as shown in figure 6.10(a). Observe that we have 'chunks' of allocated memory.

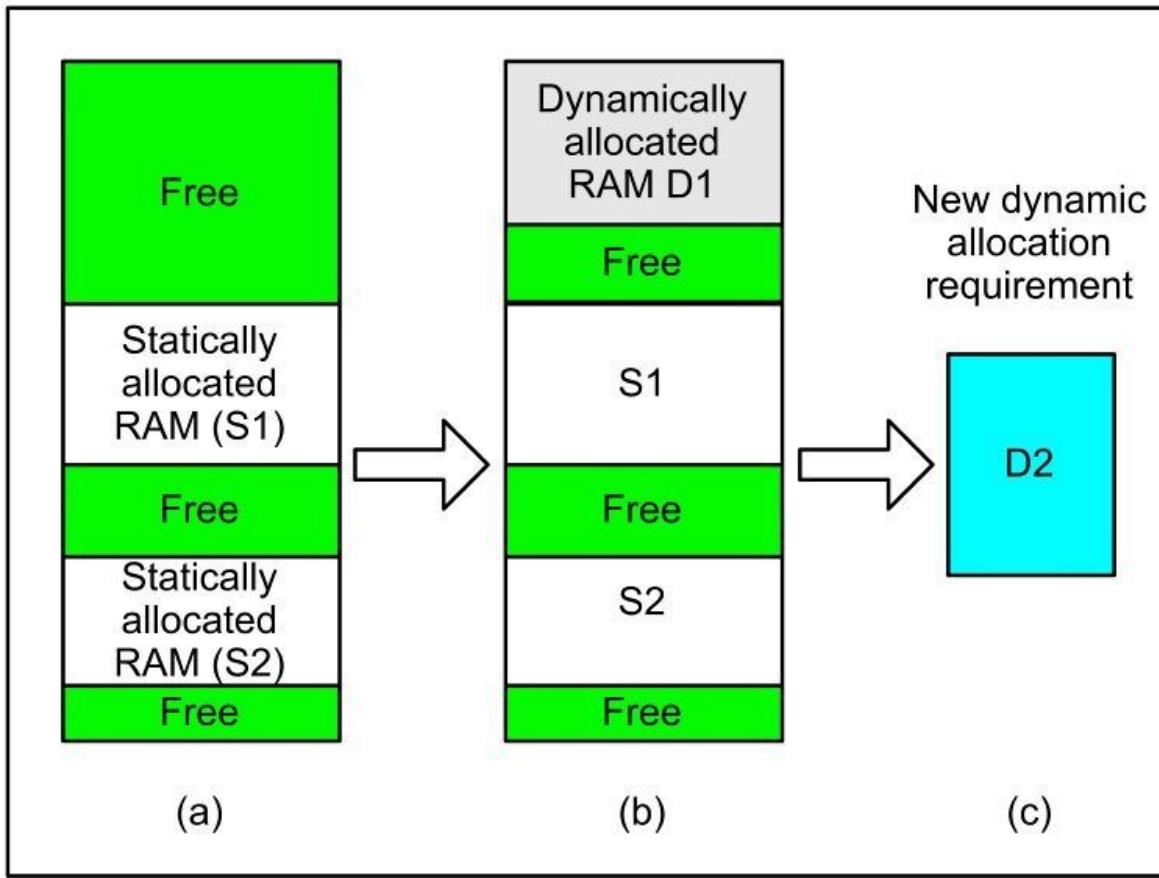


Figure 6.10 Memory allocation, fragmentation and its consequences

In many small systems the allocation would never change (of course the amount of RAM actually in use will vary). Unfortunately there are situations where this approach causes difficulties. Consider, for instance, a data monitoring system that periodically (say once per minute) measures, processes and displays system information. This operation may use a large amount of RAM store; yet it is needed only for a short period of time. It is wasteful to permanently tie up a resource that is used only intermittently. A much more efficient way is for the system to allocate RAM only when it is needed; on completion of the required activities it is retrieved for future use ('deallocated'). That is, allocation and deallocation is dynamic in operation, figure 6.10(b).

All is fine with this strategy as long as the chunks of free RAM space are large enough to meet our needs. Unfortunately, if memory allocation and deallocation isn't carefully controlled, the system will almost certainly fail. Some day a task will request an allocation of memory greater than any of the free areas, figure 6.10(c). It might take months of operation before the problem surfaces, but when

it does the failure can be deadly. This can happen even when the total amount of free memory is sufficient to meet our needs.

The problem has arisen because the allocated memory is not nicely, compactly, organised; instead it is fragmented throughout the available memory space. Such a situation can arise in C programming, for example, by the use of the standard functions malloc() and free(). This points up the need to have a well defined, organized and controlled memory allocation techniques. Slow or soft systems don't present a major problem; there is usually plenty of time to rearrange (shuffle) existing allocations under the control of a background task. By contrast, hard fast implementations are much more challenging; techniques must be used that prevent memory fragmentation in the first place.

6.4.2 Memory allocation and leakage

Now let us look at a second problem, that of memory leakage, figure 6.11.

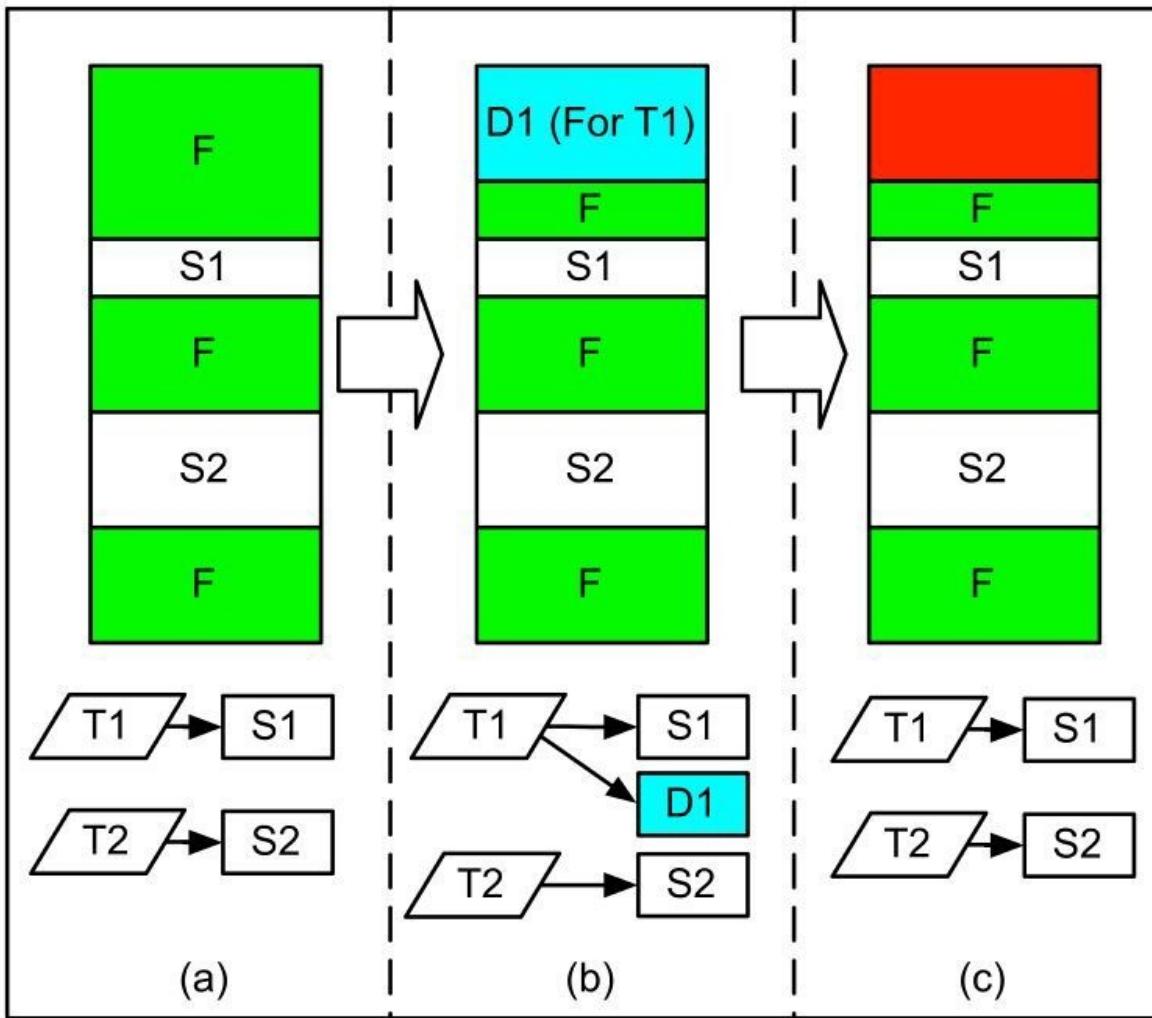


Figure 6.11 Memory leakage problem

Figure 6.11(a) shows a situation where a static memory allocation has been made for two tasks, T1 and T2. Task 1 uses memory area S1, task 2 using area S2. Let's assume that at some later stage, figure 6.11(b), task 1 invokes a function which calls malloc(), to dynamically acquire extra memory, D1. Unfortunately, owing to a programmer mistake, the function terminates without calling free(); thus it doesn't deallocate D1. As a result the amount of free memory is now permanently reduced for the duration of the program. We have 'leaked' memory from the available pool.

A single leak is unlikely to be a problem; the same cannot be said for multiple occurrences.

Leakage is not merely a tasking issue; it applies anytime where memory can be

dynamically allocated and deallocated. In particular it is a well-known source of angst in C and C++ programs resulting from 'dangling pointers'. There are two basic techniques to deal with this; OS activated (implicit) or programmer controlled (explicit).

When the OS retrieves allocated but unreferenced memory it is said to perform 'garbage collection'. It carries this out with the aid of a task called the garbage collector. This has the job of keeping track of all the memory items (objects) and references to such items. If it sees that an object is no longer referenced, it retrieves the memory space for future use. While this method is safe, secure and simple to use, it has one major drawback. The garbage collector task itself introduces uncertainty into the timing performance of the system. In designs where the garbage is collected in one go, the effect can be substantial. Such methods are totally unacceptable for hard/fast systems. The problem can be minimized by collecting the garbage incrementally - 'a bit at a time'.

Explicit memory management has been a standard programming feature for many years. The problems associated with allocating and then failing to deallocate memory (the dangling pointer) are well known. It is strongly recommended that - even if performance suffers somewhat - you take a safe and secure approach in your programs. One proven, practical technique is to encapsulate matching allocation and deallocation operations within a subprogram. Failing that, check your programs using a run-time analysis tool designed to catch memory leaks.

A footnote: Dynamic memory manipulation should be avoided wherever possible in real-time systems. It should never be used in critical applications.

6.4.3 Secure memory allocation

The key to providing secure memory allocation is to make all operations deterministic. First, find out exactly how much RAM you have in your embedded system. Second, decide what memory, and how much of it, you will use for dynamic operations. An example is shown in figure 6.12, typical of that found in single-chip microcontrollers. Here the volatile store of the micro is 64 kbyte of RAM, mapped into the processor address space as shown. We have decided to use 32 kbyte for dynamic allocations. Control of allocation and deallocation of memory from this area is handled by an RTOS-provided memory manager.

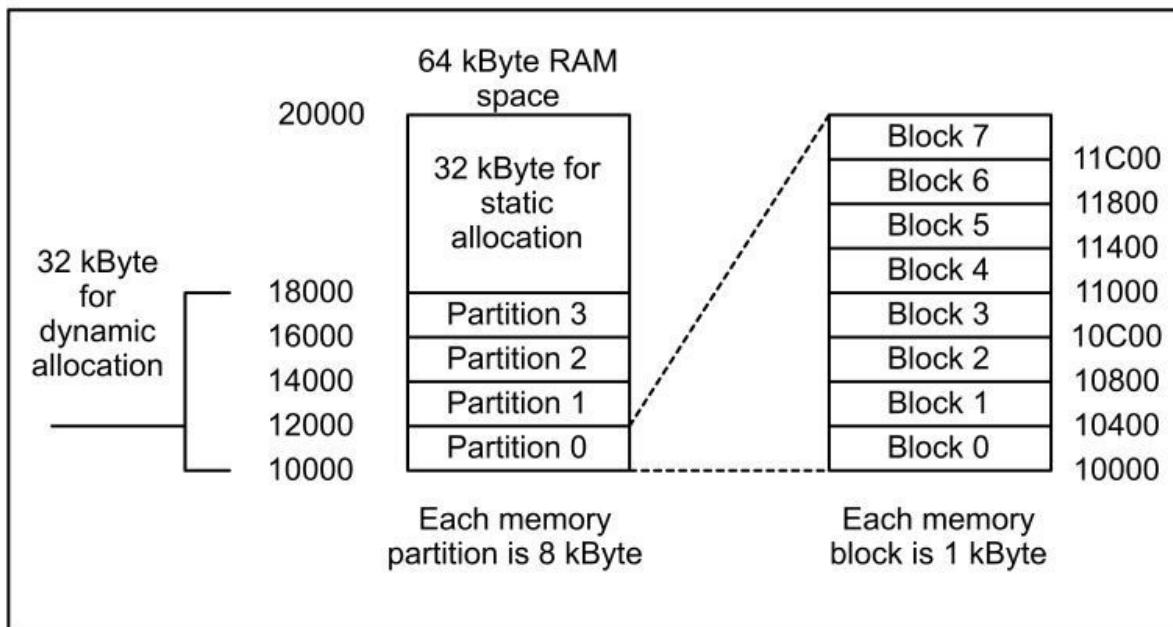


Figure 6.12 Secure memory allocation

The 32 kbyte is split into four 8 kbyte sections or 'partitions' (also called 'pools'). Note that these are located at fixed positions in the memory map. Each partition is, in turn, formed as eight 1 kbyte 'blocks', all blocks being located at fixed addresses. As shown here the rules for memory allocation and deallocation are simple and straightforward.

- Memory is allocated from selected partitions.
- Only one block is allocated for each request made.
- A count is held of the number of blocks currently allocated.
- Deallocated memory is always returned to the partition it came from in the first case.
- Only one block is returned for each return request made.
- Allocation/deallocation times are deterministic.
- Errors are flagged up if an allocation request is made to an empty partition or if a return is attempted to a full partition.

An example of allocation/deallocation in action is shown in figure 6.13, which should generally be self-explanatory. One point that might be puzzling initially is that the returned block (block 0) occupies a new position; it is inserted so that the available memory forms a contiguous block. However, this is essential to prevent fragmentation.

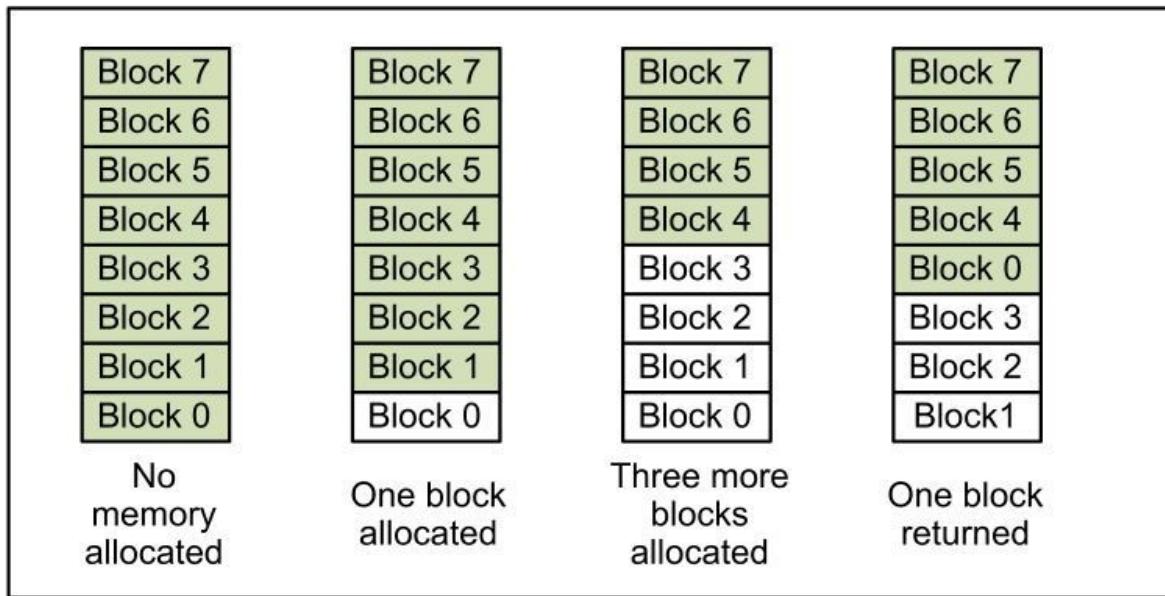


Figure 6.13 Memory block allocation and deallocation

To make this work in practice we use what is generally called a 'memory control block', figure 6.14. A better name is 'partition control block' (as used here the word 'block' is a general term meaning data structure, cf. task control block)

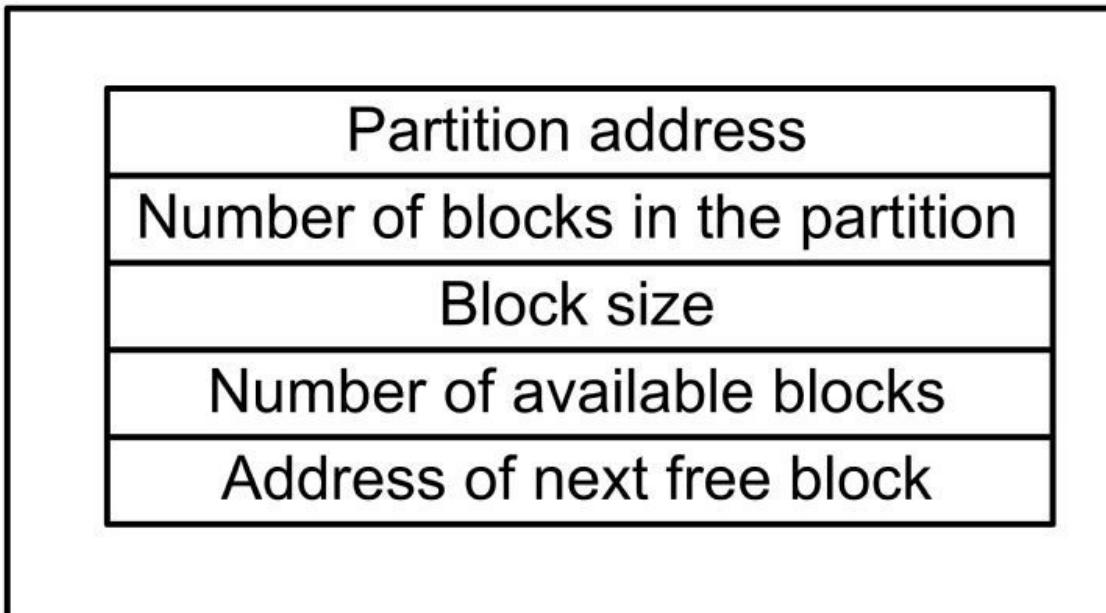


Figure 6.14 The memory (partition) control block

Each partition has its own control block, being structured as a number of data fields. Here there are five fields, these holding the information defined in figure 6.14. Key to the reordering of blocks is the use of address (pointer) techniques to create a linked list of memory locations. This makes it quite easy to reorder the logical arrangement of the physical memory blocks.

The following listing 6.1 is a pseudo-code example for the handling of dynamic memory. This creates a memory partition that consists of eight blocks, each block having space for 1024 integers. The address variable MachineryLog points to the start of the partition.

```
/* ===== Create a memory partition ===== */
int NoOfBlocks = 8;
int SizeOfBlock = 1024;
int PartitionAddress = 0x10000;

/* Creating an address (pointer) variable of RTOS-defined type */
RTOS_MEM    MachineryLog;

/* Defining the memory partition and its size      */
int    MachineryLogPartition [NoOfBlocks][SizeOfBlock];

{
    MachineryLog = RTOS_CreateMemBlock (MachineryLogPartition, NoOfBlocks,
                                         SizeOfBlock, PartitionAddress);

}
/* ===== */
```

Listing 6.1

Now to get a block from the partition, listing 6.2:

```
/* ===== Returning a memory block ===== */
RTOS_MemPut (MachineryLog, MachineryData);
/* ===== */
```

Listing 6.2

After the memory has been allocated the variable MachineryLog points to the start of the second block in the partition. A count of (current) allocations is held within the memory control block.

Although we have the concept of a block being 'removed' from a partition, things are very different at the physical level. The result of 'getting' this block (the 1 kbyte memory space starting at address 0x1000) is that it becomes accessible to the application software. That is, data can now be written to and read from it using the pointer MachineryData.

Finally, when the memory block is no longer needed, it must be returned to its partition, listing 6.3.

```
/* ===== Returning a memory block ===== */
RTOS_MemPut (MachineryLog, MachineryData);
/* ===== */
```

Listing 6.3

6.5 Memory management and solid-state drives.

There are two major categories of solid-state drives. First there are the removable media, such as USB memory sticks and secure digital (SD) camera cards. Then there are the fixed media, such as large Flash-based stores. Many systems, in fact, use Flash memory to mimic a mechanical disk, the solid-state drive. This belongs to the 'Ramdisk' family, having the structure shown in figure 6.15.

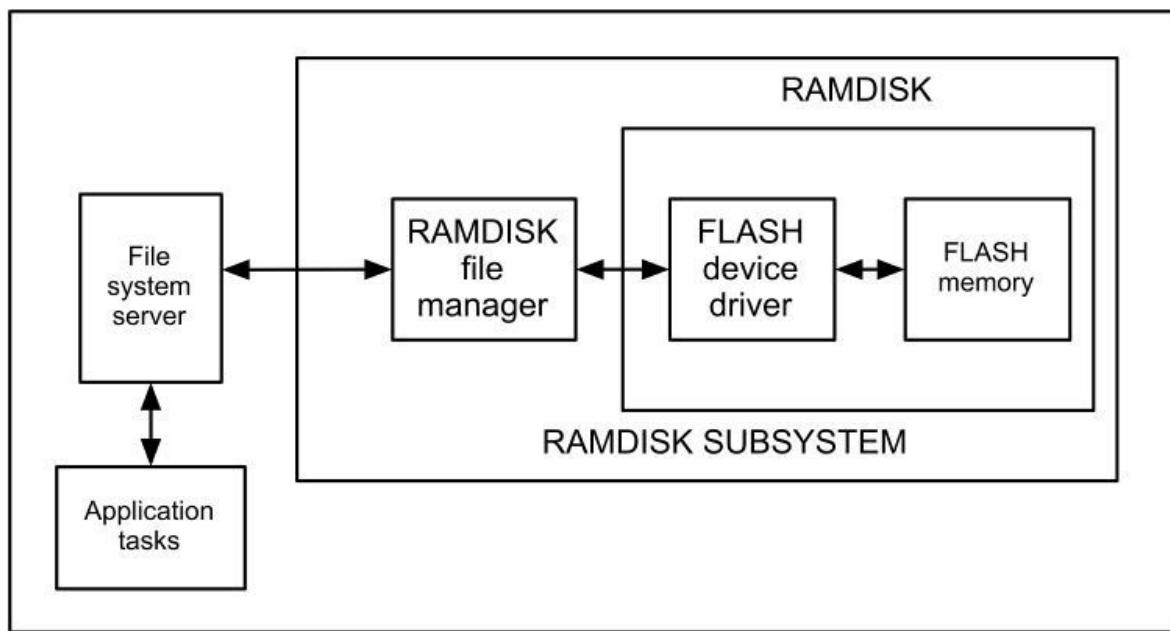


Figure 6.15 Embedded file system - ramdisk storage

It is normally provided as a sub-system within a full embedded file system structure. Here application tasks interface to the individual storage subsystems (e.g. hard disk, floppy disk, PCMCIA device) via a file system server. Each subsystem has its own individual file manager, thus effectively becoming a 'plug-in' component. The application programmer shouldn't need to know anything about the type of storage being used; this should be made transparent by using standard APIs. More will be said on this topic later.

On a closing point, be careful with the use of Flash memory. Remember that this, like EEPROM, 'wears out' as a result of changing its stored data (i.e. write operations). When it is used as a 'read-mostly' memory, this is unlikely to be a problem. However in Ramdisk applications, writing is much more commonplace; wearout is a real issue. Thus file managers often use special

techniques to minimize this effect (note: minimize, not eliminate). The approach adopted is usually a two-pronged one:

- Spread out the wear across the device memory locations and
- Provide spare (reserve) storage space.

The first method is called wear levelling. It works by not using fixed locations for stored data. Instead, each time a data item is rewritten it is stored in a new location. This, on average, reduces the number of re-write operations carried out on each individual store location. More complex wear-levelling techniques also move static blocks of data (i.e. those who's contents don't change) around the memory. The vacated spaces can then be used for re-writing operations.

The second method is called bad block management. To use this, spare capacity must be provided on the disk (e.g. the size of a 32 GByte disk might actually be 35 GBytes). In this approach all disk accesses are checked for errors. If errors are detected then the blocks are marked as bad and then replaced by a spare.

Note that many of these features are also used with removable media.

Review

You should now:

- Know why volatile and non-volatile data storage is used in embedded systems.
- Know what devices are used for such storage.
- Know what OTPROM, EPROM, EEPROM, Flash, SRAM and DRAM devices are.
- Understand the features and relative merits of the various memory devices in terms of cost, speed, power consumption and reliability.
- Know how memory devices are organized and used in embedded systems.
- Know how RTOS elements relate to memory usage.
- Understand why task interference can occur and how MPUs and MMUs are used to prevent such interference.
- Understand the difference between static and dynamic memory allocation techniques.

- Know that dynamic memory allocation can result in memory fragmentation and leakage.
- Understand how a secure memory allocation strategy can prevent these.
- Understand the principles and practices of secure memory allocation.
- Know what solid-state drives are and how such memory is managed.

Chapter 7 Multiprocessor systems

The objectives of this chapter are to:

- Explain why we would choose to use multiple processors.
- Describe what hardware is available to implement such systems.
- Describe the structures of multicore and multicomputer devices.
- Show practical ways to partition software for use in embedded multiple processor systems.
- Look at how to allocate the partitioned software to the processors.
- Describe ways to control the execution of this software.
- Explain the principles and practices of symmetric and asymmetric multiprocessing systems.
- Describe the problems that come from the use of symmetric multiprocessing techniques.

7.1 Embedded multiprocessors - why and what?

7.1.1 When one processor just isn't enough.

You've produced and delivered a successful RTOS-based product. It does the right job in the right time and also has spare processing power and memory space. Wonderful! But as time goes by change requests come in (very much a fact of life in real developments). And, not surprisingly, these usually increase the workload of the processor. Well, as long as there is spare processing capacity, that isn't a problem. But what do you do when changes do impact on performance?

Most likely your first line of attack is to start optimizing the code for speed, using assembly language where necessary. When that doesn't work any more you move to plan B: running the processor at higher speeds. It may also turn out that you can move software algorithms into hardware (using, for example, coprocessors or FPGA technology). However, where do you go to when these no longer solve the problem? The reality, unfortunately, is that more processing power is needed, which can be achieved only by a hardware redesign. And now is the time to consider if it would be better to use multiple processors rather than a single high-performance one.

Of course, if you're starting out on a brand new design, this question should be asked (and answered) right at the outset.

7.1.2 Processor structures - a general overview.

For embedded applications multiple processor systems can be roughly categorized as being physically centralized or physically decentralized (more precisely, distributed). The driving forces behind these structures are quite different. Where high performance is a key requirement the best solution is a centralized one. In contrast the rationale for developing decentralized systems is quite different, but that is the subject of the next chapter.

In this chapter we'll look at the hardware needed to implement high-performance physically centralized systems. Figure 7.1 shows the most common practical structures used in embedded multiprocessor systems. A chip-based design is that of the multicore device whilst a board-based one is the multicomputer. Here, for example, the multicore processor consists of two CPUs together with key processor-related devices: interrupt management, memory and timers. This

processor is embedded within a single chip microcontroller that also includes various real-world devices. By contrast, the multicomputer structure consists of a set of boards, each one housing a microcontroller. These microcontrollers don't have to be identical; you could, for example, have a conventional microcontroller such as the Cortex on one board and a DSP on the second one (or even a multicore device). Extremely high-performance computers can be built using multicomputer structures, well suited to demanding applications such as command and control systems. Also, board-based designs have the advantage of supporting processor redundancy, a feature of critical systems.

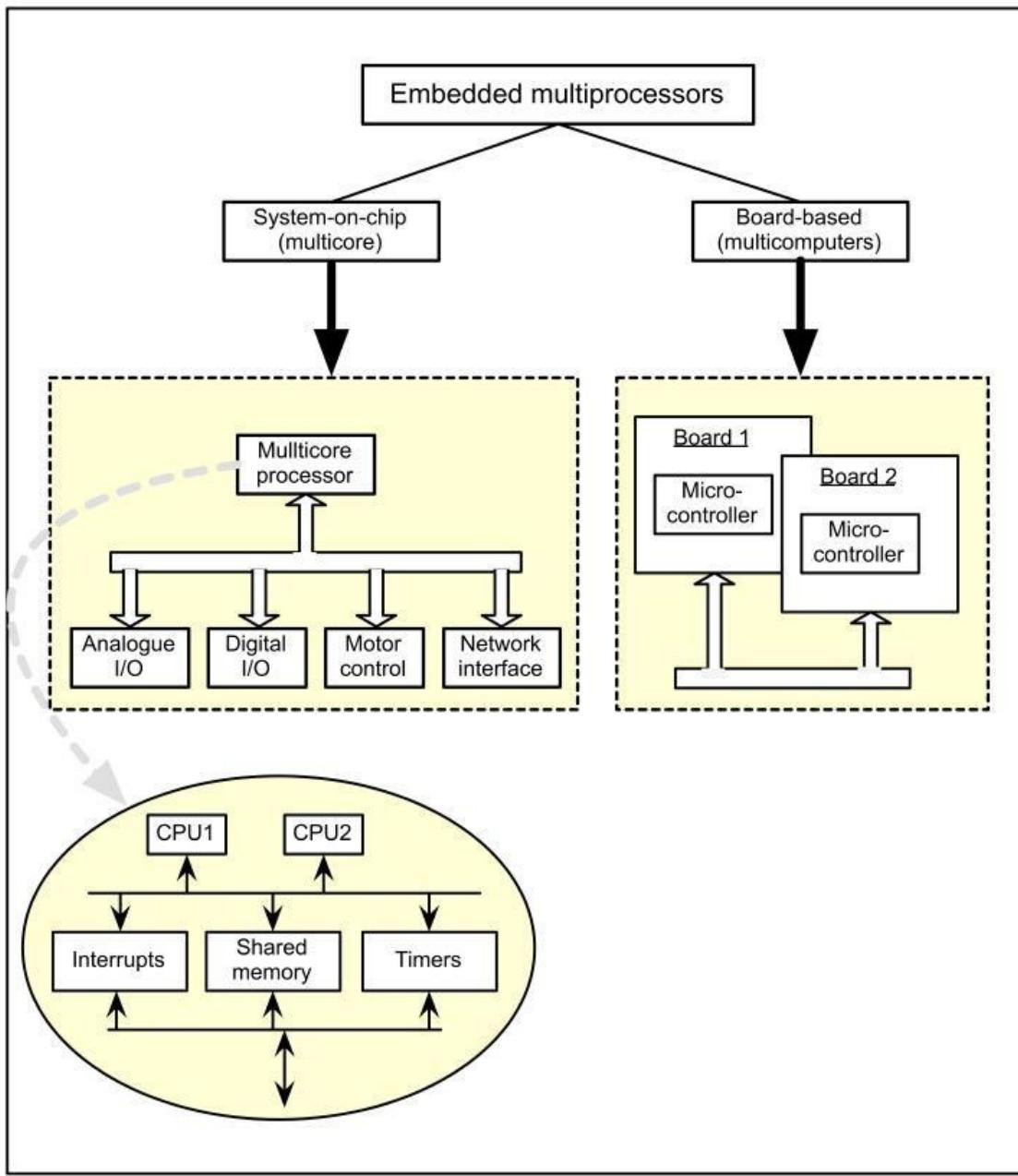


Figure 7.1 Multiprocessors in embedded systems

7.1.3 Multicore processors - symmetric and asymmetric types.

From a hardware perspective processors come in two forms: symmetric and asymmetric. Basically, with a symmetric multiprocessor design, all the processing units are identical; with asymmetric multiprocessors the units differ.

An example of an embedded multicore symmetric multiprocessor is the ARM Cortex A9, figure 7.2 showing a simplified description of its key features. This

has four identical processing units (cores), each one consisting of a CPU, hardware accelerator, debug interface and cache memory. It can be seen that several on-chip resources are shared by all the processing units. From a software perspective the device can be used in two ways. First, each core can be allocated specific tasks, and hence is considered to be a dedicated resource. Second, any core can run any task, thus being treated as an anonymous resource.

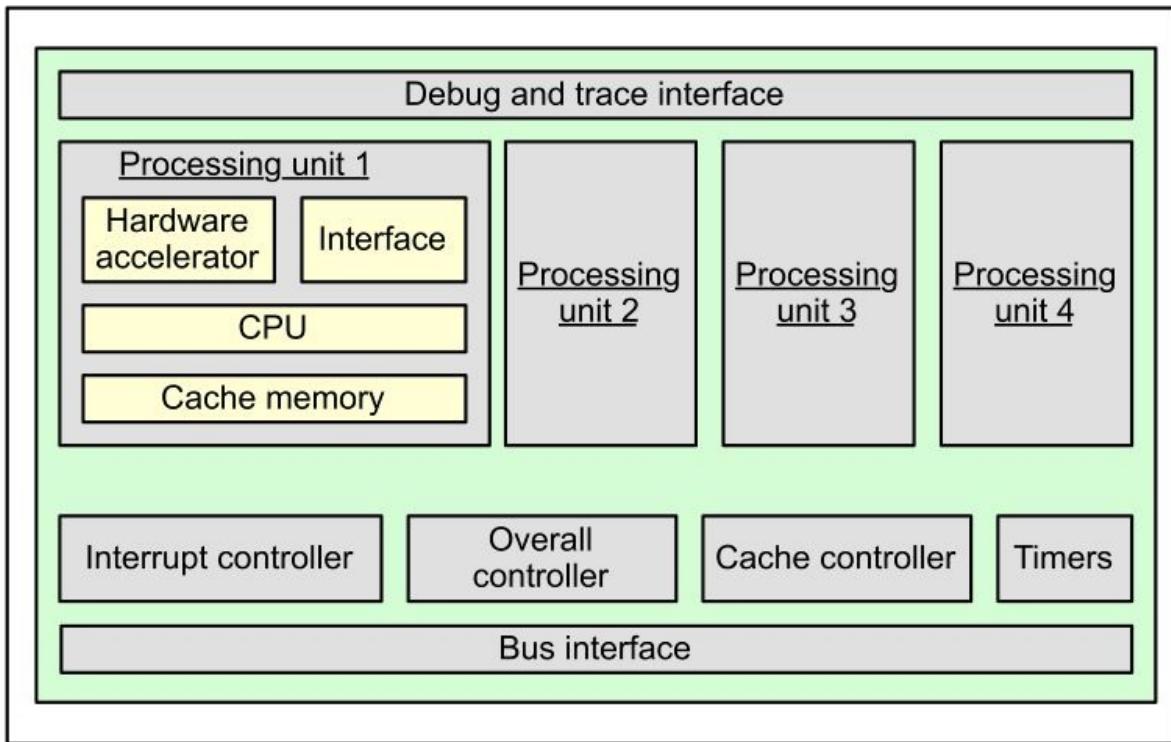


Figure 7.2 Example symmetric multiprocessor - multicore implementation

An example of an asymmetric multicore multiprocessor is the Texas TMS320DM6443, figure 7.3.

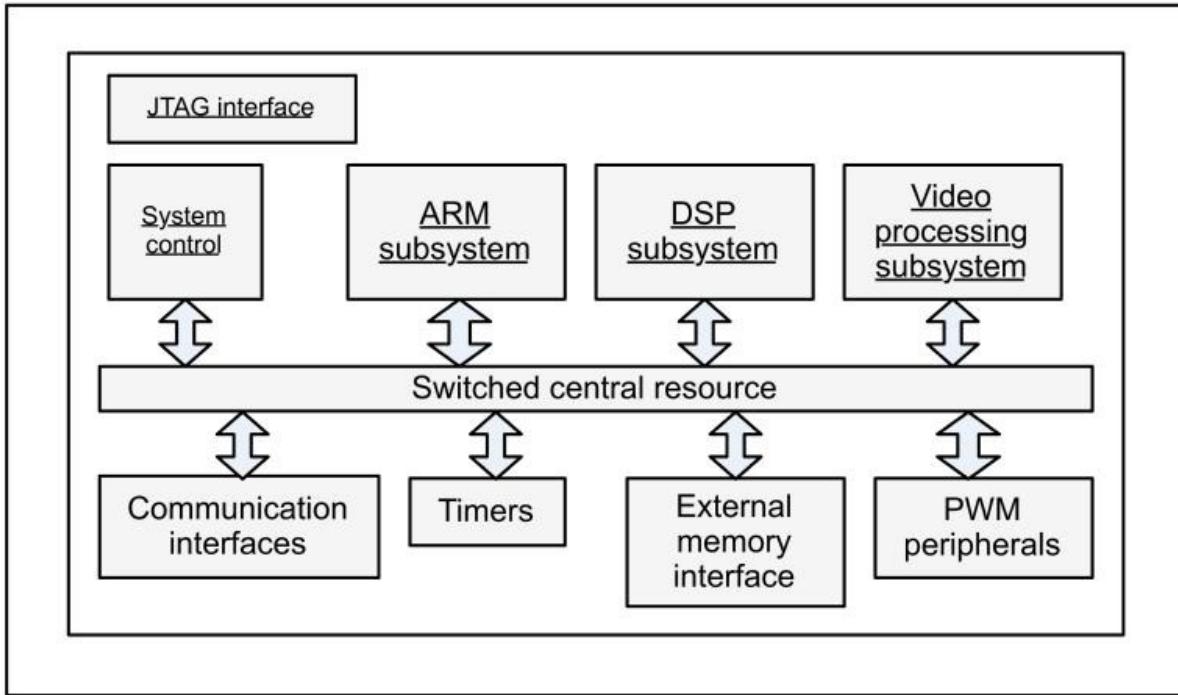


Figure 7.3 Example asymmetric multiprocessor - multicore implementation

In this device there are two distinct processing units, one for general purpose computing and the other for digital signal processing.

Generally multicore processors are organized as single-board computing elements. These may be part of a complete single-board computer or else form the single processing unit of multi-board designs. Either way they can be viewed as being equivalent to very high performance single core devices, quite different to multi-board multicomputer structures.

7.1.4 Multicomputer structures.

Embedded multicomputer systems come in two flavours, figure 7.4: loosely-coupled (a) and closely-coupled (b). In most cases the individual computers are built as single-board units which house all required hardware devices. Typically these are mounted in some form of electronic rack unit, all signalling being done via a backplane bus or a simple serial signalling scheme (e.g. I2C).

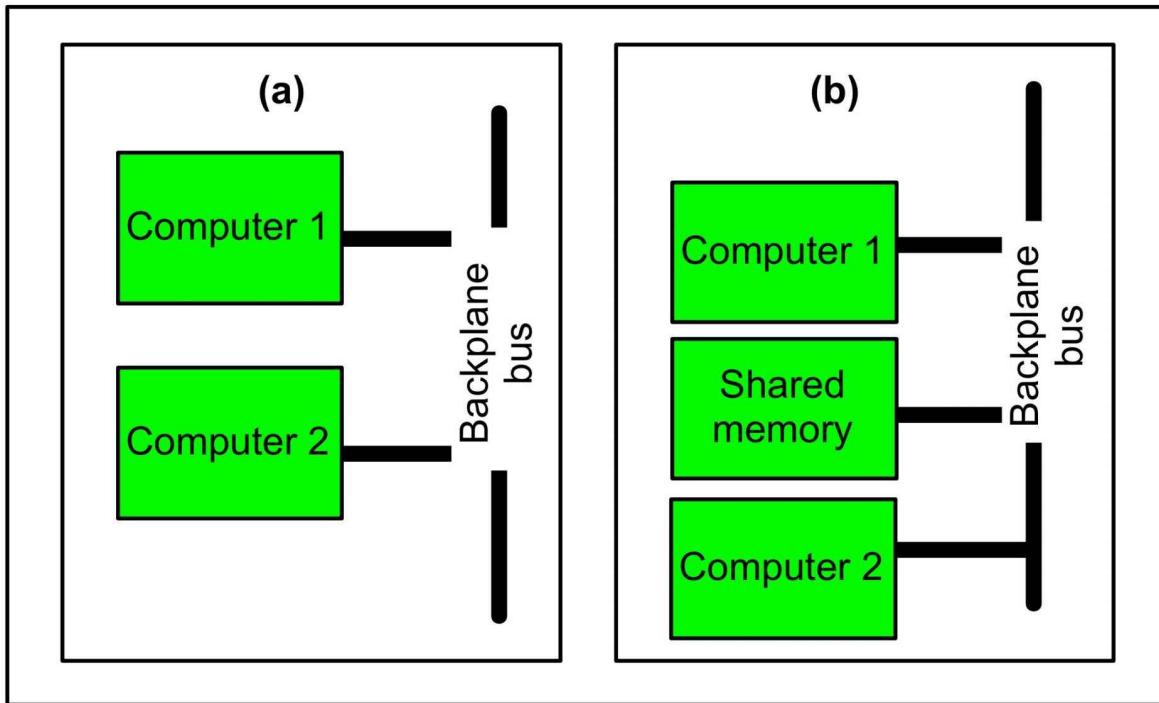


Figure 7.4 Basic multicommputer structures

In loosely-coupled systems communication between the computers is done using message-passing techniques. As a result individual units don't need to know very much about the others, thus minimizing dependency (coupling) between them. What this gives us is a very flexible technique for the building, configuring and reconfiguring of multicommputer systems. Its one drawback, however, is that the communication time overhead (a combination of protocol handling and data transmission) may prove to be a bottleneck for system performance. One way to alleviate this is to use closely-coupled designs.

With close-coupling, communication between the computers is done using shared memory. This eliminates the need for protocol handling, merely requiring some form of access control mechanism. From a design perspective this looks more like a distributed task structure, tasks communicating via flags, mailboxes, etc. Where communicating tasks sit on different processors their common communication components reside physically in the shared memory.

Please understand that this is a very broad-brush view of the topic but is sufficient for the needs of this book.

7.2 Software issues - job partitioning and allocation.

7.2.1 Introduction.

When designing and implementing the application software, three key decisions have to be made:

- How should the software be split up - partitioning.
- Where should the resulting software units reside - allocation.
- How should software execution be controlled - scheduling.

Let us first look at software partitioning and allocation and deal with scheduling later.

7.2.2 Structuring software as a set of functions.

In chapter 1 the basic ideas of partitioning, the structuring of a system as a set of concurrent sub-systems or functions, were laid down. For simplicity we'll call this functional structuring (not to be confused with traditional software functional decomposition). Let us look again of the example of the oxygen-producing plant, figure 7.5. This, remember, is an abstract model which is independent of the target hardware.

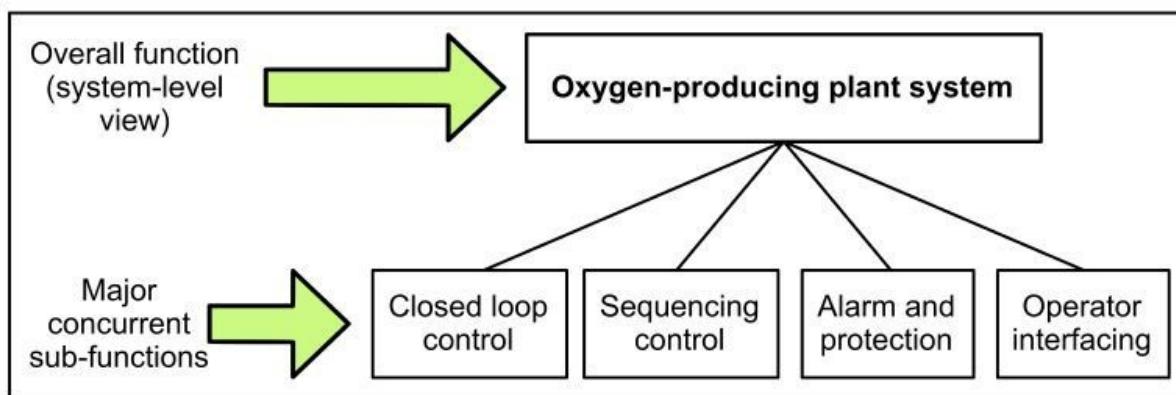


Figure 7.5 Functional structuring of the oxygen-producing plant system

What we'll now do is partition the overall software in exactly the same way and then develop the design and code models. A very natural mapping of the ideal design onto the real hardware is shown in figure 7.6, where each major sub-function runs on its own processor.

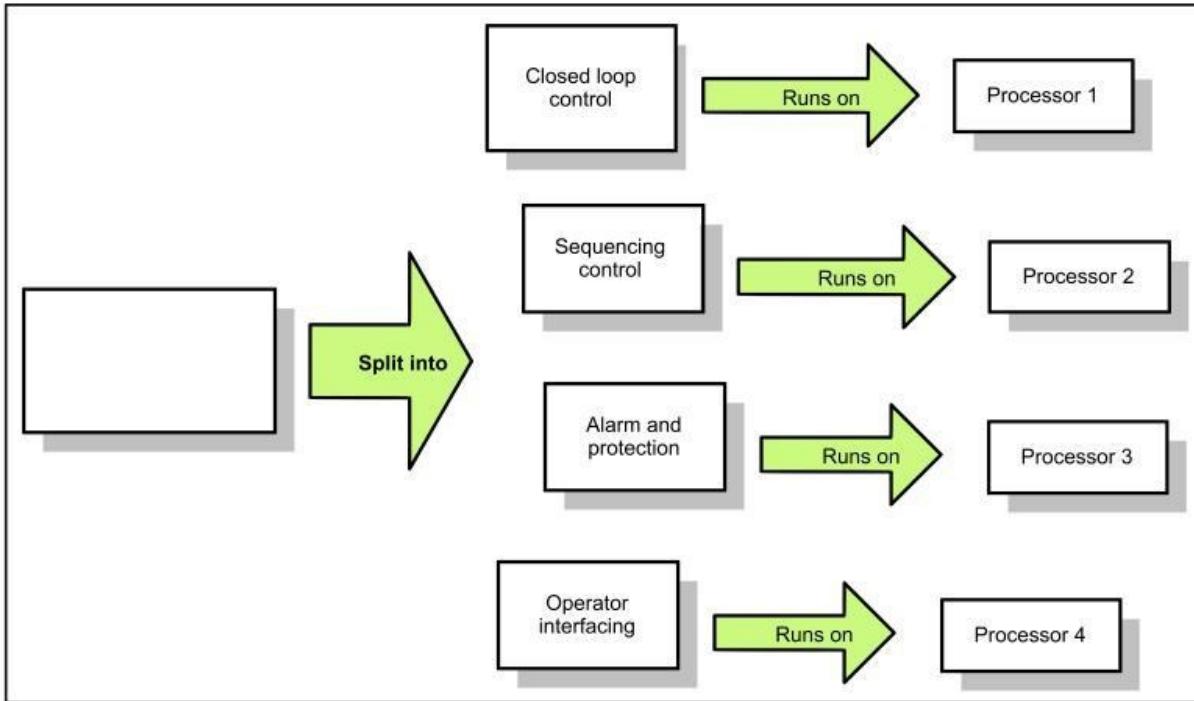


Figure 7.6 Allocating the software to processors

There isn't a unique hardware design solution here, but the two leading contenders are:

- A four-core multicore SOC unit or
- A four-board multicompputer system.

What this example points up is that effective software allocation is heavily dependent on the available hardware. Also, the interfacing requirements are likely to play a big part in the choice of hardware. Regrettably there isn't a cookbook solution to this challenge; brainpower combined with software expertise is needed here.

It can be seen that the functions performed by the individual processors are quite different. Does that mean we have to use an asymmetric multiprocessor? In this case the answer clearly is no; we should be able to use a symmetric multiprocessor without problem.

Now let's look at an example which naturally calls for an asymmetric multiprocessor implementation, figure 7.7.

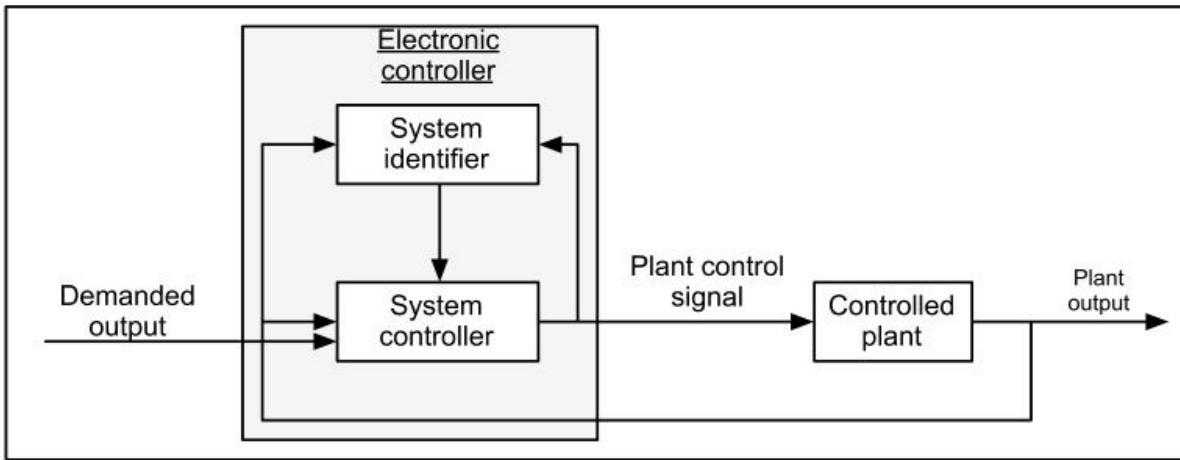


Figure 7.7 Adaptive control system example

Here the electronic controller has two major functions, system control and identification. A conventional processor can readily handle the control function but the identification process is more of a problem. The reason is that the work involved is heavily mathematical, hence computationally demanding. Such software is best suited to running on a digital signal processor, figure 7.8.

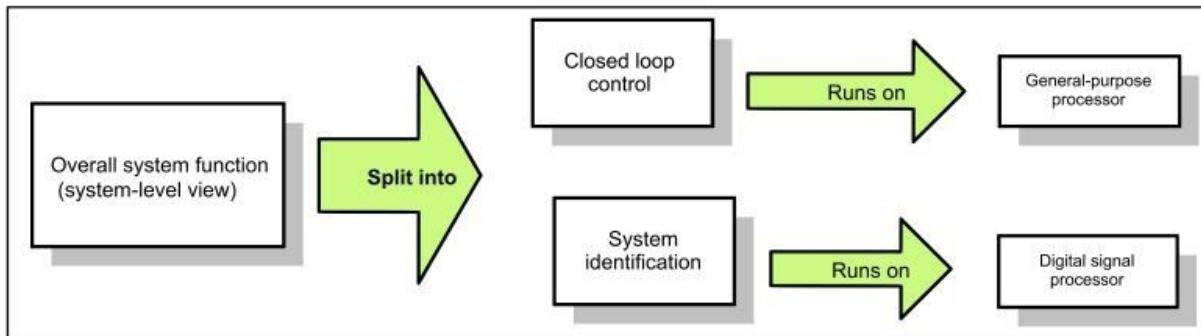


Figure 7.8 Allocating the software to processors - adaptive control system

The chosen system structuring dictates an asymmetric multiprocessor solution, one possible candidate being the multicore device shown in figure 7.3.

Functional structuring is the bedrock of concurrent software design. It allows the designer to handle complexity using a simple 'divide and conquer' strategy. Moreover, it gives, at all time, a clear view of system coupling and cohesion. But be aware; in some cases it may not deliver the best performance for multiprocessor-based designs, especially data-intensive ones.

7.2.3 Structuring software as a set of data processing operations.

What have the following systems got in common?

- Video and audio.
- Image processing.
- Radar and sonar.
- Pattern recognition.

Basically they all are signal-processing systems, mostly having to work in real-time. As such the computing workload is usually very high, often exceeding the capability of a single DSP. Once again we need to turn to a multiprocessor solution, either multicore or multicomputer as appropriate. However, functional structuring, in this case, is not the best solution; a model based on data processing structures maps very well onto multiprocessing hardware. Take, for example, a typical simple signal processing algorithm of the form:

$$y = F_0(x_0) + F_1(x_1) + F_2(x_2)$$

Here y is the current output, x_0 is the current input sample, x_1 is the previous input sample, etc.; the F values represent the associated mathematical operations. For simplicity this has been limited to three data samples, giving three parallel data streams, expressed diagrammatically in figure 7.9.

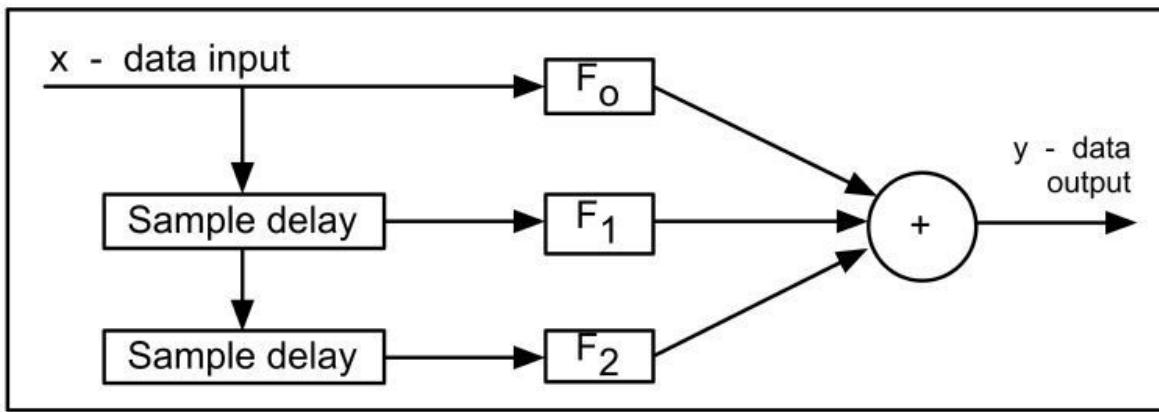


Figure 7.9 Example digital signal processing algorithm

The equivalent block diagram of the software processing is shown in figure 7.10; figure 7.11 details the mapping of these processes onto hardware. In this case the target system is a symmetric multicore structure running four concurrent software units.

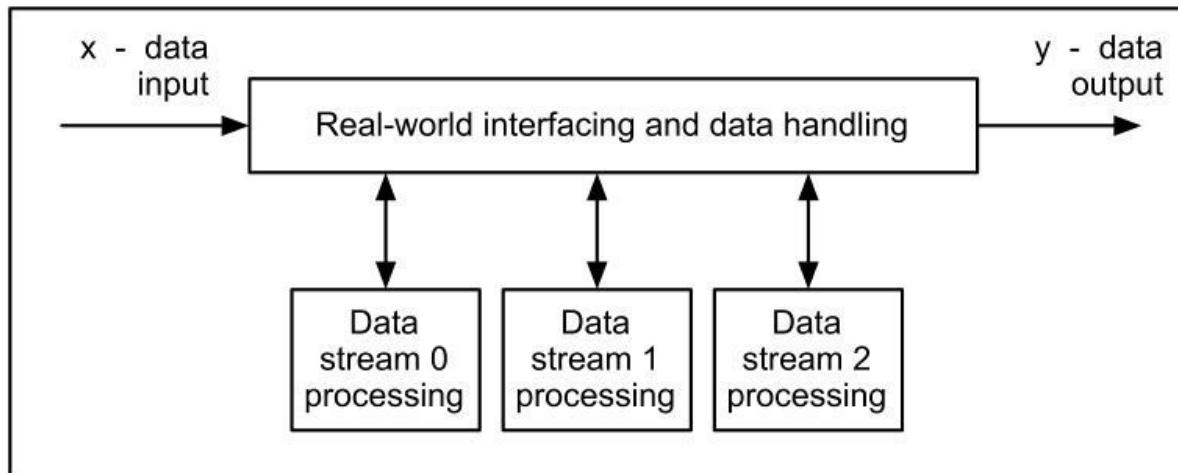


Figure 7.10 Software processing block diagram

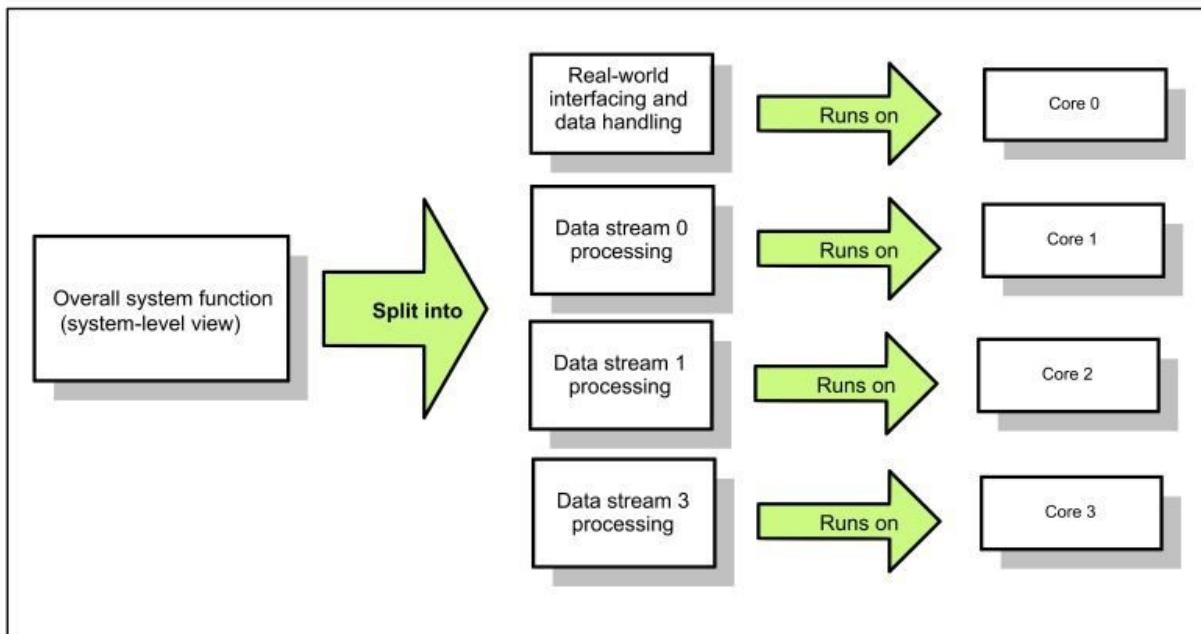


Figure 7.11 Allocating the software to processors - signal processing example

At this point we need to take a brief digression from the main topic to consider the definitions of tasks and threads. Until now these have, for simplicity, been taken to be synonymous. Unfortunately, the use of sloppy and incoherent terminology in the software world has muddied the waters. In reality tasks and threads may or may not have the same meaning; it just depends whose definition you use! We will define it in this context to define threads to be equivalent to 'lightweight' tasks. This means they are concurrent units having minimal context switching overheads. So the program implementation of figure 7.10 can be

viewed as being a four-thread one.

So far, so good; the software structure has mapped very nicely onto the given hardware structure. But what happens when things don't go so smoothly? For example, given a four-core multicore unit, how would we implement the following algorithm:

$$y = F_0(x_0) + F_1(x_1) + F_2(x_2) + F_3(x_3) + F_4(x_4) + F_5(x_5) + F_6(x_6)$$

The partitioning aspects are clear cut, a natural solution being eight tasks: seven for data processing plus one for interfacing and data handling. But the same isn't true concerning the allocation of work to cores. With four cores and eight threads, work must be carried out as a combination of parallel and serial activities. We could, of course, make design-time decisions as to how the work is farmed out to the cores. But would this make the best use of system resources; in particular would it deliver the best performance? Possibly. But a better solution is to make such decisions at run-time, operations being handled by the operating system. This approach is the basis of symmetric multiprocessing, more of which later.

And now for a little bit of advice. Don't be surprised if your own designs fail to map onto hardware as easily as those shown here.

7.3 Software control and execution issues.

7.3.1 Basic OS issues.

The examples of the previous section illustrate two key points concerning the allocation of software:

- First, if particular software units are best run on specific hardware, then an asymmetric multiprocessor should be used.
- Second, if each software unit can be run on any processor (core or computer), then a symmetric multiprocessor should be used. Note that the software units do not have to be functionally identical.

Figure 7.12 gives an overview of task and thread allocation in multiprocessor systems.

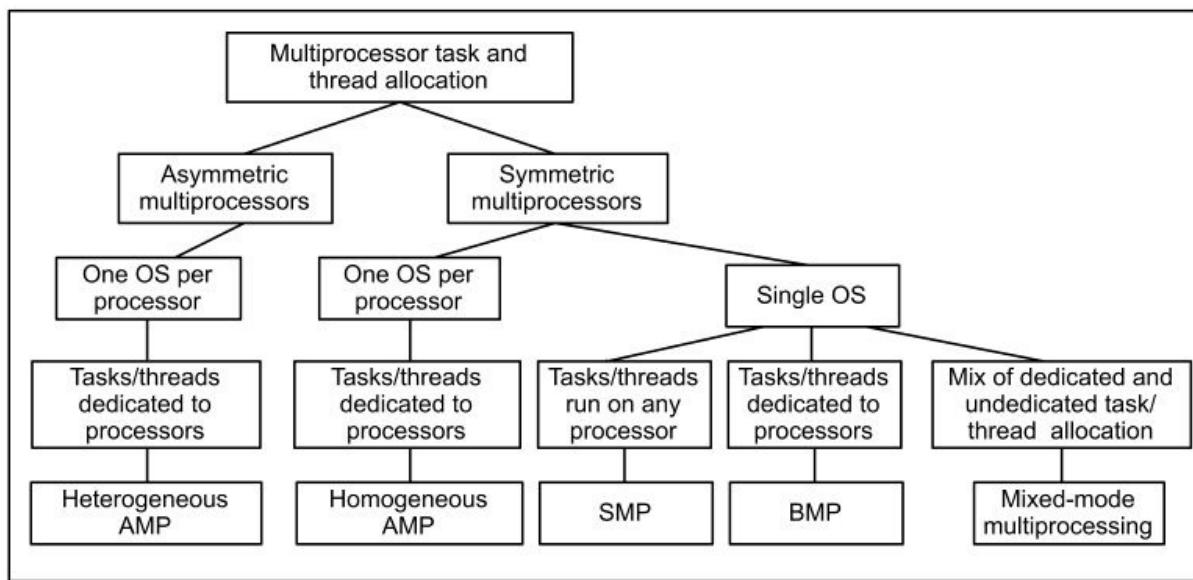


Figure 7.12 Multiprocessor task and thread allocation

With asymmetric multiprocessors the OS and scheduling issues, and the way to deal with them, are fairly straightforward. Each processor usually has its own executive (or even a full-blown RTOS), one that is most suited to the job in hand. For example the TMS320DM6443 could run with Linux on the ARM core and the Texas Instruments real-time kernel on the DSP. This mode of operation is normally called asymmetric multiprocessing (AMP). Sometimes it is described as heterogeneous AMP, as the operating systems are dissimilar.

With symmetric multiprocessors the situation is not so clear-cut; two distinct

modes of operation have to be considered. First, each processor runs its own OS, these being identical, with tasks/threads being dedicated to specific processors. This is called homogeneous AMP. Second, one OS only is used to control all operations. Within this mode there are three sub-modes. In one implementation any task/thread can run on any processor, the symmetric multiprocessing (SMP) mode of operation. In the second, tasks/threads are dedicated to defined processors. There isn't an agreed, standardized name for this; examples in use include hybrid SMP, bound multiprocessing (BMP) and single-processor mode. The third is a mixture of SMP and BMP, here called 'mixed mode' processing.

In SMP and BMP designs the OS software can run on any processor, not necessarily being dedicated to a specific one. Some designs allow the OS to migrate between processors during run time.

Multiprocessor OSs often support a mixed mode of operation where some cores work in BMP mode and the others operate as true SMP types.

In general the term AMP covers both heterogeneous and homogeneous types. It will be clear from the hardware structure which type is being used.

Please note that while these features apply generally to multicore units there may be significant differences at the detailed level. For these you'll need to check out the information provided by the vendors.

7.3.2 Scheduling and execution in AMP systems.

AMP systems behave like a set of collaborating microcomputers sharing a predefined set of resources such as memory, interrupts and peripherals. Because each micro has its own OS the designer is not limited to a single scheduling technique (this is true of both multicore and multicomputer designs). In a two-core system for example, core 1 could execute in a priority pre-emptive mode while core 2 could run using round-robin scheduling (in practice this will depend on the chosen OS). Software is normally provided to support application-level communication, synchronization and mutual exclusion functions.

The designer normally determines the allocation and use of the various resources, and does this explicitly. Such information is 'hard-wired' into the source code, being executed during processor initialization. Generally, for embedded applications, resources aren't dynamically reallocated during program execution.

AMP systems give us improved performance without sacrificing the

determinism, predictability and security of the single core RTOS. This is particularly important for critical hard systems. Moreover, it is relatively straightforward to port an existing single-processor design to an AMP system. But the fact that the processors have dedicated tasks means that individual processor loadings (hence utilization) may, in practice, be quite different. This limits the speed-up advantages of an AMP design; to achieve very high utilization a different approach is needed.

7.3.3 Scheduling and execution in SMP systems.

The key feature, the whole purpose, of SMP systems is to deliver high performance. We do that by trying, whenever possible, to keep all processors fully utilized (load balancing). The key to this is to:

- Have one OS only.
- Allow this OS to see the whole system.
- Enable the OS to dynamically control the allocation of tasks/threads to micros.
- Permit the OS to allocate and control the use of system resources.

This method also eliminates the need for the programmer to consider low-level detailed activities; instead he can concentrate on the application itself.

To demonstrate this point, consider programming the algorithm

$$y = F0(x_0) + F1(x_1) + F2(x_2) + F3(x_3)$$

for a four-core device using Pthreads. Here we'll concentrate on one part of the problem only, viz.:

- Creation of four threads.
- Execution of the code of the individual threads and
- Synchronization of the completed threads.

Each thread will perform a single multiplication; after thread synchronization the y value is computed (not shown here). To make things very clear, assume that we program four functions to do this work, as follows:

```

void* ComputeF0X0(void* arg)
{
    Code of function
}

void* ComputeF1X1(void* arg)
{
    Code of function
}

```

and similarly for the remaining two functions.

```

/* Declare the threads */
pthread_t thread0, thread1, thread2, thread3;

/* Create the threads and execute their code */
pthread_create (&thread0, NULL, ComputeF0X0, &Thread0Data);
pthread_create (&thread1, NULL, ComputeF1X1, &Thread1Data);
pthread_create (&thread2, NULL, ComputeF2X2, &Thread2Data);
pthread_create (&thread3, NULL, ComputeF3X3, &Thread3Data);

/* wait for all threads to complete execution - thread synchronization */
pthread_join (&thread0, NULL);
pthread_join (&thread1, NULL);
pthread_join (&thread2, NULL);
pthread_join (&thread3, NULL);

/* if we reach this point then all threads have finished their work */

```

Listing 7.1

(Note that by using generic names, arrays and iterations this source code could be reduced to a minimalist, and possibly impenetrable, form).

You can see from this that the programmer plays no part in deciding where the threads should run. This is, in fact, decided dynamically by the OS software, and depends on the current run-time loading. The code is also highly portable; for example, there is no need to change the source code to run it on a two-core device. Another advantage is the flexibility of the implementation technique; exactly the same approach could be used for (say) an eight-thread filter implementation.

Here we have a classic solution to the need to program a set of essentially independent parallel, repetitive calculations. In this case it doesn't matter which cores the threads run on. Nor, from a functional point of view, does it matter if there are variations in the execution times from run to run; the join operation ensures that things progress in a predictable manner. So it's no surprise that multicore designs feature strongly in signal processing applications. But for the broader range of embedded applications SMP may not be the best solution because:

1. It is often impossible to predict the execution order of individual threads, even with a priority pre-emptive scheduler. If execution order is important (e.g. for coordination purposes) the code must be explicitly designed to guarantee that this will happen.
2. It is impossible to predict on which core the processes will run. When the software is flawless this isn't an issue; the same isn't true when bugs are present.
3. Real-time behaviour may not be guaranteed for some processes. For example, in a single processor system a high priority interrupt service routine will always pre-empt normal scheduled threads. This ensures that there can be no conflict between the two. However, in a multicore design the ISR could be executed on one core whilst the others continue to run the scheduled threads.
4. These factors, taken together, can be important when porting an existing single processor design to a multiprocessor system. Synchronization, coordination and exclusion properties may not be maintained.

This leads us into the subject of bound multiprocessing.

7.3.4 Scheduling and execution in BMP and mixed-mode systems.

With BMP you can specify which processors the tasks/threads should run on. This may have advantages when used with critical software, providing a high degree of predictable behaviour. The downside is that it's a somewhat rigid approach, limiting the overall performance of the processor system. However, by combining SMP and BMP techniques (mixed-mode processing) we can get the best of both worlds. Good flexibility, high performance and sound predictability can be attained.

With mixed-mode processing we simultaneously run tasks that aren't dedicated

to specific processors (i.e. SMP) with those that are (i.e. BMP). Where the key attributes of tasks are performance, reliability and robustness, then the BMP solution is best. In general we would use this for hard-fast tasks. However, soft tasks can in many situations be run in SMP mode, taking advantage of the processing power of the micro. An example of this is shown in figure 7.13. Here the Autopilot and Autoland tasks can execute only on their designated cores; the other four tasks are run on cores 2 and 3 as defined dynamically by the scheduler. However, these soft tasks cannot use cores 0 and 1, even when these are idle.

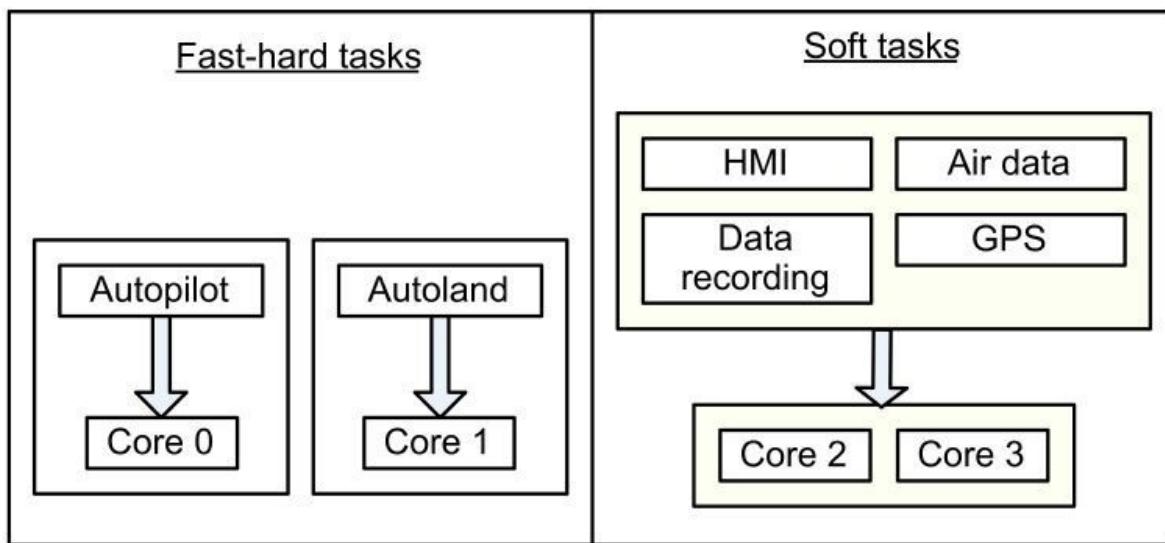


Figure 7.13 Example task/thread allocation in a mixed-mode system

7.3.5 A last comment.

Where high throughput is a major requirement then SMP seems to be the best solution. But simply porting software from single-processor designs onto a multiprocessor platform may not achieve very much. Programs really have to be written to make full use of a parallel processing environment. There is no doubt that computationally intensive applications are well suited to SMP systems.

The attributes of AMP have been discussed earlier. These make it applicable to a wide range of real-time embedded systems, combining performance quality with enhanced speed. Programs originally written for single processors can be ported to multiprocessors without too much angst (that, of course, assumes they were well designed in the first place).

Mixed-mode processing, as previously noted, seeks to get a balance between SMP and BMP. This approach is likely to give best results in the more complex embedded applications.

The final point to make is that if you don't have good tool support you're not going to get very far in developing these software systems.

Review

You should now:

- Appreciate the reasons for using multiple processors in place of single-processor devices.
- Know how, in general, multicore and multicomputer devices are structured.
- Understand what is meant by symmetric multiprocessors, symmetric multiprocessing, asymmetric multiprocessors and asymmetric multiprocessing.
- Recognize how to tackle the partitioning and allocation of software in real-time systems.
- Know what is meant by AMP, SMP, BMP and mixed-mode systems, their relative advantages and disadvantages, and the application areas they're most suited to.

Chapter 8 Distributed systems

The objectives of this chapter are to:

- Show how to tackle the issue of software structuring in a distributed computer system.
- Briefly review the general communication and timing aspects of distributed systems.
- Illustrate factors that determine the mapping of software onto hardware in real-time embedded distributed applications.

8.1 Software structuring in distributed systems.

In the section on multicomputer systems we looked at various issues that arose from spreading system software across a number of processors. Here in this section we'll extend that work, but place it clearly in the context of physically-distributed systems. But don't be misled into thinking that all we need to do is scale up the ideas developed earlier. Not so! What you'll find is that, for embedded applications, software design cannot be done in isolation from system design. Later we'll see how system factors affect our software decisions; for the moment let's establish the key ideas in developing distributed designs.

Let us start by looking at a realistic problem: the development of a computer-based controller for the system of Figure 8.1(a) (a simplified version of a marine propulsion system).

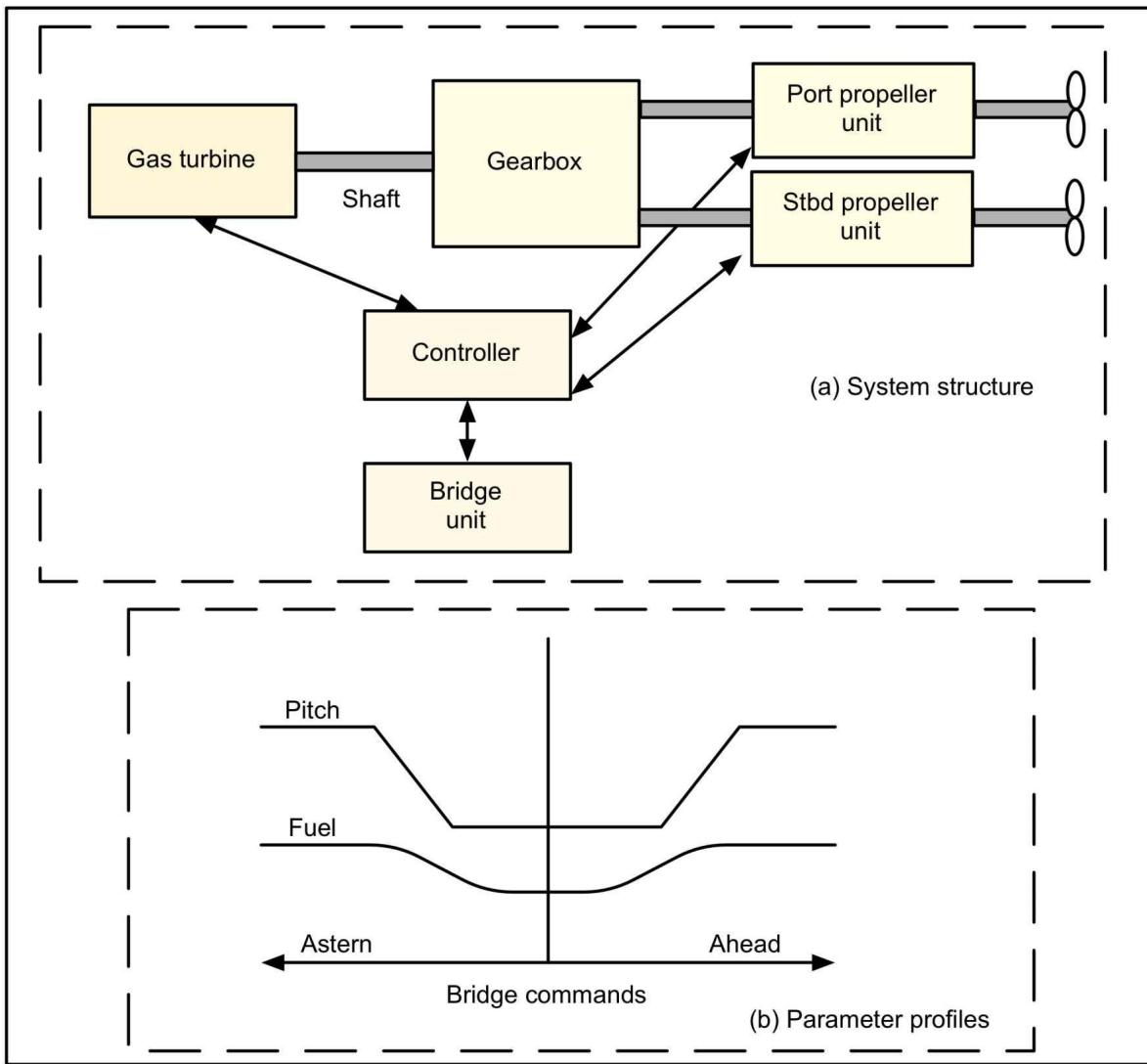


Figure 8.1 Example system - marine propulsion

Here a single propulsion engine - a gas turbine (GT) - drives two propeller shafts via a gearbox. The propellers are of the variable pitch type, being operated by individual propeller units. The function of the controller is to control three items: GT fuel flow, port propeller pitch and stbd. propeller pitch. These are set in response to bridge commands using the parameter profile information of figure 8.1(b).

Initially a single, centralised computer scheme is proposed (stay with us; the reason for going down this route will shortly become clear). After studying the problem a first-cut multitasking (seven-task) design scheme is produced, Figure 8.2.

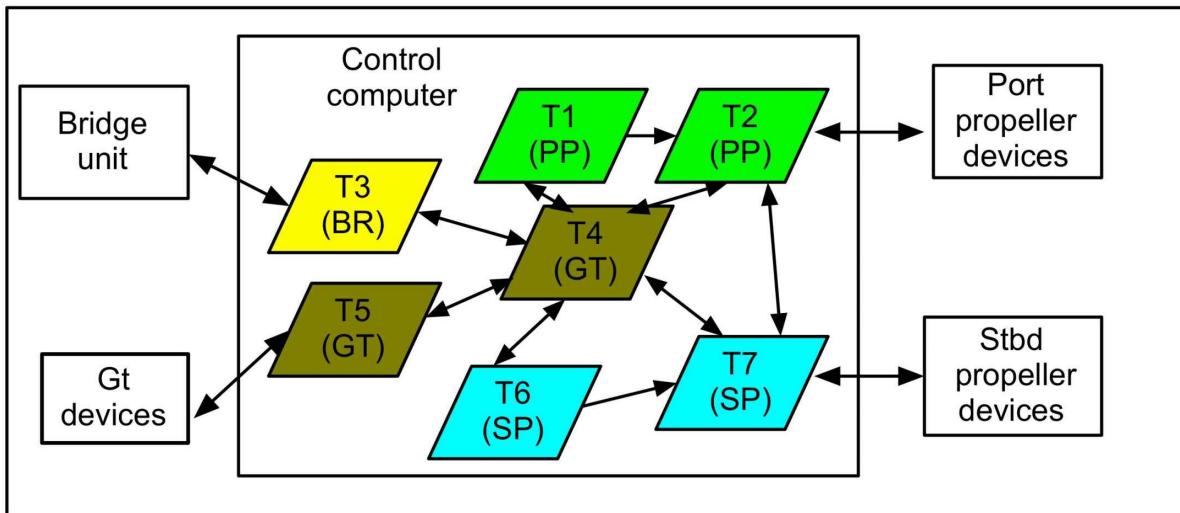


Figure 8.2 Centralised control - simplified tasking structure

For clarity comms components have been excluded from the tasking diagram.

It is now decided that a distributed networked arrangement would be a better solution to the problem. The resulting design is that of Figure 8.3.

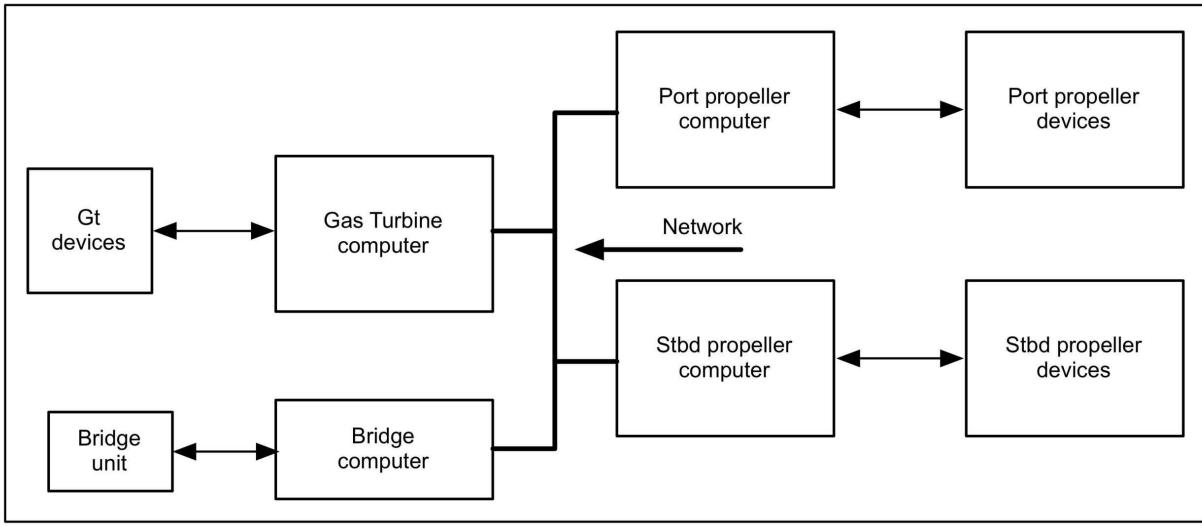


Figure 8.3 Distributed control - computer structure

The next point may sound too obvious to need making (but that won't stop us). From a propulsion system point of view, the function, behaviour and performance of both solutions should be identical.

We review our previous design work and conclude that the overall software structuring of Figure 8.2 is an optimal one. That is, it not only does the job; it is the best solution taking all factors into account. This can now be treated as an abstract software model that has to be distributed across the system. The outcome is shown in Figure 8.4, which initially deals only with the tasks of the abstract model.

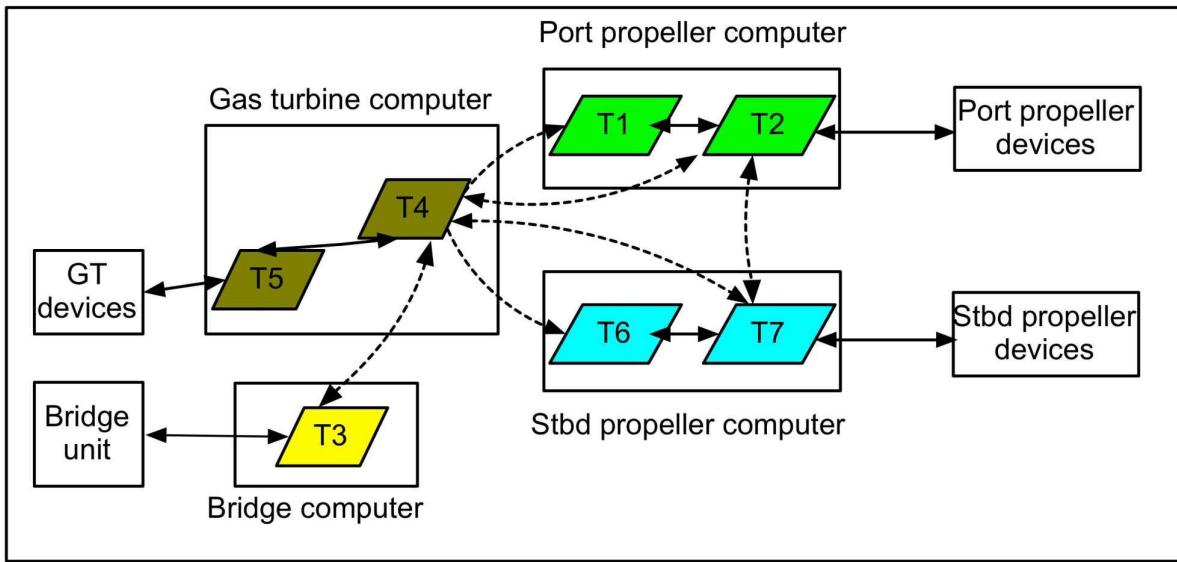


Figure 8.4 Distributed system - task allocation

Naturally enough, the inter-task communication defined in Figure 8.1 must be maintained in the new design. However, some of this now goes across the network, shown as the dashed lines of Figure 8.4. From this it is clear that, because the system is distributed, we need support for:

- Network message handling (physical and protocol aspects).
- Message routing across the network.
- Message routing within each computer.
- Co-ordination and timing of computer activities.

8.2 Communication and timing aspects of distributed systems.

There are a number of ways to deal with the communication and timing requirements of distributed applications. The approach taken here is a very practical one. Moreover, it has proved to be highly successful in the implementation of real-time embedded distributed systems. It hinges on the following:

- All inter-processor communication is based on message-passing techniques.
- Communication is essentially asynchronous; any synchronization requirements must be explicitly designed into the software (precluding, for instance, the use of remote procedure calls).
- For hard and/or critical systems the location of software functions is pre-defined (no dynamic redistribution is allowed at run time).
- The software within individual computers is organized as desired (e.g. multitasking, interrupt-driven, etc.).
- If timing is critical, then a system-wide time reference signal ('global time') must be provided.

By taking these factors into account the basic design of Figure 8.4 becomes the practical one of Figure 8.5. Here, within each computer, the application tasks are

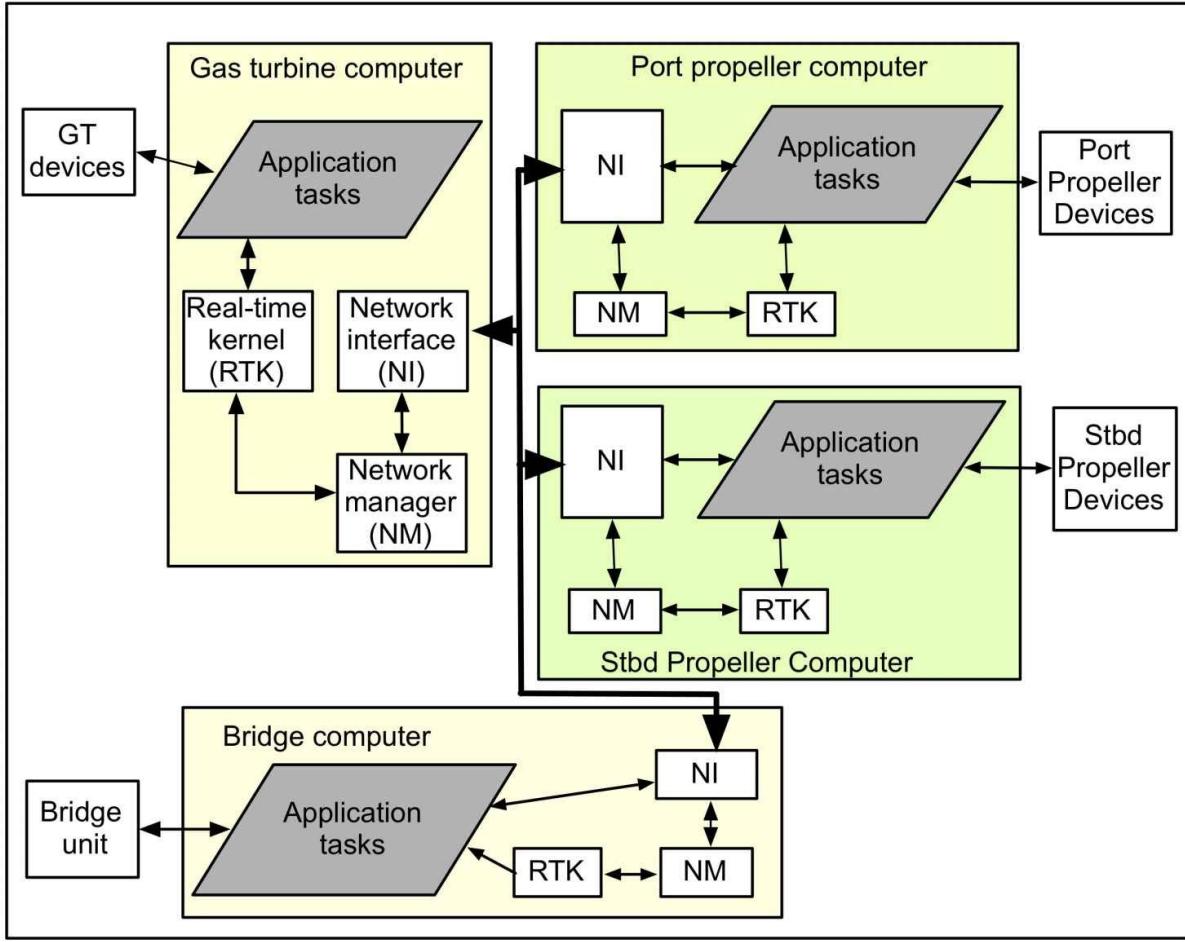


Figure 8.5 Distributed system - overall system and software structure

- as before - run under the control of the real-time kernel. A network interface section deals with message handling aspects whilst a network manager deals with message routing. This design can be implemented in many ways, ranging from DIY solutions to complete reliance on operating system facilities. The first is hard, time consuming, and requires much skill and knowledge. However it gives full visibility and control of the design, together with a complete understanding of timing aspects. The second, which (relatively speaking) is easy and fast, shields the designer from all low-level computer and network details. A key feature of this approach is the provision of transparent network support, the so-called 'virtual circuit'. One technique based on this is the 'virtual connection, figure 8.6(a).

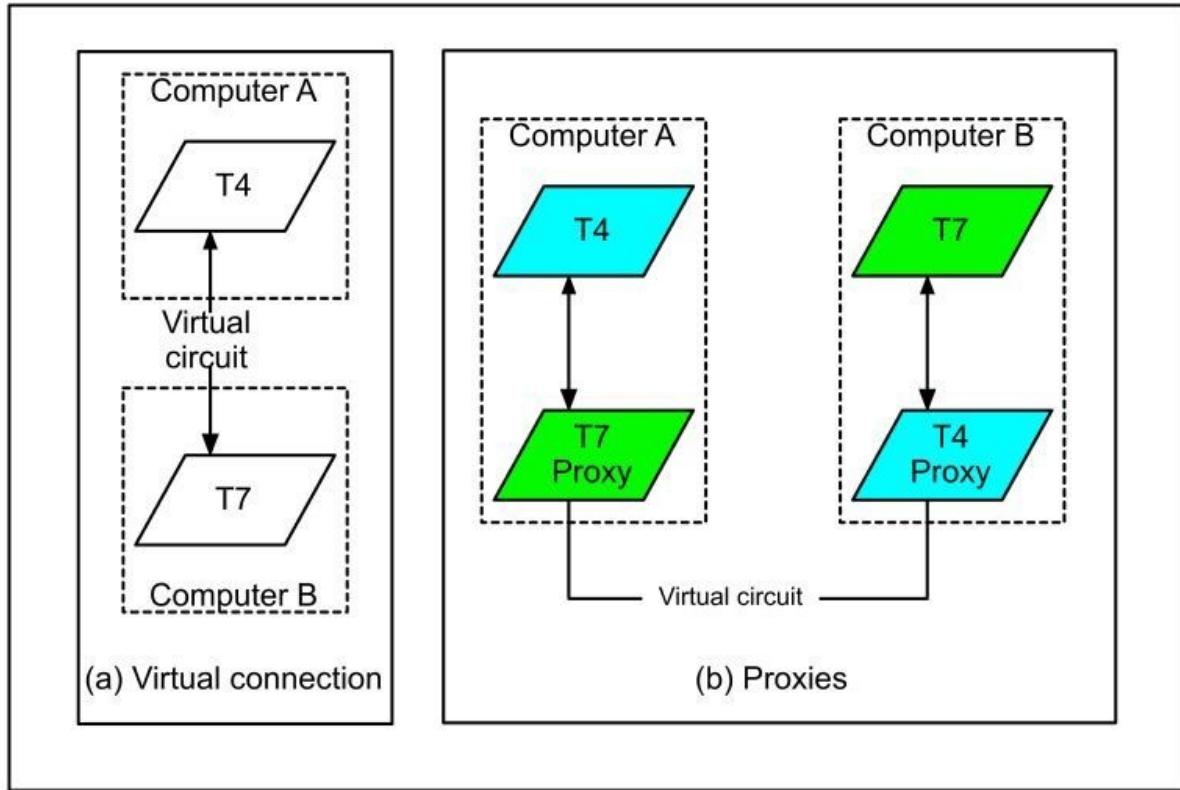


Figure 8.6 Virtual connections and proxies

In this example tasks 4 and 7, each on a different computer, need to communicate with each other. The OS provides mechanisms to connect the two together, the connection channel being invisible to the programmer. When task 4 wishes to speak to task 7 (a 'remote' task) it merely includes the identifier of task 7 in the message call. All location and routing details are handled by the operating system network manager.

An alternative method is based on the use of 'proxies', Figure 8.6(b) (although this is normally met in OO designs as proxy objects, not tasks).

Proxy, OED: 'A person authorised to act for another'.

With this arrangement, extra tasks - the proxy tasks - are created to simplify the programming of the communication software. On computer A, for instance, task 4 talks to the proxy of task 7 (a 'local' task). As far as task 4 is concerned task 7 lives on the same machine; thus standard (non-distributed) task communication features can be used. Task 7 proxy carries information about the real task 7 so that it can forward all messages from task 4 to it. A similar situation applies in computer B, except that we now have a proxy of task 4.

This is a very powerful technique and can be of great value in large systems. Implementation is very much simplified by using standard software ('middleware') to build the proxies and the virtual circuits. Unfortunately, you now lose sight of what precisely happens at run-time. Thus it can be quite difficult (if not impossible) to guarantee timing performance. Further, the behaviour of the system under fault conditions may not be predictable (in particular from a time point of view).

The example used here, although highly simplified, does point up the extra difficulty met in designing distributed systems. Moreover, it reinforces the key point; the fundamental issue is one of system and software design, not just task scheduling.

8.3 Mapping software onto hardware in distributed systems.

Experienced designers know that, for any system, there is no uniquely right design method. The reality is that there are usually a few good techniques and a whole host of dreadful, kludge-type solutions (usually dressed up as 'cutting-edge' in the software world). Moreover, different applications often need to be tackled in different ways. So we'll be parochial here and limit ourselves to distributed real-time embedded applications having a mechatronic bias. In such systems the route to producing successful designs is to:

- Define the system architecture.
- Develop an abstract software model that meets all functional and timing requirements.
- Partition the software across the processors of the system.
- Design the networking and multitasking software within each processor.
- Check that the design still meets its requirements. If not, go round the loop again.

To use a cliché, this is easier said than done. But it is the path most likely to bring rewards.

In an ideal world partitioning decisions can be based on software considerations only. Unfortunately if you do this for real-time embedded systems there's likely to be a collision between aspirations and reality. It's not a case of software only; many other factors have to be taken into account, the major ones being:

- Physical organization.
- Safety.
- Architectural issues.

These, in general, carry much greater weight than software aspects. They impact heavily on system structure and hence processor organization. And these factors aren't necessarily independent; we may well have to juggle with all of them within a single system.

(a) Physical organization.

Many real-world systems consist of a collection of separate physical units that are geographically dispersed. These range from the very large to the relatively

small, including, for example:

- Supervisory control and data acquisition (SCADA) on oil platforms.
- Control and monitoring of public utilities such as water, electricity and gas systems.
- Environmental control of buildings.
- Smart-house monitoring.
- Remotely-controlled vehicles.
- Robotic-based systems.

One feature of such systems is the use of 'intelligent' devices - motors, actuators, and sensors, etc. - for interacting with the real-world. An even more demanding application is the use of autonomous vehicles, as for example in warehouse control operations.

The consequence of this is that certain software functions must be local to devices. And, the number and siting of processors are determined by system, not software, requirements.

(b) Safety.

Where safety aspects enter the design equation, two important questions need to be addressed. If a problem comes up, who deals with it and where is it dealt with? For us that means:

- Which bit of software handles the problem? and
- Where is the software located?

The answers to these will emerge from the design process, driven primarily by two factors:

- The physical structure of the computer system: uni-processor, centralized multiprocessor or distributed systems.
- The safety-critical level of the system.

For distributed systems the safest, securest and most reliable approach is to localize safety-related software. That is, we make devices as autonomous as possible. This decision is based on a number of factors, the two most important ones being:

- The consequence of a failure of the communication system.

- The required speed of response to fault conditions.

When we enter the realms of high safety-critical systems things become much more complex. Here redundant components are essential, including items such as sensors, actuators and displays and not just computers. Triple-redundancy is commonplace, whilst at the highest-level full quad redundancy is used.

Moreover, not only is there a need to use redundant computers: a mix of processor types, programming languages and software development teams is also called for. This is a major subject in its own right, beyond the scope of this book.

(c) Software architectural issues.

These are best explained by example, in particular looking at the difference between federated systems and highly integrated systems, figures 8.7 and 8.8. These actually apply to two physically distributed systems that are almost identical in their functional and physical structures.

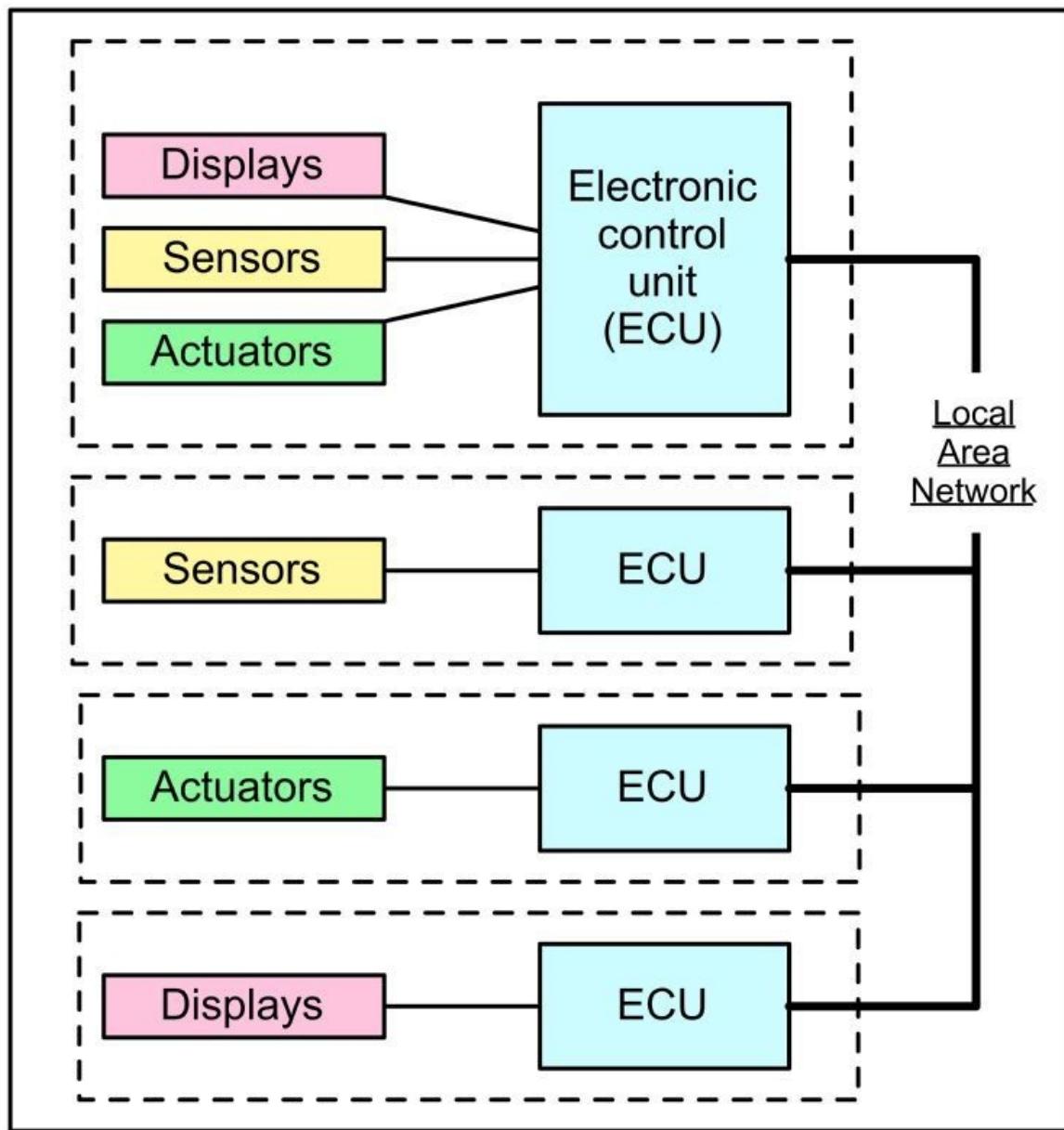


Figure 8.7 Modular system - federated structure

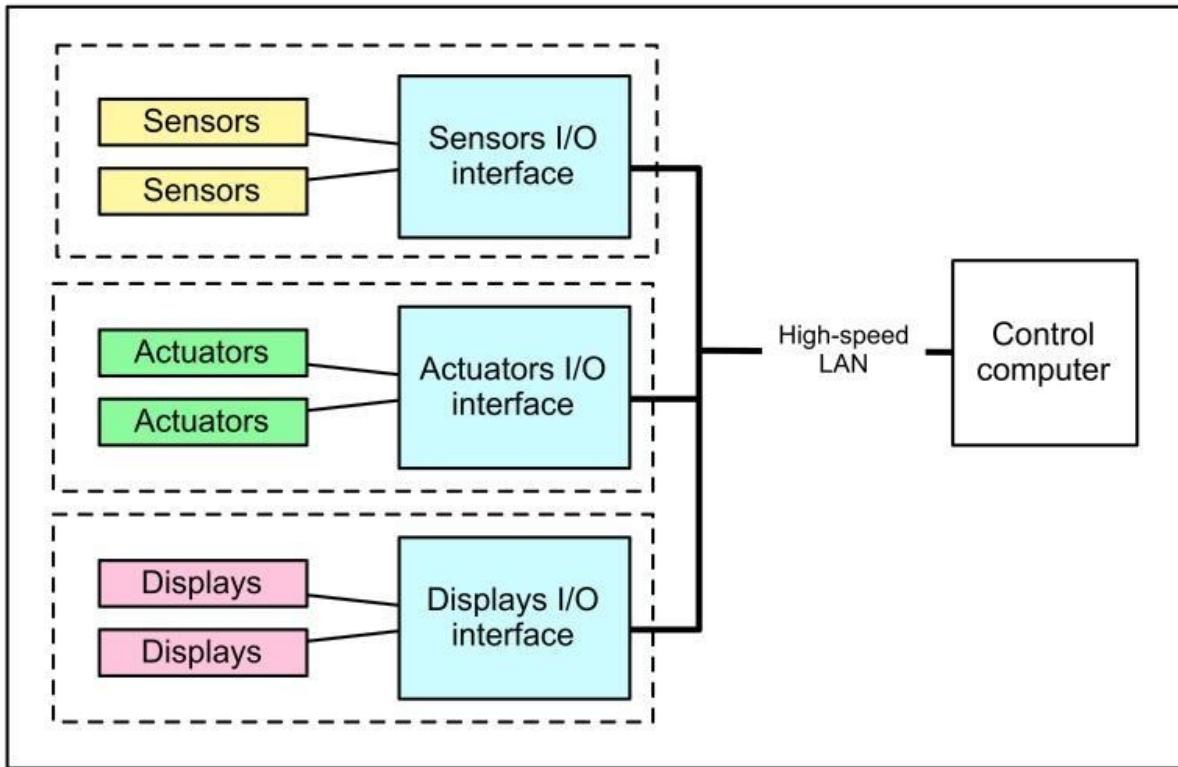


Figure 8.8 Modular system - integrated structure

With the federated structure we have a collection of individual but interacting, cooperating modular units. This design approach is well suited to situations where different companies develop the individual electronic modules. As such, the detailed software design is handled on a module-by-module basis. Hence, at the system level, the key factor is not a software design one per se; it's mainly a case of making sure that all the units work together correctly. That is, we need to clearly specify the functional, temporal and interfacing requirements of the various units.

In the integrated structure the role of the electronic modules is limited mainly to collecting data and issuing control commands. The application software, which consists of a set of individual subsystems, runs on the control computer. Conceptually these subsystems are separate. However, the reality is that there is no actual physical isolation between them (unlike those of federated schemes).

It is self-evident that the two software architectures are radically different.

Review

You should now:

- Understand why, when designing software for distributed systems, a system approach must be used.
- Understand why the decisions made when mapping software onto hardware are crucial ones.
- Know why, in general, it's best to localize decision-making as much as possible.
- Understand the role and structure of comms software in distributed applications.
- Appreciate that the networking software increases the time and complexity overheads of the design.
- Know what virtual networks and proxies are.
- Realize why timing and performance aspects need to be addressed right from the outset of the design.
- Understand why software needs are usually subordinated to system requirements.

Chapter 9 Analysis and review of scheduling policies

The objectives of this chapter are to:

- Provide a broad overview of scheduling policies and show their relationships to each other.
- Explain the fundamentals of static and dynamic scheduling policies.
- Give an in-depth view of priority-based pre-emptive static scheduling techniques.
- Give an in-depth view of priority-based pre-emptive dynamic scheduling techniques.
- Explain the concept of rate-groups and how they are used to provide more predictable performance.

9.1 Overview

Earlier we looked at some basic ideas of scheduling and scheduling rules ('policies'). The objective here is to introduce the broader range of scheduling policies which are - or may be - used in real-time systems. Many different techniques are available, yet relatively few are widely applied. By the end of the section it will become clear why this is so. First though, a general overview, Figure 9.1. This, it should be said, shows one way of categorizing the various scheduling policies; other classification schemes are possible.

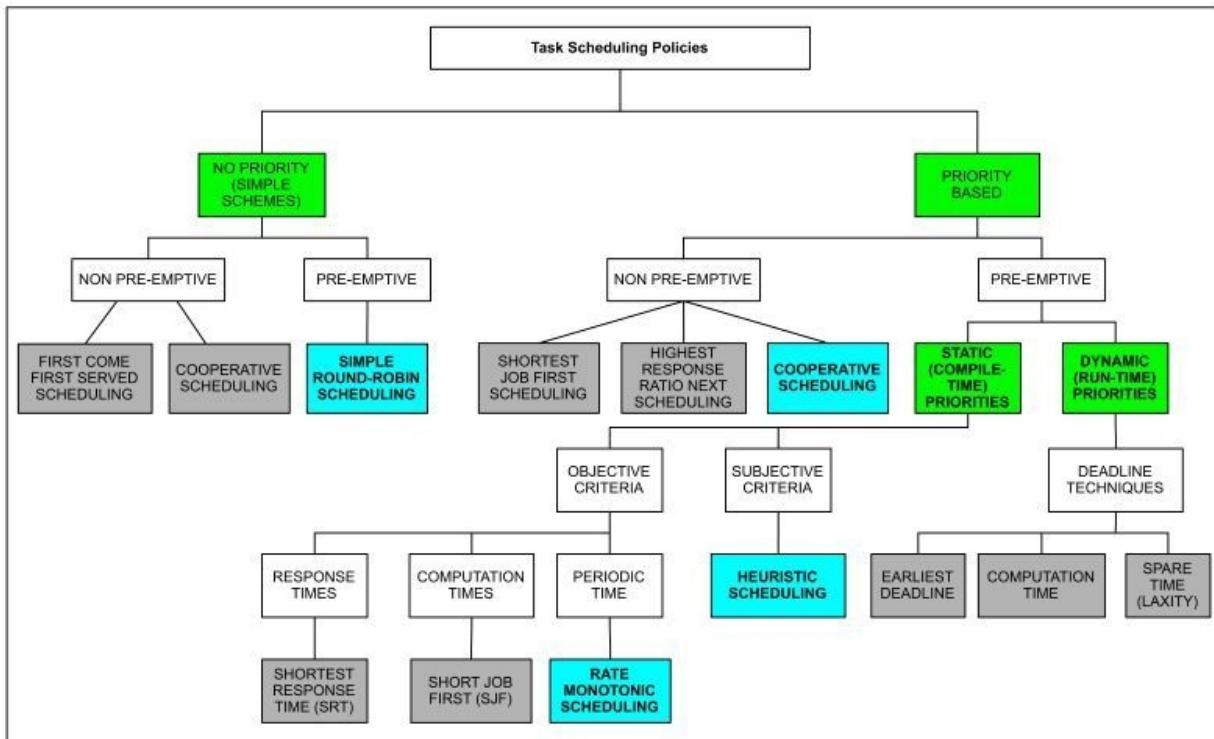


Figure 9.1 Scheduling policies for real-time systems

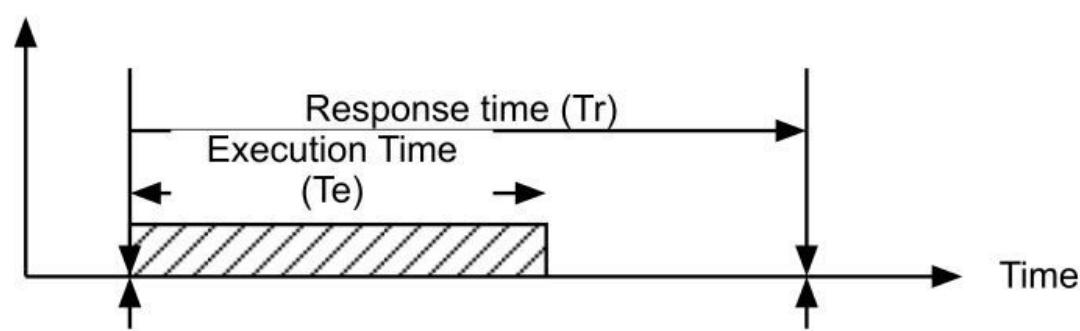
For brevity those methods which don't use priority mechanisms are called 'simple' schemes. We have already met the First Come First Served policy as a FIFO one. Also, simple Round-Robin scheduling has been covered in depth. A discussion on Cooperative scheduling will be deferred for the moment, so allowing us to move on to the priority-based schemes. These, it can be seen, are collected into 'non-pre-emptive' and 'pre-emptive' groups. The pre-emptive category is in turn divided into two major sets, 'static' and 'dynamic'. Here the word static is used to denote schemes where task priorities are assigned at compile time (these are also called 'a priori' or 'off-line' scheduling). By contrast, dynamic policies set priorities as the software executes, changing them to

respond to operational conditions. This implies that there must be predefined, mechanised rules for adjusting such priorities 'on-line'.

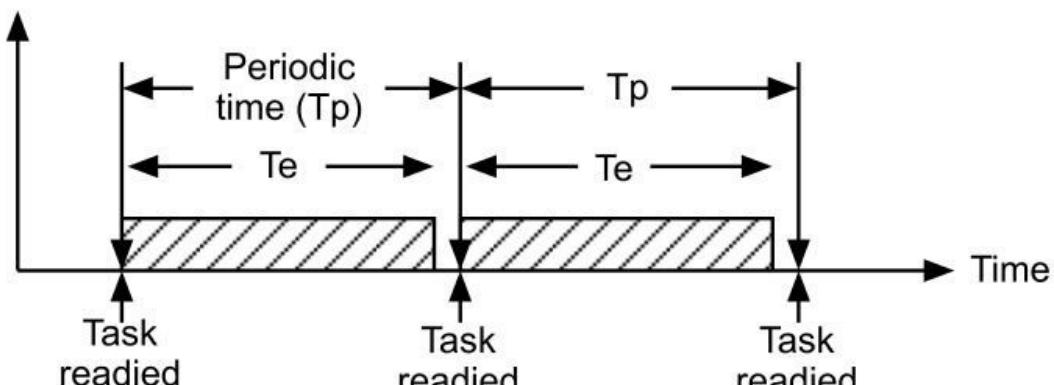
With off-line scheduling it is the responsibility of the programmer to define task priorities. This raises a most important - and difficult - question. How does a designer deduce the best (optimal?) task schedule for a particular application? Two approaches are in general use. The first, an objective method, relies on task time information to define what the various priorities should be. The second is subjective; that is, the selected schedule depends mainly on the opinion of the designer. Here this is called 'heuristic' scheduling.

The more-important policies will be discussed in detail after some basic timing definitions have been revisited, figure 9.2 (most of these were introduced earlier in the book). Figure 9.2(a), which applies to aperiodic tasks, identifies four important items:

- Task arrival time (T_a): the time at which a task is put into the ready-to-run state.
- Task deadline (T_d): the time by which the task must deliver its result.
- Response time (T_r): The time allowed for a task to carry out its function (note: if timing is done on a relative, not absolute, scale, then response time and deadline have the same value).
- Execution time (T_e): The actual execution time of the task.



(a) Aperiodic tasks



(b) Periodic tasks

Figure 9.2 Task timings - some basic definitions

For periodic tasks, Figure 9.2(b), the time between successive invocations of a task is called the Periodic time (T_p).

A last point; static scheduling does not imply fixed priorities. It may - depending on the design of the kernel - be possible to change a task's priority during program execution. However, such changes are normally invoked as a result of decisions made by the application program: not by the operating system itself. By contrast, with dynamic scheduling priority settings are:

- Usually determined by the rules of the OS.

- Likely to change during program execution.

It is also possible have scheduling policies that combine both static and dynamic techniques.

9.2 Priority-based non-pre-emptive scheduling policies.

The schemes considered here are those of Shortest Job First (SJF), Highest Response Ratio Next (HRRN) and Cooperative scheduling. In these an executing task cannot be pre-empted no matter how low its priority. Either they run to completion or else voluntarily make way for another task. However, the position of tasks in the ready queue is determined by their priorities. Both SJF and Cooperative scheduling are static schemes; the HRRN algorithm used here is a combination of both static and dynamic methods.

(a) SJF scheduling: In essence this is a priority-based version of FCFS scheduling. Each task has its priority (P) calculated as follows:

$$P = 1/T_e$$

The order of tasks in the ready queue is determined by their priorities; that with the highest is placed at the front of the queue.

Compared with the simple schemes, SJF scheduling generally produces a better average response time. The operative word here is average. There may well be significant variation in actual response values obtained during program execution. There is a further complication when a system contains a mix of short and long tasks. Those having long execution times have low priorities. As a result their response times are likely to be adversely affected by the presence of short (higher-priority) tasks. Hence this technique is not suitable for use as the core scheduling policy in real-time systems. It may though be effectively incorporated within one of the priority-based schemes (e.g. to run base-level tasks).

(b) HRRN scheduling: This improves on SJF (in terms of task responsiveness) by taking into account the time tasks have been waiting in the ready queue (T_w). T_w is used dynamically in the calculation of the task priority index, one algorithm being:

$$P = (1 + T_w)/T_e$$

Note: Rewrite this as $P = 1/T_e + T_w/T_e$

From this it is clear that the first component can be calculated off-line ('static')

whilst the second must be computed at run time ('dynamic').

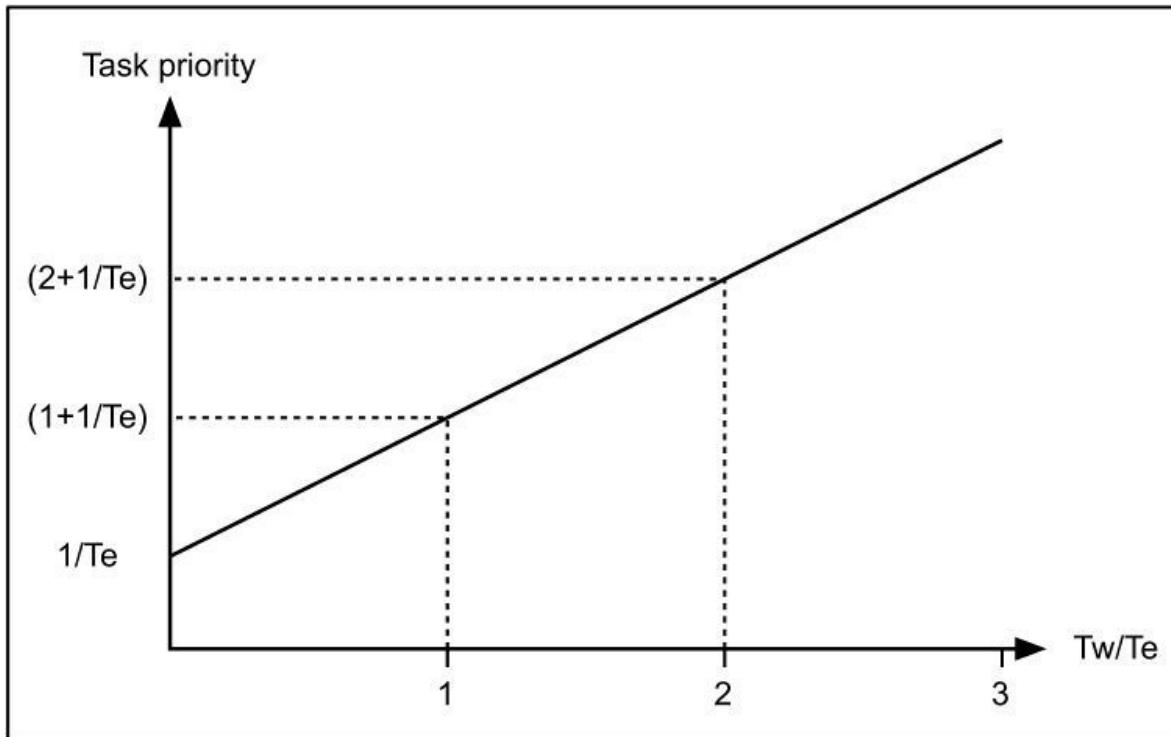


Figure 9.3 HRRN task priority profile - normalised

The normalised priority profile produced by this calculation is shown in figure 9.3. Observe that the 'time' scale is a normalised one, Tw/Te . Consequently each and every task has this same profile. Actual (as opposed to normalised) priority values depend of course on actual execution and waiting times. This is illustrated in figure 9.4 for two tasks that have different execution times. It can be seen that as a task waits in the ready queue its priority is gradually raised (e.g. task 2 reaches the initial priority value of task 1 after it has been waiting for one second). This ensures that tasks that have long execution times (and thus begin with low priority ratings) get better service than that provided by SJF scheduling.

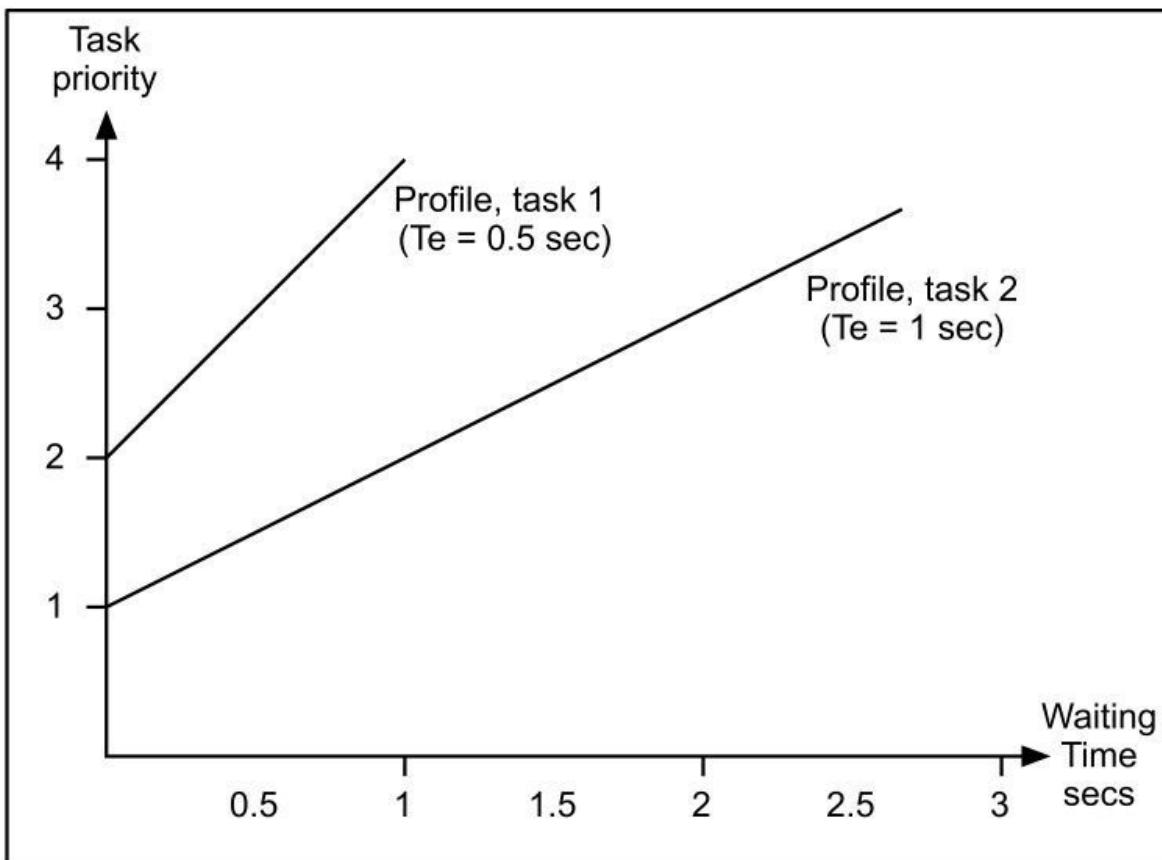


Figure 9.4 Example HRRN task priority profile

HRRN is unlikely to be used in fast real-time systems. However it provides a simple way to introduce three important points:

- The a priori setting of initial task priorities - a static schedule.
- The adjustment of these priorities as the system runs - a dynamic schedule.
- Priority profiles.

These are further discussed in the topic of deadline techniques.

(c) Cooperative scheduling.

With cooperative scheduling the decision to allow a context switch is made by the running task itself. Moreover, such switches are made at very specific program points. In doing so the executing task may:

- (i) Designate the next task to be set running or
- (ii) Return control to the scheduler, or

- (iii) Return control to the scheduler together with task synchronization information (typically using signals).

Both (i) and (ii) are very easy to implement if your programming language supports coroutines. Coroutine structures have been applied to good effect in soft real-time systems. They have also been used as sub-policies within pre-emptive schedulers. One feature of this technique is that system behaviour is much more predictable - particularly when all tasks are periodic ones. As such the policy may, when applied with care, be especially suitable for use in critical systems (even in hard-fast ones). However, without this language feature it may be easier to apply other methods.

Situations sometimes arise which require high throughputs together with high processor utilization. Where other policies fail to deliver the required performance approach (ii) may be used. An essential aspect here is to eliminate the need to save task information at context switch time. Clearly this calls for some 'hand-crafting' of the source code, paying great attention to detail. Normally this method would only be used as a last resort; upgrading the processor hardware is likely to be much more cost-effective. Unfortunately this isn't always possible. The good news is that impressive performance improvements have been achieved using this technique (personal correspondence).

9.3 Priority-based pre-emptive static scheduling policies - general.

In static priority scheduling schemes, task priorities are set at compilation time. It therefore is the responsibility of the programmer/designer to establish and define such priorities. Once set, these are not usually changed at run-time.

Four criteria may be used to arrive at priority settings. Three are objective, the fourth being subjective. The objective criteria, all based on time, are:

- Response times - shortest response time, SRT, gives highest priority (SRT scheduling).
- Computation times - shortest execution time gives highest priority (shortest job first; SJF scheduling).
- Periodic execution times (periodicity) - shortest period gives highest priority (Rate Monotonic scheduling, also called Rate Monotonic Analysis, RMA)

By contrast, designers often devise task schedules based mainly on experience and judgement. This subjective heuristic approach normally takes into account timing and criticality factors.

Let us see how the objective criteria work when applied to an example task set, figure 9.5.

Task	Type	Response time (milliseconds)	Computation time (milliseconds)	Period (milliseconds)
1	Periodic	20	10	100
2	Periodic	18	15	120
3	Aperiodic	110 - Deadline	5	-
4	Aperiodic	5 - Deadline	2	-

Figure 9.5 Static priority pre-emptive scheduling - example task attributes.

The resulting task schedules are shown in figure 9.6.

Scheduling policy →	Response time (SRT)	Computation time (SJF)	Periodic time (Rate Monotonic)
Task ordering → (priority listing, highest to lowest)	4 - Highest	4 - Highest	1 - Highest
	2	3	2
	1	1	?
	3	2	?

Figure 9.6 Static priority pre-emptive scheduling - example task schedules.

(a) Priority as a function of response times.

With SRT, the shortest task is given highest priority, calculated by:

$$P = 1/T_r$$

While this criterion is easy to apply it is, in practice, less objective than it seems. Required response times are often a matter of opinion, especially in the area of human-machine interfacing.

(b) Priority as a function of computation time.

This scheduling policy, in essence, is that of shortest-job-first (SJF) used in a pre-emptive mode (also called Shortest Remaining Time scheduling). Priorities are calculated as:

$$P = 1/T_e$$

One major problem with this scheme is being able to define computation times accurately. This is especially true in the early development stages of a project.

(c) Priority as a function of periodicity.

With this technique, tasks having high repetition frequencies (low periodic times) are assigned high priorities. The theoretical groundwork for this was done by Liu and Layland, where the policy was defined to be 'rate-monotonic priority assignment'. Unfortunately, we are unable to assign priorities to tasks 3 and 4 using the basic method - they are not periodic. More of this in a moment.

It is frequently claimed that rate monotonic scheduling offers superior performance. Moreover it is also supported by various vendors (e.g. Aonix), provided as a component within their design environments. As such it is important enough to warrant a section to itself

9.4 Priority-based pre-emptive static scheduling - rate monotonic scheduling.

First, a recap. Rate monotonic scheduling (also more commonly known as rate monotonic analysis, RMA) sets task priorities according to their periodicity, as follows:

$$P = 1/T_p$$

The task with the shortest period (thus highest priority) is placed at the front of the ready queue. Next in line is the task having the second shortest period; following this is the one with the third shortest period, and so on to the end of the queue. Thus as we go down the queue task periods always increase - a monotonic sequence. Central to RMA scheduling theory are the following assumptions:

- All tasks are periodic.
- A task's deadline is the same as its period.
- Tasks may be pre-empted.
- All tasks are equally important - criticality doesn't enter the equation.
- Tasks are independent of each other.
- The worst-case execution time of a task is constant.

A fundamental question in scheduling is 'is the given task schedule feasible?'. That is, will the system meet its requirements within the specified time frame(s)? Clearly a heavily loaded processor - one having high utilization - is less likely to succeed than one which is lightly loaded. Before discussing this further we need to be careful with the use of the word utilization (U). In the basic RMA algorithm it is defined as the percentage of available processor time spent executing tasks, figure 9.7. Here task 1 has an execution time of 30 milliseconds; its period is 100 milliseconds. Task 2 has an execution time of 20 milliseconds and a period of 200 milliseconds. The relevant utilization factors are:

- (i) Task 1 only scheduled - $U = 0.3$.
- (ii) Task 2 only scheduled - $U = 0.1$.
- (iii) Both tasks scheduled - $U = 0.4$

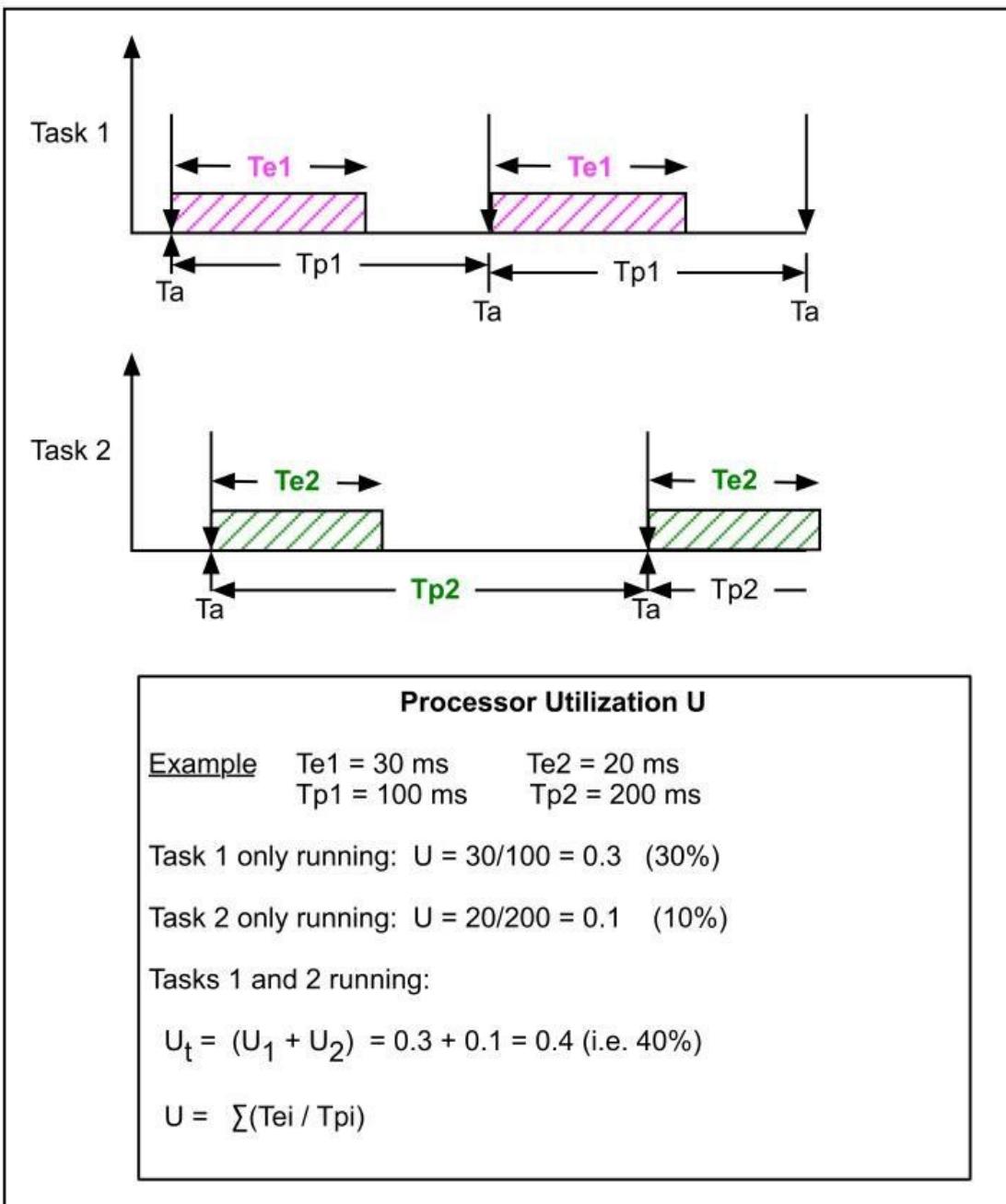


Figure 9.7 Rate monotonic analysis - definition of processor utilization.

A second term that needs explaining is 'full utilization' of the processor. This, you might think, is when the processor is fully occupied executing software, having no spare capacity (i.e. $U = 1.0$). Not so (at least where RMA is concerned). Here it describes a very specific situation: where a given task schedule is feasible but any increase in execution times will lead to failure. We might, for instance, find that only 80% of the processor time is spent executing

tasks; yet any attempt to use the spare capacity will cause task(s) to miss deadlines. Take it on trust that for a given processor having:

- A fixed set of tasks and
- A fixed total computation time for all tasks.

the 'full utilization' figure is not a fixed value. It depends on the individual task timings and the relative activation time of tasks. Liu and Leyland showed that the lowest figure for n tasks is:

$$U = n (2^{1/n} - 1)$$

As n increases the value of U decreases, eventually having a value of 0.693 when n is infinite. This means that if more than 69.3% of the processor time is spent executing tasks, we may not meet our deadlines (i.e. there is no guarantee that the schedule is feasible). More usefully put; a task set scheduled using RMA is guaranteed to be feasible provided its utilization does not exceed 0.693.

This seems fine; unfortunately it contains three major weaknesses where real-time systems are concerned. These are:

- It cannot deal with aperiodic tasks.
- Task deadline and period are considered to be synonymous.
- Tasks are independent, i.e. non-interacting.

Let us take these items in turn.

(a) Aperiodic tasks are present.

There are two ways to handle aperiodic tasks. First, turn them into periodic tasks by using polling techniques. Second, provide computing resources to deal with the random arrival of an aperiodic task within a predefined period. This is called an 'aperiodic server'.

Let us return to the example set of tasks of figure 9.5 and modify it to show all tasks as periodic ones, figure 9.8.

Task	Type	Response time (milliseconds)	Computation time (milliseconds)	Period (milliseconds)
1	Periodic	20	10	100
2	Periodic	18	15	120
3	'Periodic'	110	5	150
4	'Periodic'	5	2	250

Figure 9.8 Rate monotonic analysis - example task attributes.

This has assumed that task 3 might be re-readied, in the worst case, every 150 milliseconds. This 'interval time' between (possible) task executions is chosen to be its periodic time.

For task 4, 250 milliseconds has been used.

Of course we still have to decide on the priorities of the aperiodic tasks. The simplest way is to use the interval time to calculate priorities, as shown in figure 9.9, column 1.

1 For aperiodic tasks:		2 For aperiodic tasks:		3 For aperiodic tasks:	
Period set by <i>-Interval time</i>	Priority set by <i>Period</i>	Period set by <i>Response time</i>	Priority set by <i>Period</i>	Period set by <i>Interval Time</i>	Priority set by <i>Deadline</i>
Task → Period	Task priority	Task → Period	Task priority	Task → Period	Task priority
T1 → 100ms	1	T1 → 100ms	2	T1 → 100ms	2
T2 → 120ms	2	T2 → 120ms	4	T2 → 120ms	4
T3 → 150ms	3	T3 → 110ms (Td = 110ms)	3	T3 → 150ms (Td = 110ms)	3
T4 → 250ms	4	T4 → 5ms (Td = 5ms)	1	T4 → 250ms (Td = 5ms)	1

Figure 9.9 Rate monotonic analysis - handling aperiodic tasks.

If we set all tasks ready at time t_0 , then task 3 will complete at $(t_0 + 30)$ milliseconds - well within its deadline. Unfortunately task 4 won't finish until $(t_0 + 32)$ milliseconds, much too late. This can be alleviated by using a different set of rules in defining the 'period' of an aperiodic task; use the task response time (or deadline - they're the same in this case) as the period (column 2, figure 9.9). In this particular case the schedule will work, even though task 4 is activated every 5 milliseconds. Unfortunately such high-rate operation dramatically

increases processor utilization; furthermore, in many cases it may generate infeasible schedules.

A technique developed to overcome this drawback is the deadline monotonic scheduling policy. With this, aperiodic tasks are treated as periodic tasks, periods being based on interval times (figure 9.9, column 3). However priorities are based on deadlines (for periodic tasks, remember, period and deadline are the same thing). As a result task 4 gets activated only every 250 milliseconds, but when it does it takes on highest priority. As a result it is guaranteed to run to completion without being pre-empted.

Polling, as a technique to handle random inputs, has a number of drawbacks:

- Excessive overheads - polling whether or not an event occurs.
- Staleness of acquired data - time between an event occurring and it being recognised by the software.
- Skew between input signals - simultaneous event signals may, as a result of the polling action, have significantly different time stamps.

The periodic server attempts to alleviate these problems by not polling; instead it builds time into the schedule to handle random inputs should they occur. One technique allocates a fixed amount of processing time in a predefined time slot for the aperiodic tasks. Once this is used up no further aperiodic processing can be done until the beginning of the next time slot.

(b) Task deadline and period are not necessarily synonymous.

In practice it is unlikely that the execution time of a task is constant from run to run; a number of factors are likely to produce variations (e.g. conditional actions, task synchronization, contention for shared resources, etc.). For many applications this isn't a problem. Take the case of actively refreshing an alphanumeric display panel. Provided a sensible update period is chosen, significant variation of actual updating times can be tolerated (remember they are always bounded by the period of the task). However in other systems such variations may lead to performance degradation (for example jitter in closed-loop control systems). Most crucially, in some cases (e.g. manufacturing processes) they may lead to system malfunction or even total failure. Thus, in situations where timing is critical, other scheduling methods may have to be used.

(c) Tasks are frequently interdependent.

In general tasks are likely to be interdependent rather than independent. Interactions take place for three reasons:

- Synchronization of operations.
- Communication of information.
- Access to shared resources.

All can lead to so-called blocking conditions. The first two items are quite difficult to deal with theoretically; this is especially true when aperiodic tasks are involved. Mathematical analysis of such interactions tends to be limited to somewhat simplistic situations (perhaps realistic structures are too difficult to analyse?).

Where tasks share resources, blocking may be produced by the activation of protection (mutual exclusion) mechanisms. Rate monotonic theory has been extended to cover this situation where a ceiling priority protocol is used. In such cases a high-priority task has to wait at most for only one lower-priority task to finish using a protected resource. If the maximum time that such blocking can occur is T_b (i.e. when the low-priority task has locked the resource), then the modified RMA algorithm for n tasks is:

$$\sum_{i=1}^n [(T_{ei}/T_{pi} + T_{bi}/T_{pi})] \leq n(2^{1/n} - 1)$$

(Note: T_{bi} is the blocking time encountered by task i)

9.5 Priority-based pre-emptive static scheduling - combining task priority and criticality: a heuristic approach.

The methods described earlier have one major weakness - they ignore the criticality of tasks. Consider tasks having the timing attributes defined in figure 9.10:

Task	Type	Response time (milliseconds)	Computation time (milliseconds)
1	Aperiodic	5	4
2	Aperiodic	8	6

Figure 9.10 Task priority and criticality - example task attributes.

Using the criteria of time, task 1 is always assigned the highest priority. Thus if both tasks get readied at the same time, task 2 will be blocked out until task 1 has executed. But what of the precise nature of the tasks? We have said nothing about this. Consider that both tasks are part of a flight control system. The function of task 1 is to remove data from a data comms (serial line) buffer. Task 2 is part of a protection system within the terrain-following subsystem. It is activated if there is a failure of the altitude sensor - its function is to put the aircraft into a safe mode. Using an adapted RMA algorithm based on deadlines, task 1 will be given the highest priority. Which is the more important task? In this situation we have a clear-cut case - task 2. However, if both tasks get readied at the same instant then task 2 will go late. An even worse situation can arise if the control system happens to have a number of messages streaming in from various other systems. If this leads to degradation of performance (or worse, physical damage) the situation is unacceptable. Task 2 must be assigned the highest priority.

In practical systems the issues are rarely so clear-cut. It takes experience and judgement to arrive at a sensible priority scheme, backed up by historical real-time performance data information. But even then the actual performance may well differ from the predicted one.

9.6 Priority-based pre-emptive dynamic scheduling policies - general.

These dynamic scheduling policies have one simple objective: to maximise performance by making scheduling decisions based on the actual (current) system state. Priorities are assigned:

- In accordance with criteria based on time factors and
- On the fly, as tasks are executed.

The approaches used are also categorized as 'deadline scheduling'. Here the programmer does not specify task priorities. Instead, when tasks are created, specific attributes are defined in the source code. These are subsequently used by the scheduler to dynamically adjust priorities.

There are a number of different approaches, discussed in detail later. All take into account - in some shape or form - the following timing information (see figure 9.11):

- Task execution (or computation) time, T_e - a predefined value.
- Required response time, T_r - a predefined value. This is also called the 'due time'.
- Spare time, T_s - a computed value (also called the 'laxity').
- Required completion time (deadline), T_d - a computed value.
- Task activation (arrival) time, T_a .

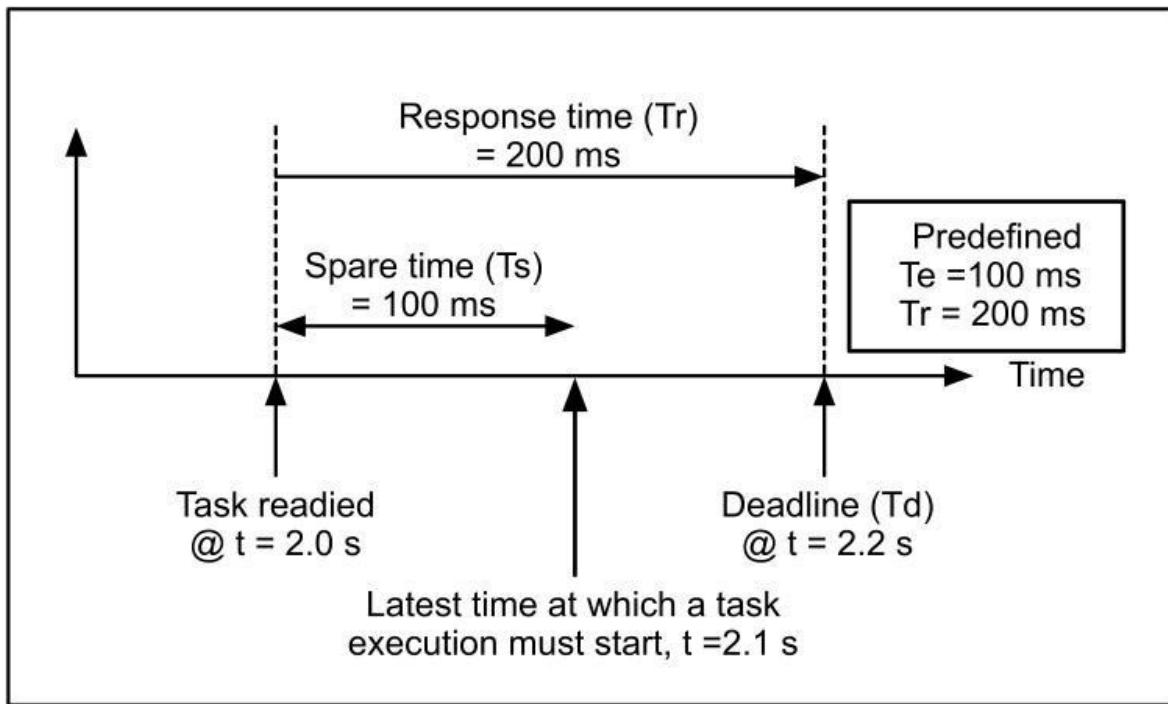


Figure 9.11 Definition of timings in deadline scheduling - 1.

As shown in Figure 9.11 a task is readied at time $T_a = 2.0$ secs. It has a required response time T_r of 200 ms. Thus its deadline T_d falls at $t = 2.2$ secs. Its predefined computation time T_e is 100 ms; consequently we have 100 ms of spare processing time T_s before the deadline expires.

When a task has been partly executed, additional timing data is of interest; all are computed values (figure 9.12):

- Amount of task execution currently completed, T_{ec} - a computed value.
- Amount of task execution left, T_{el} - a computed value [$T_{el} = (T_e - T_{ec})$].
- Time to go to deadline, T_g - a computed value.

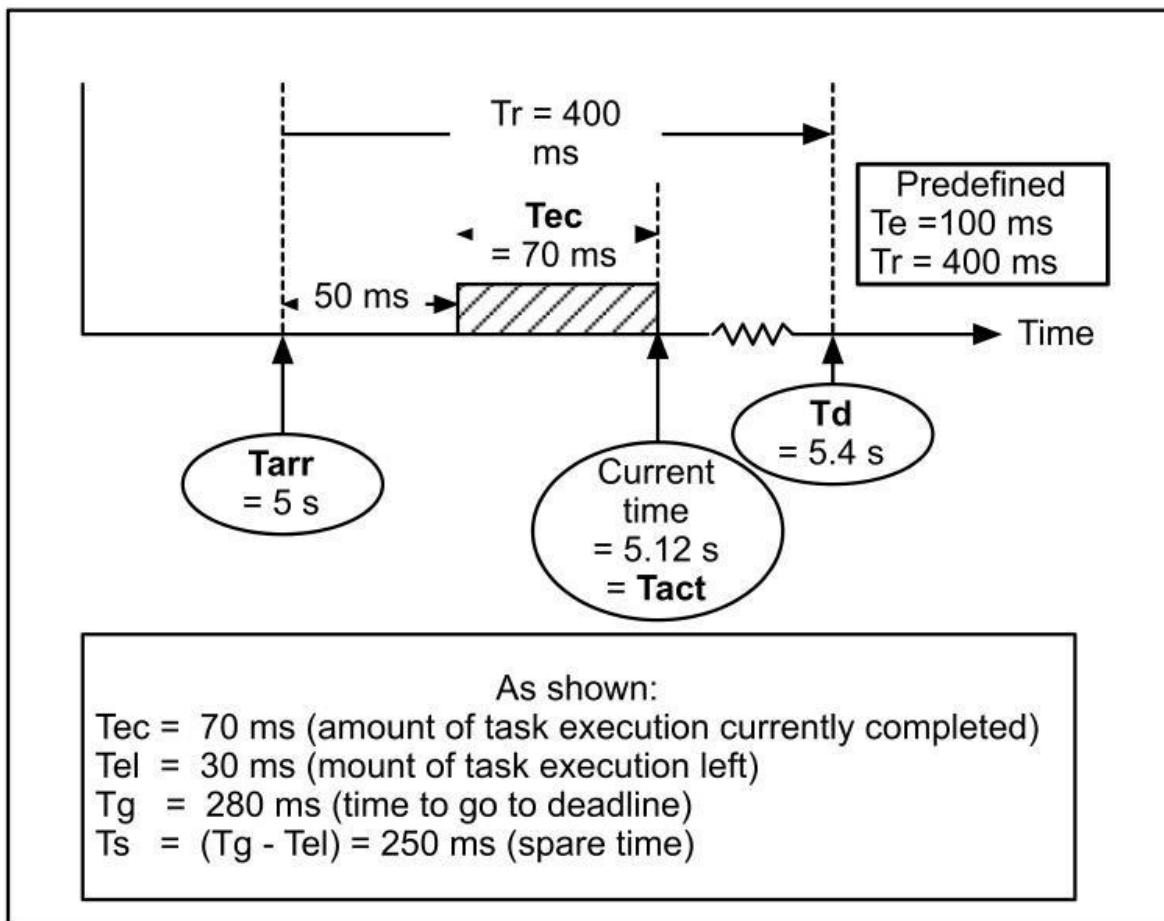


Figure 9.12 Definition of timings in deadline scheduling - 2.

In figure 9.12 the example task also has a predefined computation time of 100 ms and a required response time of 400 ms. At time = 5.0 seconds it was readied. The current time is 5.12 secs. As shown, it has been executing for 70 ms (Tec), performed after an initial delay of 50 ms. Thus the time to go to its deadline (Tg) is 280 ms, while the remaining spare time (Ts) is 250 ms.

Notes:

(i) When a task is readied ('activated')

$$T_d = (T_{actual} + Tr) = (T_a + Tr)$$

$$T_g = (T_d - T_{actual})$$

(ii) Tec is also called the *accumulated execution time*.

(iii) Ts is also called the *residual time*, calculated as $(T_g - Tel)$.

More strictly, we can define a task's deadline as the 'time at which the due-time expires'.

The decision to reschedule tasks can be based on numerous factors, three important ones being:

- The nearness of the deadlines (Earliest deadline scheduling).
- The amount of computation still to be done (Computation time).
- The amount of spare time remaining before we have to run the task (Laxity).

These are discussed below.

9.7 Priority-based pre-emptive dynamic scheduling - earliest deadline scheduling.

Assume that at re-schedule time we have the situation shown in figure 9.13.

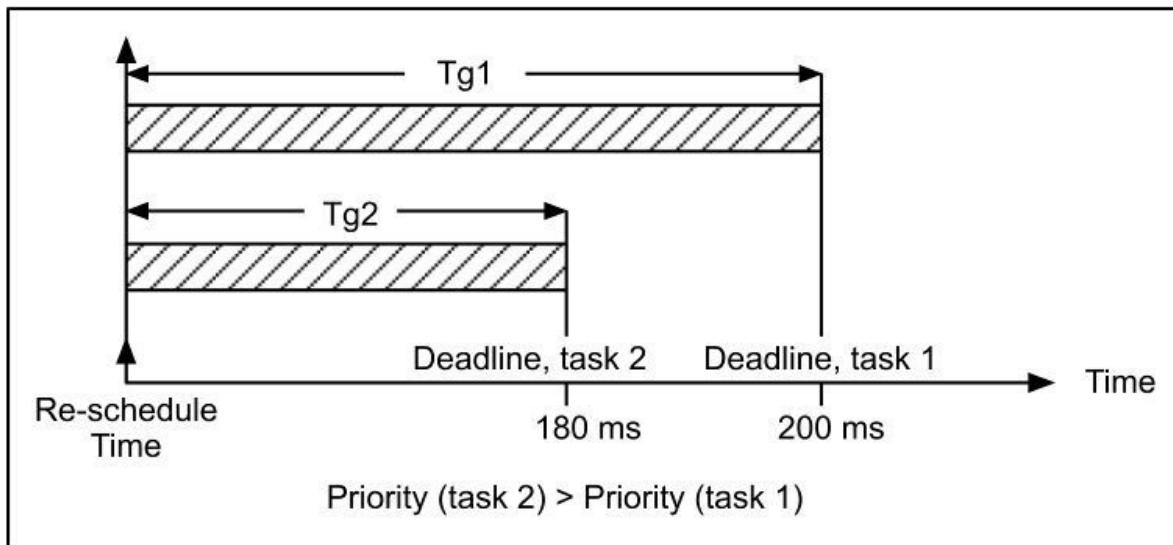


Figure 9.13 Priority as a function of due time.

Here two tasks are ready to be run - therefore we must decide which one. Using earliest deadline scheduling, the task which has the smallest (shortest) due time is run (in this case, task 2). That is:

$$P(\max) \Rightarrow \text{Task}(\min Tg)$$

This policy is also called the Earliest Due Date.

The priority profiles of both the running and the ready tasks depend on the scheduling algorithm being used. For example, the algorithm

$$P = 1/Tg \leq P_{\max}$$

(P_{\max} is a pre-defined value)

generates the profile shown in figure 9.14. Applying this to task 1 of figure 9.13 results in the actual profile of figure 9.15. Observe that P_{\max} has been set to 15 (an arbitrary choice).

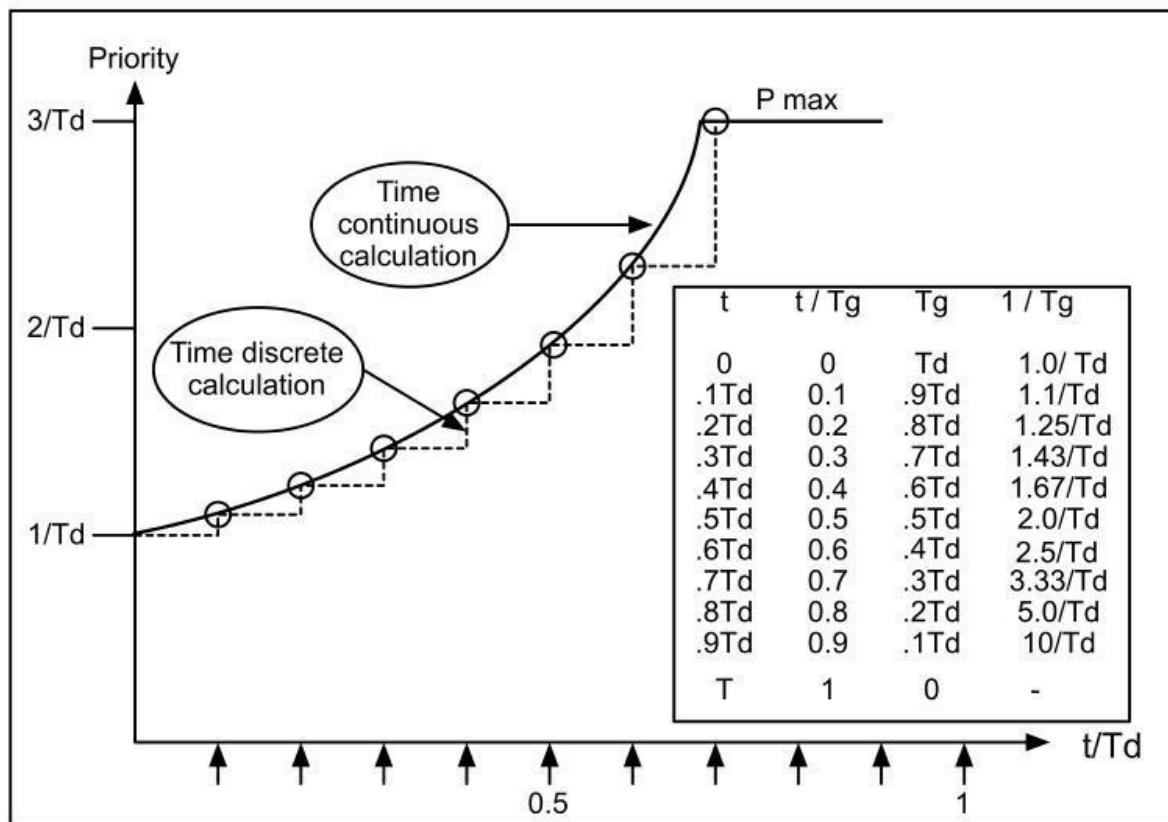


Figure 9.14 Earliest deadline scheduling, task priority profile - normalised

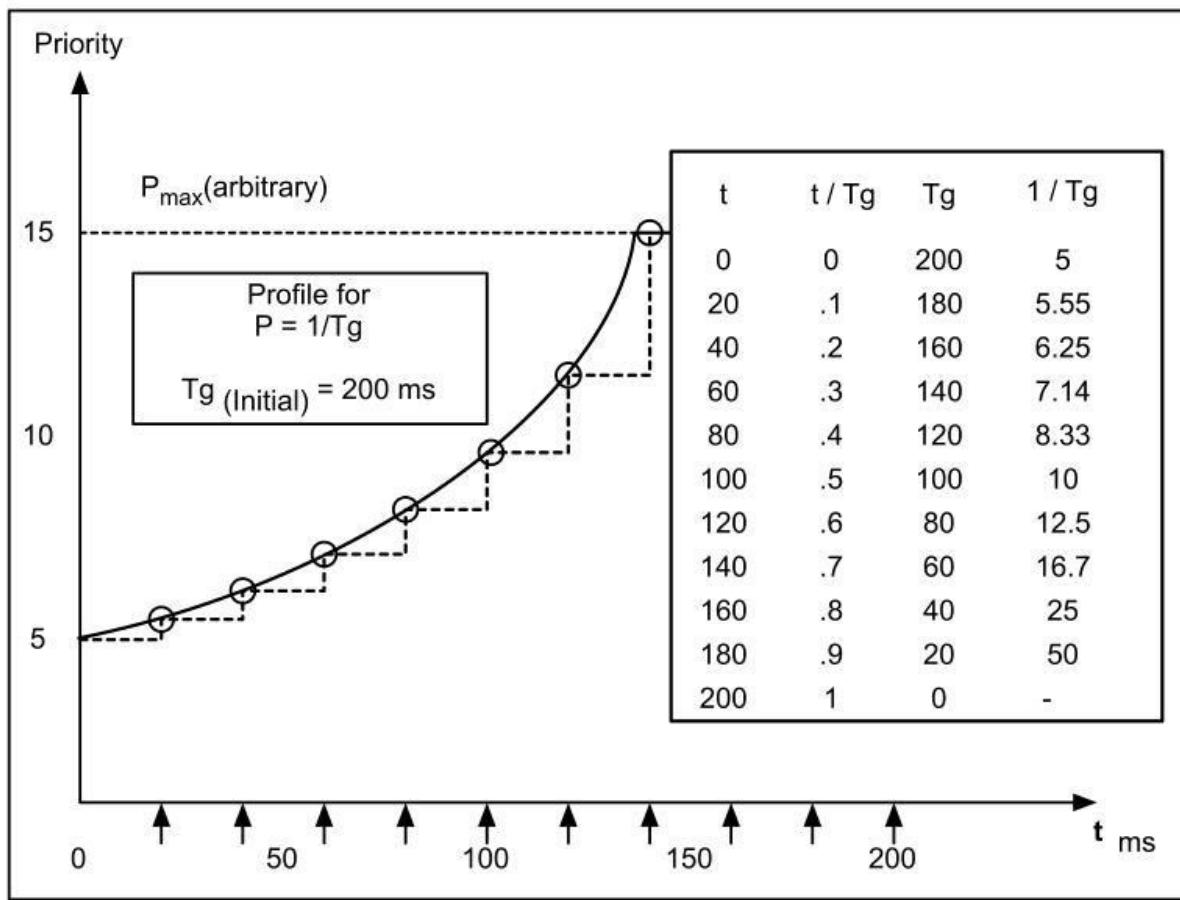


Figure 9.15 Example EDS task priority profile

A further simple illustration of the use of earliest deadline scheduling is given in figure 9.16. This describes the behaviour of a two-task system whose individual priority profiles are shown in Figure 9.16(a). Fig.(b) shows what happens if both are readied at the same time, a fairly straightforward case. In fig.(c) task 2 is readied first and then set running. At a later point task 1 is activated but, as it has a lower priority, remains in the ready queue. There comes a time, however, when its priority exceeds that of task 2. As a result task 2 is swapped out, being replaced by task 1. In this instance it will run to completion; only then will task 2 resume execution.

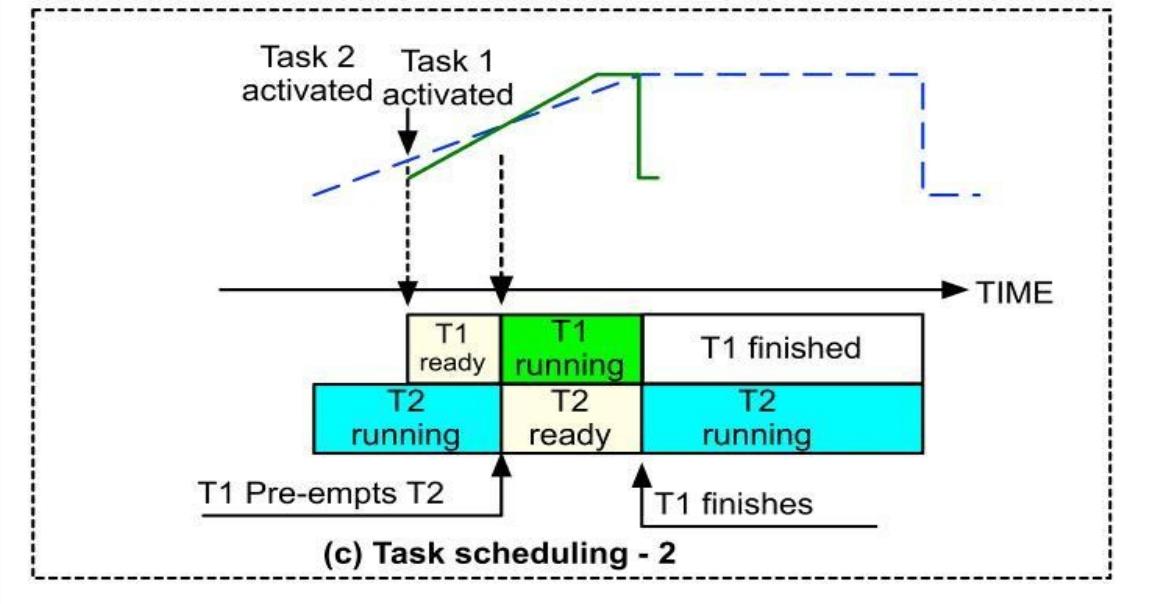
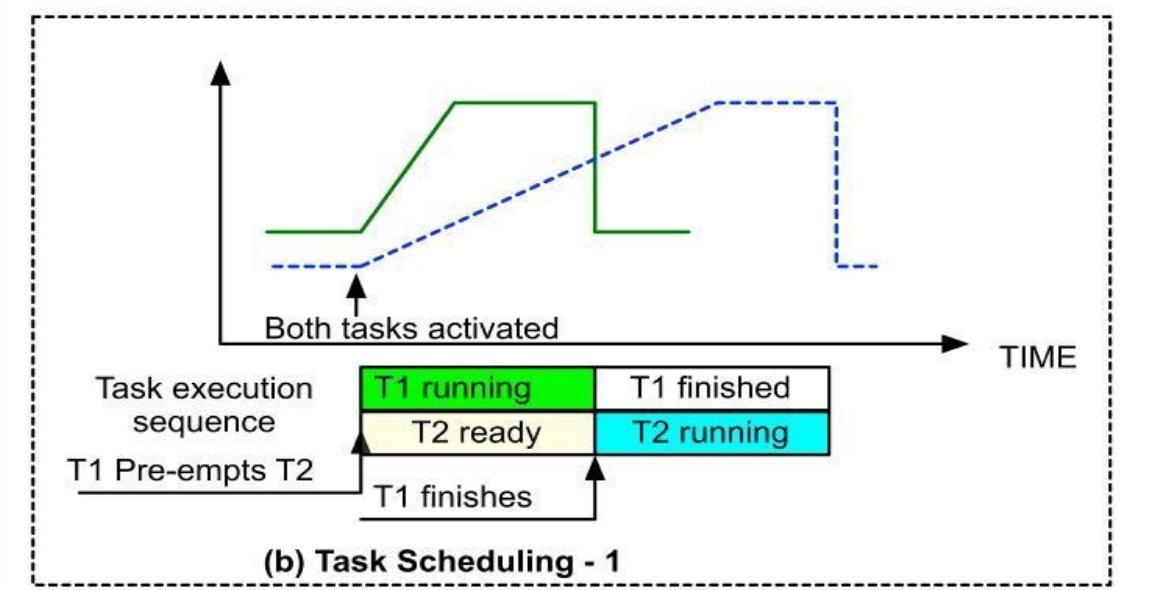
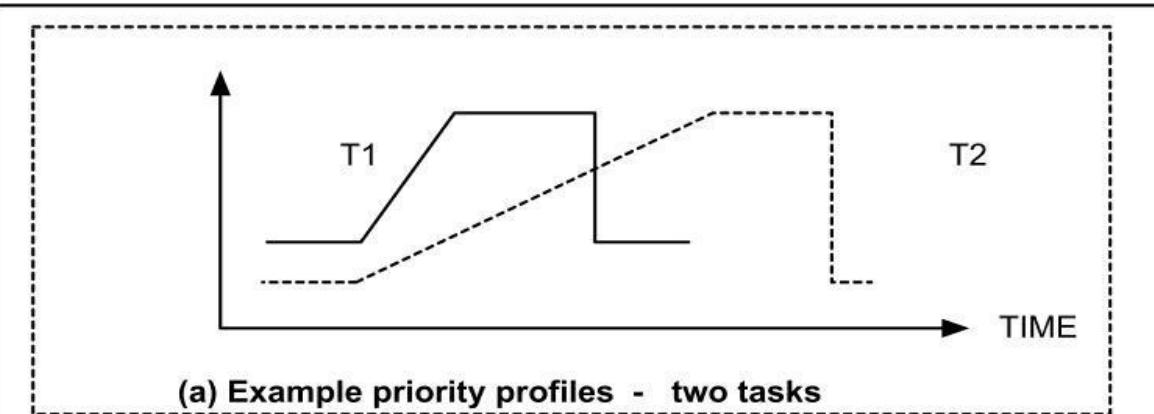


Figure 9.16 Task priorities and execution profiles in deadline scheduling - example

9.8 Priority-based pre-emptive dynamic scheduling policies - computation time scheduling.

The scenario given in figure 9.17 is the same as the one described in figure 9.13. Here, though, priority is set depending on the amount of computation still to be carried out within the tasks (Tel). As shown, task 1 needs 70 ms to complete, whilst task 2 requires 100 ms. We assign highest priority to the task which needs the least computation time to finish - in this case task 1. Therefore:

$$P(\max) \Rightarrow \text{Task}(\min \text{ Tel})$$

A workable algorithm is:

$$P = 1/\text{Tel} \leq P_{\max} \quad (\text{P}_{\max} \text{ is a pre-defined value})$$

It can be seen that this is a dynamic version of the (non pre-emptive) shortest job first policy.

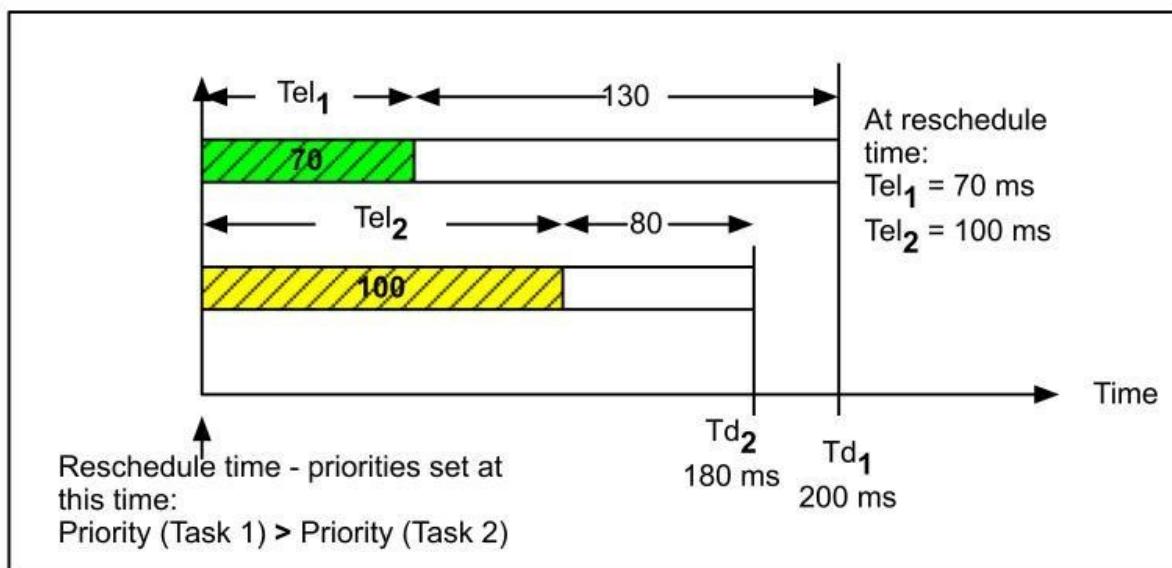


Figure 9.17 Priority as a function of time needed to complete task.

9.9 Priority-based pre-emptive dynamic scheduling policies - spare time (laxity) scheduling.

Laxity is the amount of time which can be 'wasted' between activating a task and having it deliver its result. It is defined as

$$Ts = [(Tr - Tel) - t]$$

where t is the elapsed time from the task being activated. Task priority P may be computed as follows:

$$P = 1/Ts \leq P_{max} \quad (P_{max} \text{ is a pre-defined value})$$

Figure 9.18 is a redrawn version of figure 9.17, emphasising the spare time (the laxity) of each task (Ts). In this instance task 2 has the lowest (least) laxity. Thus it is assigned highest priority.

This scheduling method is also called the 'Least Laxity First' (LLF) policy.

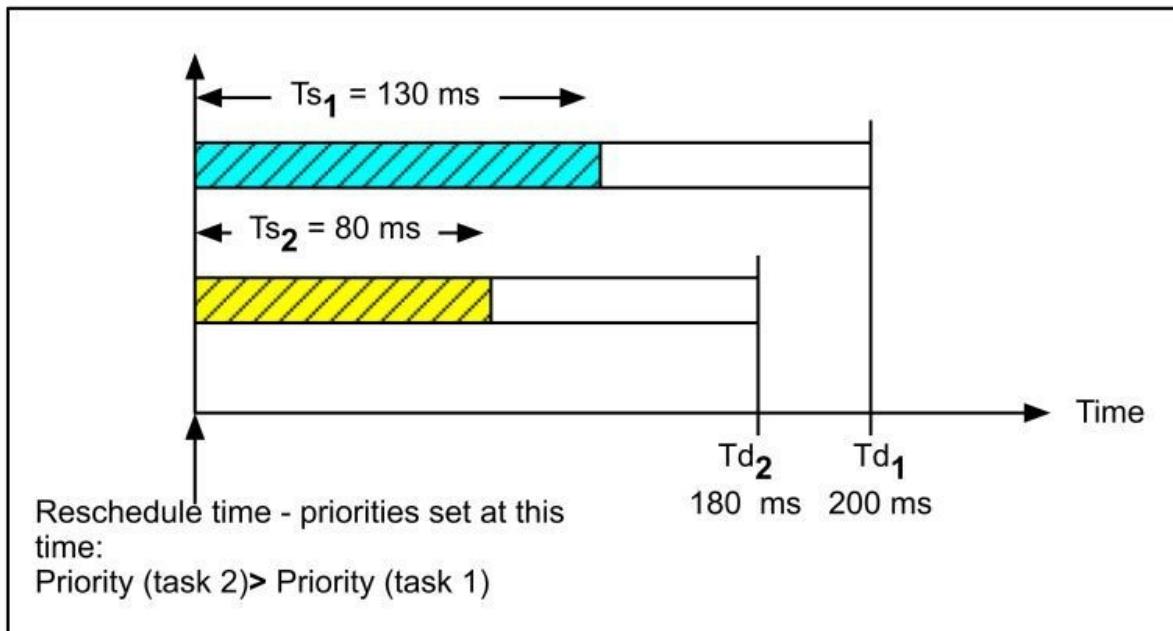


Figure 9.18 Priority as a function of task laxity (spare time).

9.10 Improving processor utilization - forming rate groups

It has been pointed out in earlier work that multitasking carries with it a penalty in terms of processor overhead. In some cases this may significantly reduce the processing time available to the application tasks. However, by minimizing the number of tasks in a system, processor efficiency can be improved. To do this we need to combine two or more tasks into an equivalent single task (the combined code is organized as a single sequential program). The resultant design must, of course, meet system objectives in terms of time and functionality. Let us look at how this can be done.

Suppose that an existing multitasking design consists of three periodic tasks, their details being (figure 9.19):

Task no.	Period (mSecs)	Run Time (mSecs)
1	20	8
2	39	8
3	81	4

Figure 9.19 Example task set.

In order to meet this requirement the tick timer must have a resolution of 1 ms. That is, it must interrupt at a 1 ms. rate. Clearly the resulting processor overhead is considerable, and may significantly degrade system performance. One technique which, in some circumstances, can reduce processor loading, takes the following approach:

- All task periods are set so that they have a simple numerical relationship to each other.
- Periodic tasks are executed at their correct periods and run to completion at each activation.
- Processor loading is spread evenly over time. This generally gives best performance in the presence of aperiodic tasks.

Consider the effect of changing task periods so that their timing relationships is a simple one, figure 9.20:

TASK No.	PERIOD (mSecs)	Run Time (mSecs)
1	20	8
2	40	8
3	80	4

Figure 9.20 Modified task set.

Task 2 now has a period which is twice as long as that of task 1. For task 3, the ratio is four. With this arrangement the tick resolution can be changed to 20 milliseconds.

What has been done is to organise processor execution as a series of minor cycle time slots. Each one has the same duration: that of the shortest period (in this case 20 milliseconds). Tasks are allocated to specific time slots, resulting in the run-time scheduling of figure 9.21(a).

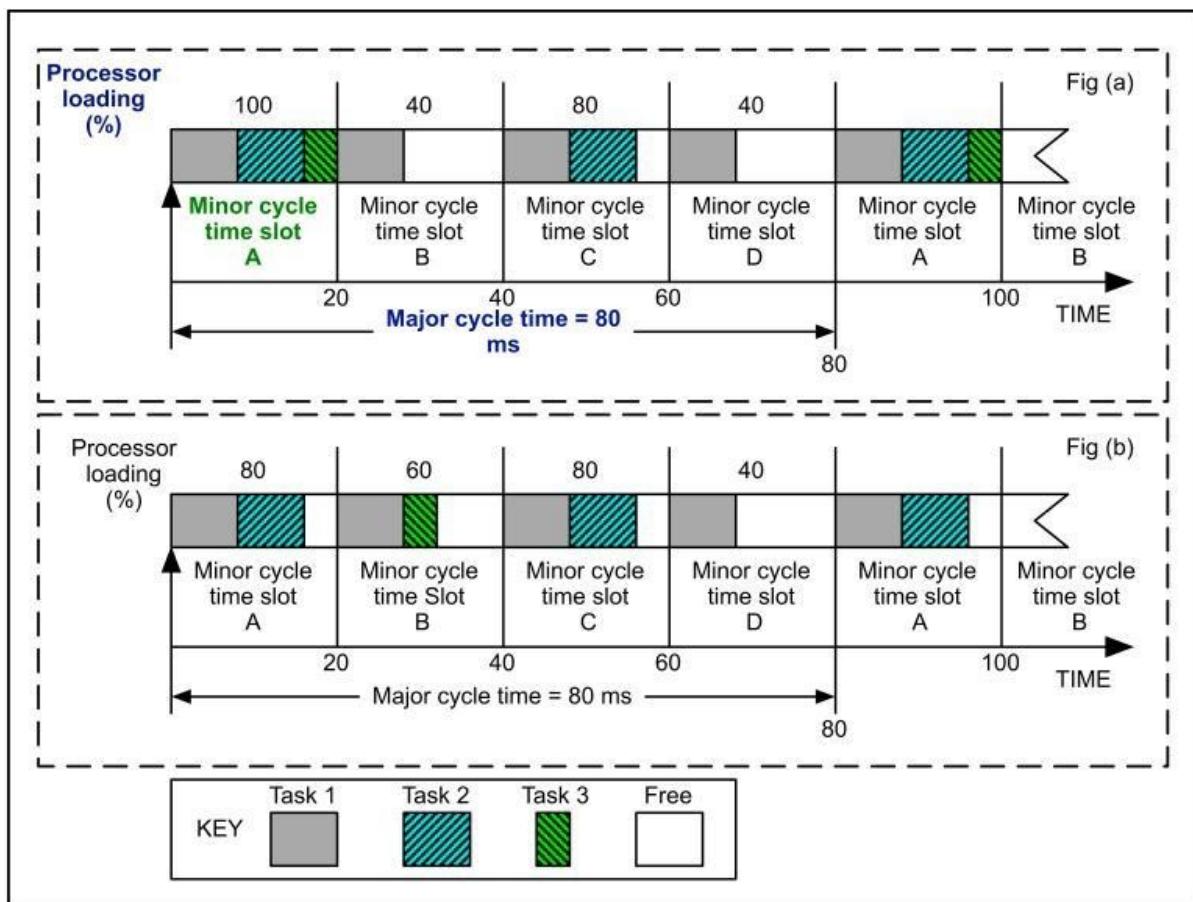


Figure 9.21 Run-time scheduling of tasks

The point at which the allocation pattern repeats defines the duration of the

major cycle time, here being 80 ms. You can see that a group of tasks are launched into execution within each time slot (leading to the expression rate groups). With this arrangement the code for all tasks is collected into one module, organised as a sequential program. At each tick time the scheduler control software merely decides which section of the code to execute.

One potential weakness with the schedule of figure 9.21(a) is that the processor loading is very uneven (ranging from 40% to 100%). As a result there could be problems if an aperiodic task arrives during a heavily loaded period. It may produce considerable disturbance, causing a number of tasks to go late. One way to alleviate this is to 'spread' the periodic tasks so as to even out loading variations, figure 9.21(b). Of course the effectiveness of the schedule depends very much on the nature of the aperiodic tasks. Unfortunately these, by their very nature, are not deterministic; they must be described in statistical terms.

9.11 Scheduling strategy - a final comment

Many scheduling strategies have been devised and put into service. But in the main these have been designed for commercial applications, being quite unsuited for real-time systems. Taking the constraints and features of real-time applications into account, the most widely used successful scheduling strategies follow a particular pattern (figure 9.22).

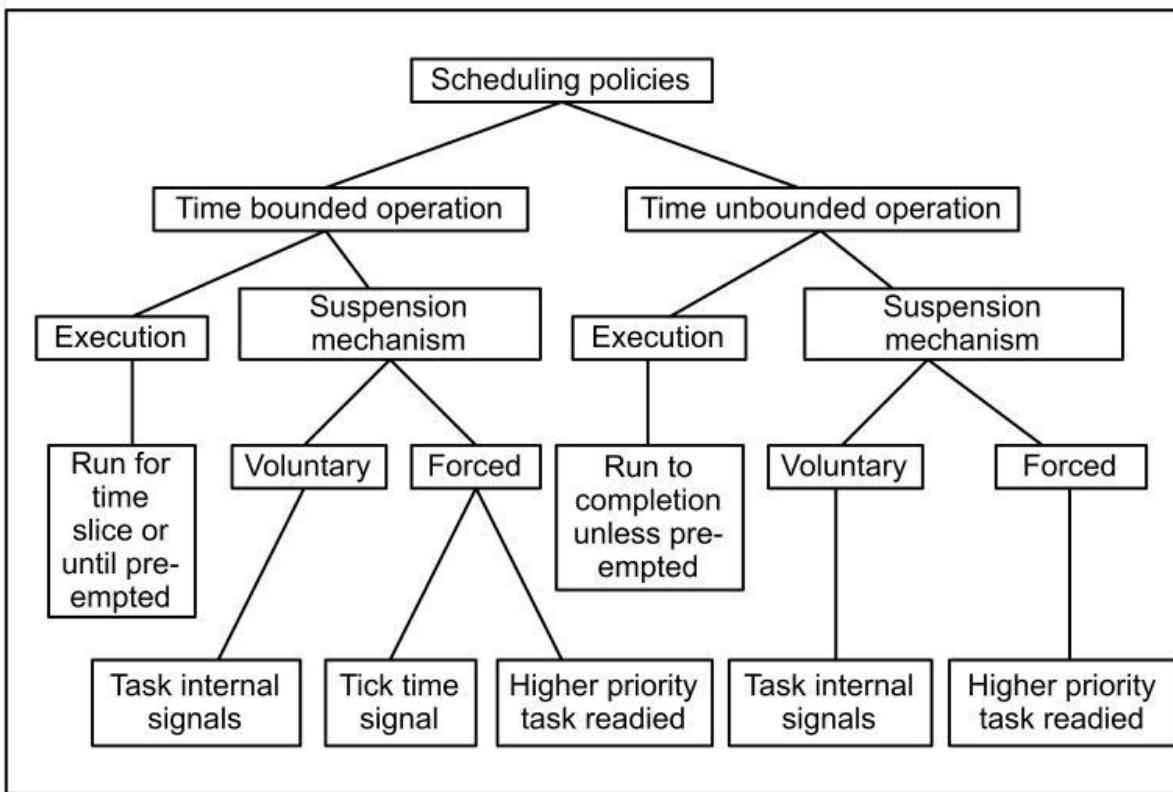


Figure 9.22 Embedded Systems Scheduling Strategy

First, tasks above base level are set in priority order. Once such a task is set running it goes to completion unless pre-empted by a higher priority task. It is resumed at the first opportunity.

The tasks, when in possession of the CPU (i.e. running), can lose it to the executive in two ways. In the first case it voluntarily gives up the CPU because it no longer wants or needs it, 'self-release'. This arises, for instance, where a process cannot proceed any further until some event occurs. It also happens where the task has completed and so the process has no further need for the CPU. The second reason for losing the CPU is a 'forced release' action. Usually this results from the readying of a higher priority task, forcing the current task to

give up the processor at the next reschedule point.

Base-level tasks are often run FIFO or round-robin, at equal priority settings. Both self and forced release mechanisms are used as described above, together with a third forced release mode. With round-robin scheduling, each task is forced to give up the CPU (even if it hasn't finished) when its allocated time slot expires.

9.12 Scheduling timing diagrams - list of symbols

- (Ta) Task arrival (activation) time: the time at which a task is put into the ready-to-run state.
- (Tb) Resource blocking time.
- (Td) Task deadline: the time by which the task must deliver its result.
- (Te) Execution time: the actual execution time of the task.
- (Tec) Amount of task execution currently completed: a computed value.
- (Tel) Amount of task execution left:
 - a computed value [Tel = (Te - Tec)].
- (Tg) Time to go to deadline: a computed value.
- (Tp) Task period.
- (Tr) Response time: the time allowed for a task to carry out its function (note: if timing is done on a relative, not absolute, scale, then response time and deadline have the same value). This is also called 'due time'.
- (Ts) Spare time or laxity:
 - a computed value [(Tr - Tel) - Telapsed].
- (Tw) Waiting time.

Review

You should now:

- Have a clear understanding of the range of scheduling policies suitable for use in real-time embedded systems.
- Understand the timing metrics used and their associated notation.
- Know the concepts of static and dynamic scheduling policies and understand their relative advantages and disadvantages.
- Know what is meant by SRT, SJF and rate monotonic scheduling.
- Understand what is meant by processor utilization in the context of rate monotonic scheduling.
- Recognize the practical problems of running both periodic and aperiodic tasks using rate monotonic scheduling.
- Know what the dynamic algorithms of earliest deadline, computation time and laxity scheduling are.
- Know what rate groups are, why they're used, and the advantages and disadvantages of using the scheduling method.
- Know the scheduling methods which have been employed extensively in real-time embedded systems.

Chapter 10 Operating systems - basic structures and features.

The objectives of this chapter are to:

- Show how embedded systems architectures influence the choice of operating system.
- Explain how interrupts can be used to provide a simple form of quasi-concurrency.
- Develop a structured software model of an interrupt-driven software system.
- Expand this model to illustrate the structures of practical nanokernel and microkernel RTOSs.
- Identify the essential ('core') software components of a practical RTOS.
- Describe the functions provided by these key components.
- Explain what a hardware abstraction layer of software is and why we use it.
- Describe the RTOS functionality found in larger embedded systems.

10.1 Setting the scene.

There is no such thing as a 'standard' real-time operating system; they come in a whole variety of shapes and sizes. Commercial offerings vary in code size from about 1 kByte to a few megabytes; RAM requirements are equally variable. Their functionality ranges from basic kernel operations through to full-blown Internet compatibility. And cost variations are equally bewildering.

In the light of this you can be forgiven for being confused; after all, how can such diverse products satisfy the multitasking needs of embedded systems? However, a moment's thought will show that you are asking the wrong question. It should be 'which product is best suited to my embedded system?'.

Embedded systems are extremely diverse; moreover they tend to be shaped very much by their areas of application, figure 10.1. Further, both technical and commercial factors determine the form, function and capabilities of real-time systems.

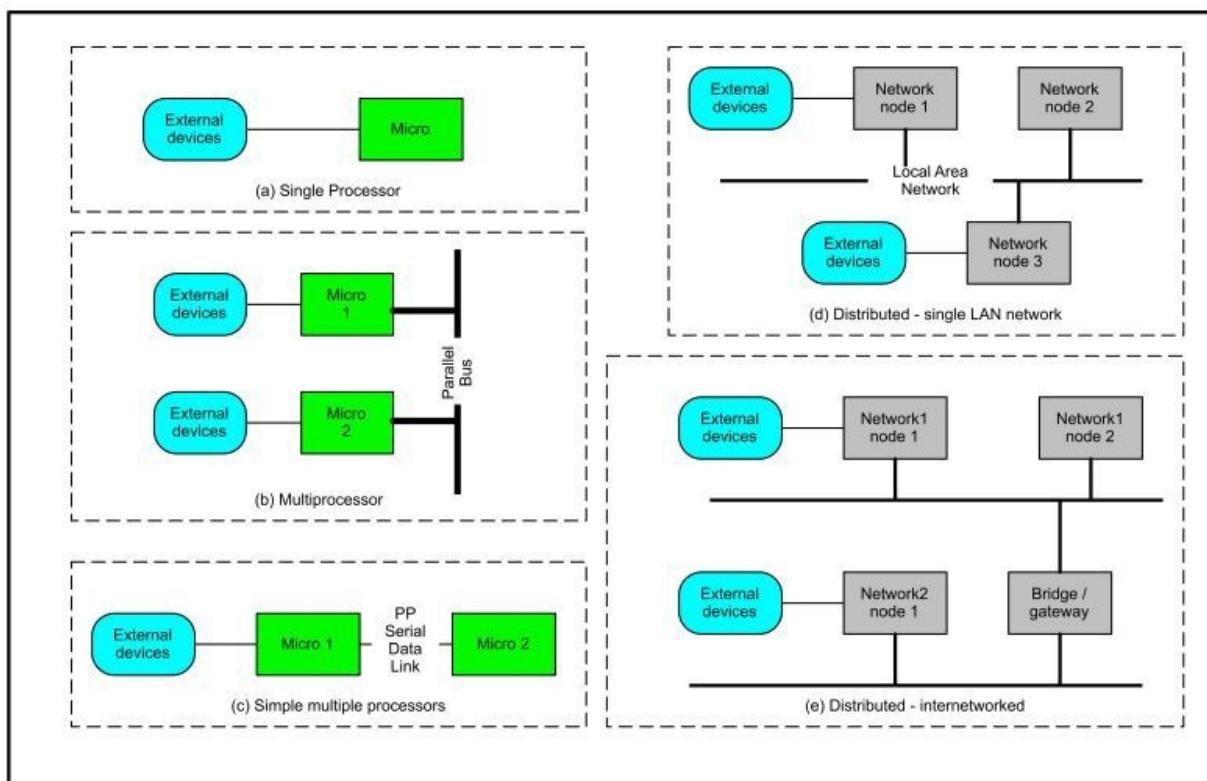


Figure 10.1 Embedded systems - some common architectures In this section we'll look at the need for and use of operating systems for the support of such architectures. Not surprisingly the 'one size fits all' RTOS approach is a somewhat poor engineering solution. For instance, with small, cost-sensitive

units (e.g. white-goods applications) an RTOS must also be small. Fortunately such systems are likely to need only limited functionality; thus a small OS should be sufficient. By contrast, applying the same RTOS to a complex control system - and expecting it to deliver the required performance - is rather optimistic (the words 'plug and pray' spring to mind). To help choose the best solution to our multitasking requirements we need to define the:

- Core functionality provided by an RTOS.
- Additional functionality needed to support more complex architectures.
- Component parts of both core and extended RTOS's.
- Structuring and interconnection of the various RTOS components.

Let us begin by seeing how a simple non-RTOS control system design might be implemented. The reasons for doing this are threefold:

- To identify key sections of the program code.
- To show how these sections relate to each other.
- To set a baseline for further work.

10.2 Simple multitasking via interrupts.

Assume that we are tasked with providing the software for a digital controller - a two-loop design - having the hardware structure of figure 10.1(a). The code for each control loop may be considered a task (in a multitasking sense, that is) needing to be executed at quite accurate intervals. One simple way to achieve this is to first write each 'application task' as an interrupt routine; then use hardware timers to generate the necessary interrupt signals. However, for this to work, the micro must be brought to a fully operational condition: that is, correctly and fully initialized. Thus the complete code for the system is likely to be organized as in figure 10.2. This, it can be seen, consists of three major groups: initialization code, background processing and application code.

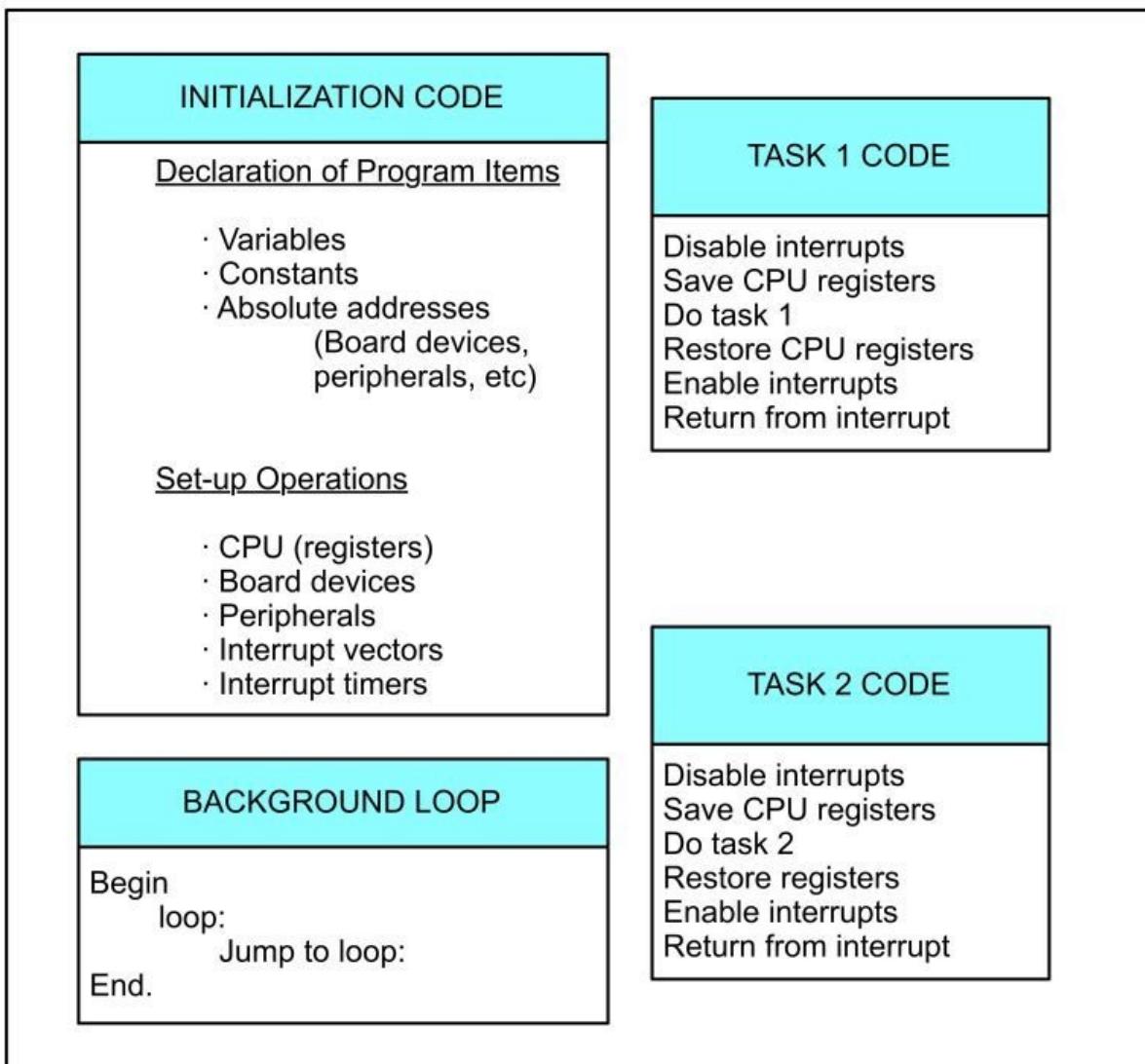


Figure 10.2 Code organization - simple single processor application Consider each in turn.

(a) Initialization code.

The purpose of initialization code is to bring the processor to a state where useful work can be done. There are two main sections: declarations and set-up operations. A key aspect of declarations is to define all items used within the program (strictly this is for use by the compiler). It is also essential to specify, where needed, the absolute address locations of such items. For simplicity, processor-related components such as timers, counters etc. are here called board devices. Components which interface to the outside world are denoted as peripheral devices.

Following this, forming part of the executable code, are the set-up operations. These ensure that the CPU, devices, peripherals etc. (i.e. those requiring programming) are correctly initialized and readied for use by the application software. In particular they must set up and activate the interrupt system.

(b) Background processing.

In this example the background loop is simply a continuous loop which does nothing. It would be a perfectly reasonable design decision to omit this altogether; merely terminate the code after initialization. However, most embedded designs do use a background loop, even if only for calculating processor utilization (in such cases the background loop is considered to be another application task). What is important is to recognize that the background loop runs at a lower priority than the interrupt-driven application tasks.

(c) Application tasks.

There are, as already described, two application tasks, each having similar code structures (figure 10.2). From this it can be seen that, in each case, the basic task code is surrounded by interrupt-specific code. Depending on the design it may also be necessary to include timer-related programming. To keep things clean, simple and maintainable, operations 'Do Task 1' and 'Do Task 2' would be calls to subprograms 'Task 1' and 'Task 2'. Both tasks will have different priorities - it is up to the designer to define the ordering in the initialization code (e.g. by appropriate set-up of a programmable interrupt controller).

Designs like this tend to be tackled to some extent in an ad-hoc programmer-dependent manner. Moreover, the resulting code tends to conceal the underlying conceptual model of the structure, figure 10.3.

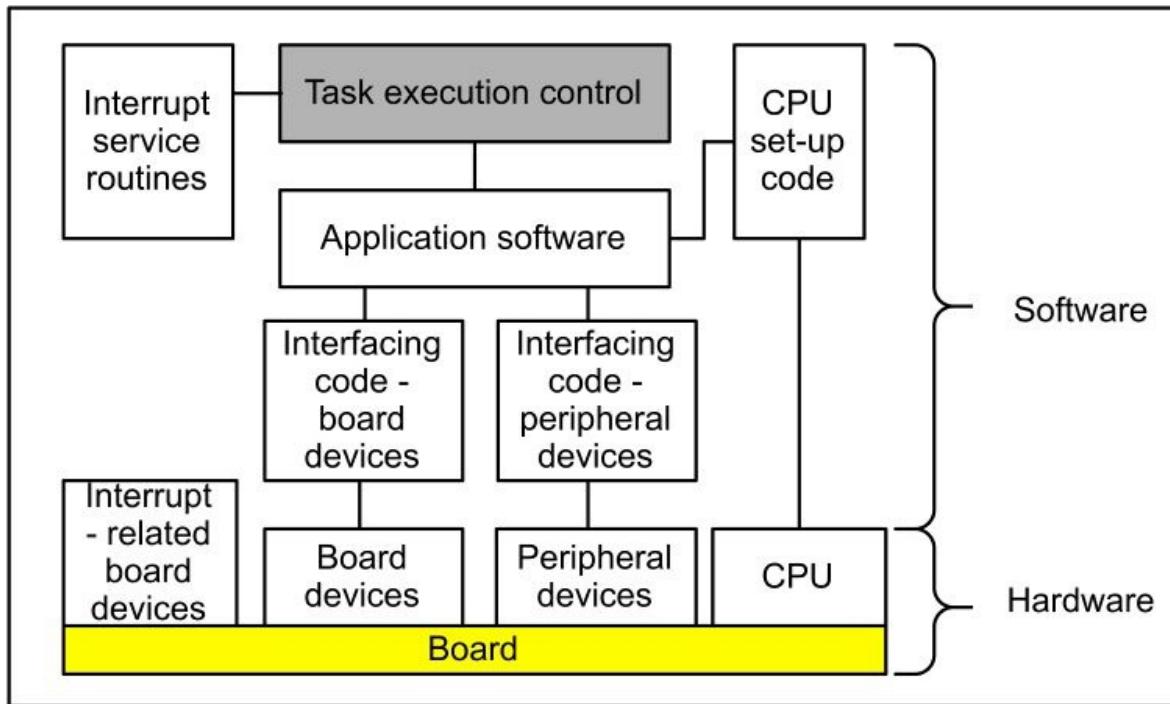


Figure 10.3 Conceptual model - simple single processor hardware-software structure Three points should be noted. First, control of hardware items - both for initialization and normal operation - is performed by the following software subsystems:

- Interrupt service routines, ISRs (for interrupt handling).
- Interfacing code, board devices (for control of standard on-board devices).
- Interfacing code, peripheral devices (for use with peripheral devices - usually non-standard or special-to-application).
- CPU set-up code (for manipulation of the CPU facilities).

Second, the application software interacts with the hardware via the interfacing code layer; in turn this runs under the control of the task execution control code. Third, task activation is carried out by ISRs, these being triggered by hardware-generated signals.

10.3 The Nanokernel.

It is only one small jump from an interrupt-driven home-grown 'tasking' design to the smallest type of RTOS, the nanokernel (more properly this should be described as a real-time executive). Furthermore, the word nanokernel isn't standardized; variations, sometimes quite major, can be found in practical designs. However, in this text it will be taken to mean a software component that provides a minimal set of OS services, namely:

- Task creation.
- Task management - scheduling and dispatching.
- Timing and interrupt management.

The conceptual model of a nanokernel-based system, figure 10.4, demonstrates two important features.

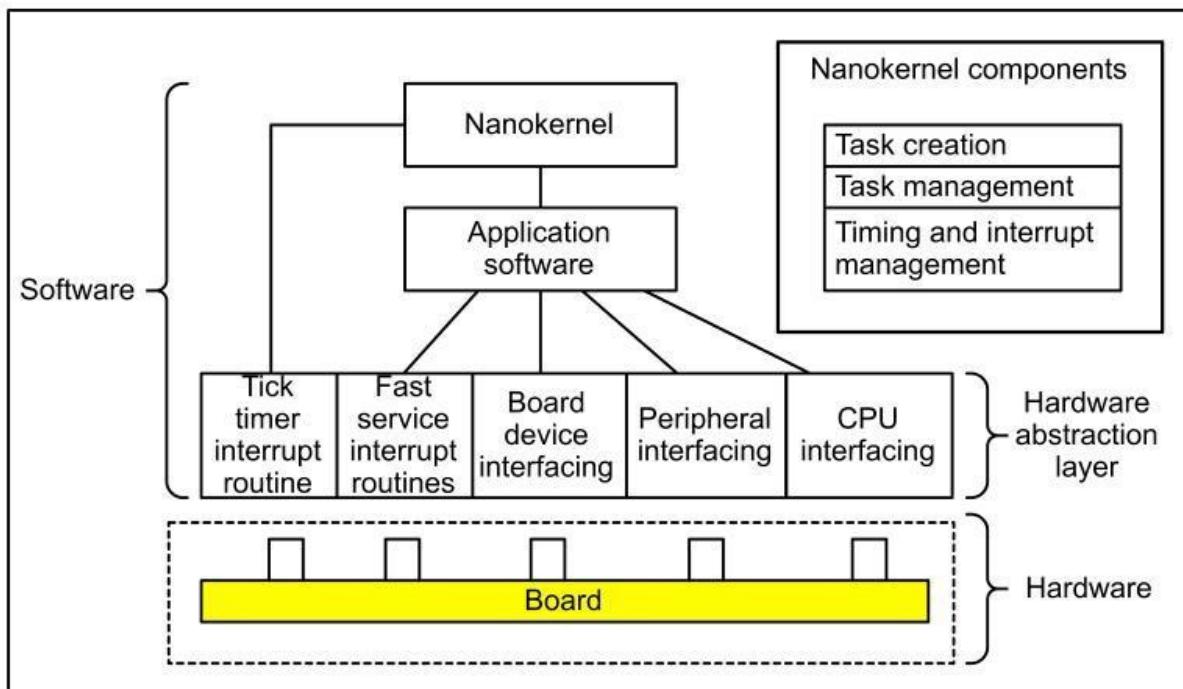


Figure 10.4 Conceptual model - nanokernel-based system First, all application tasks - except time-critical ones - are controlled by the kernel. Critical tasks require the fastest of responses, and cannot tolerate delays introduced by the OS. They are activated by the fast service interrupt routines which invoke immediate task dispatch (thus by-passing the kernel). A separate interrupt routine is provided for the tick timer, this providing the basic timing mechanism of the

nanokernel.

Second, the layer of software that interfaces directly to the hardware:

- Is intended to provide an abstract interface to the application software and the kernel itself.
- Should encapsulate all code needed for the set-up and operation of the board hardware.
- Depends on the nature of the computer hardware.
- Is usually produced by the programmer as a special-purpose design.

This software is sometimes called the Hardware Abstraction Layer, HAL.

The design of the HAL is entirely the responsibility of the programmer. A good design will provide strong encapsulation of this service software (so called as it provides a service to the application) together with the hiding of implementation details. The application programmer should merely have to call on these services without needing to know anything of their innards. Unfortunately, there is no guarantee that the software will be designed in this way; you may well find detailed service code within the application code.

An examination of the code organization (figure 10.5) shows that requirements for timing and interrupt handling have apparently disappeared. In fact they are contained within the kernel. The kernel code is usually composed as a set of subprograms encapsulated within some software container, often a monitor (which also provides safe operation as a result of its inherent mutual exclusion mechanism). Typically these enable us to:

- Initialize the OS.
- Create a Task.
- Delete a Task.
- Delay a Task.
- Control the Tick (start time-slicing, set the time slice duration).
- Start the OS.
- Set the Clock Time.
- Get the Clock Time.

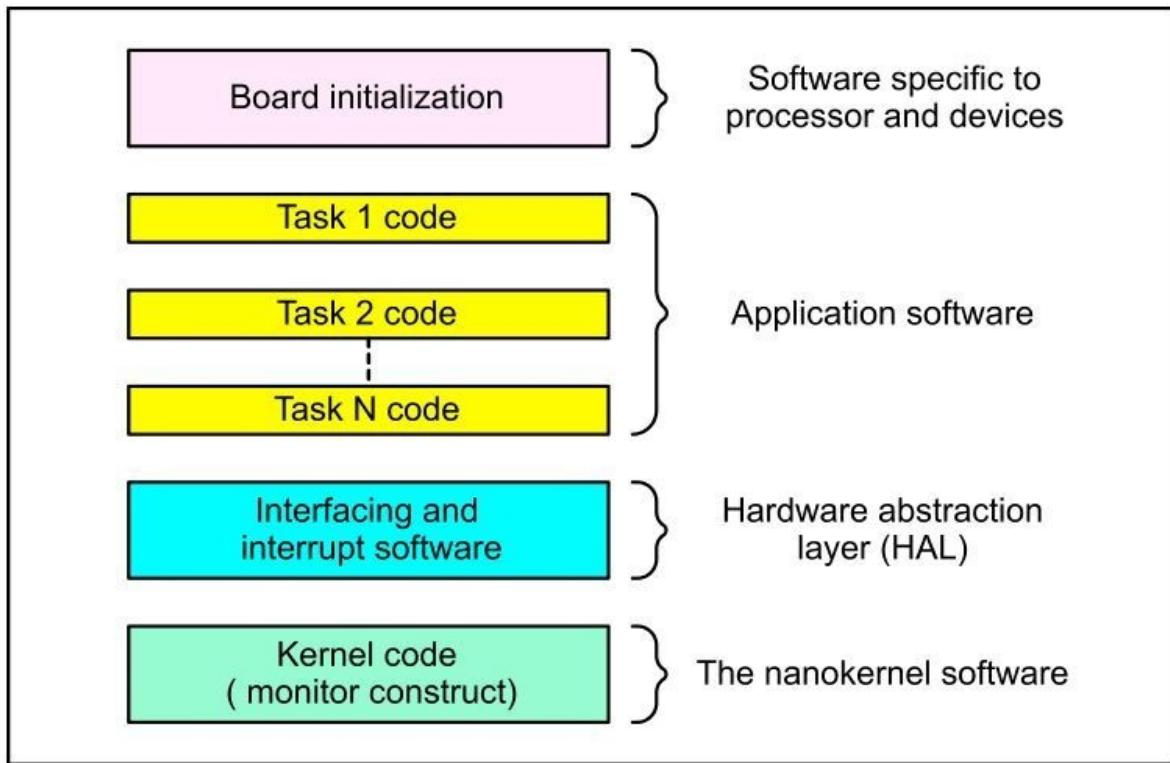


Figure 10.5 Code organization - nanokernel-based system

10.4 The Microkernel.

The next step up the OS ladder brings us to the microkernel. Its code is often organized in a manner similar to that of figure 10.6. There are two significant improvements over the nanokernel: increased kernel functionality and the use of a board support package (BSP). Let us turn first to the BSP.

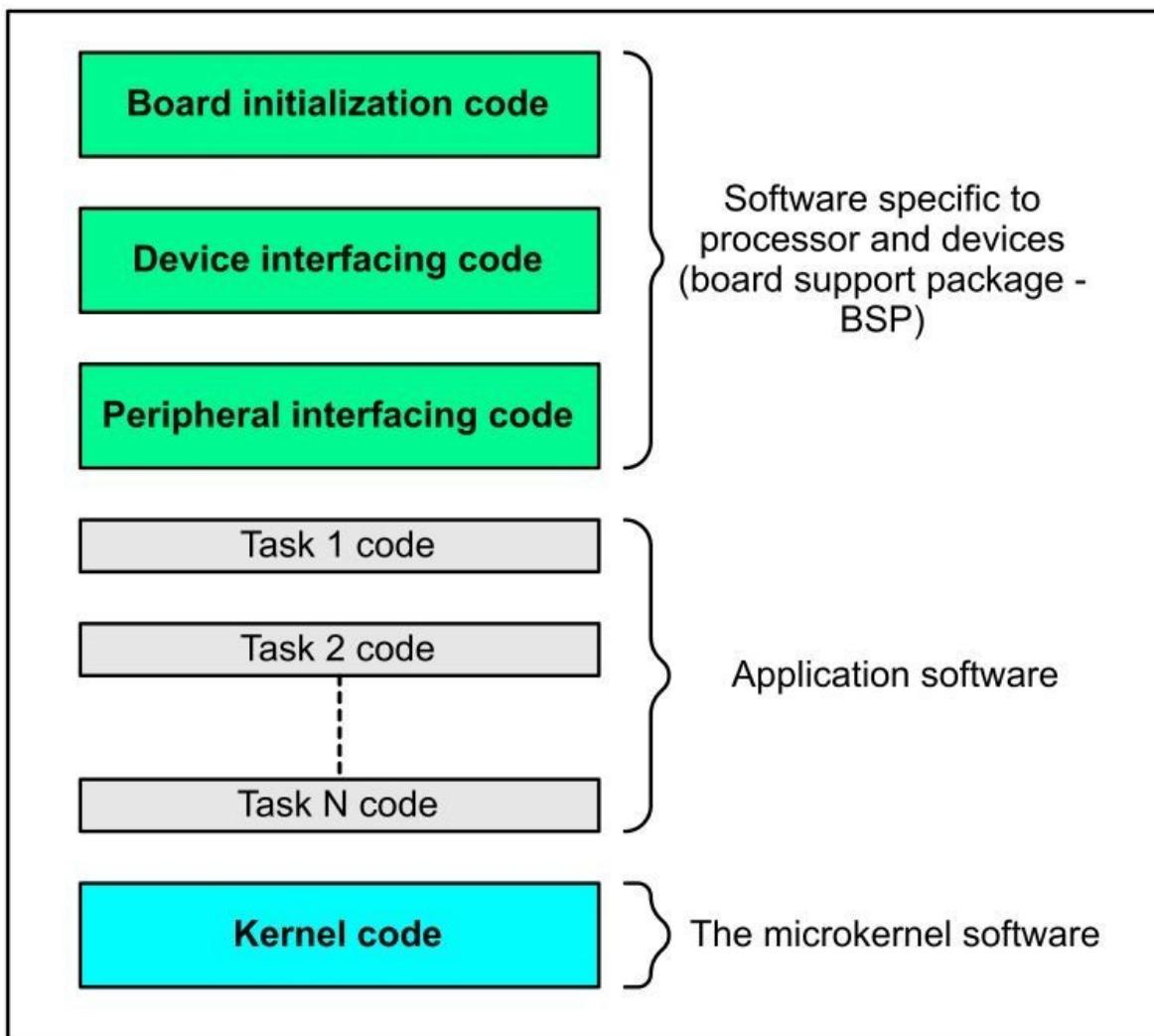


Figure 10.6 Code organization - microkernel-based system This, the BSP, is provided as part of the microkernel package to support both custom and standard hardware designs. Its purpose is to minimize the efforts involved in developing interfacing software for new designs. Board initialization, device interfacing and peripheral interfacing code are all contained in the BSP (which normally includes interrupt handling operations). Though features vary from RTOS to

RTOS the following facilities are found in many packages:

- Board-specific functions, including general initialization, RTOS initialization and interrupt configuration.
- Device-specific driver software, supplied in template form. This is board-independent and therefore needs configuration by the programmer.
- Detailed low-level code used by the device drivers, but applicable to specific devices (e.g. Intel 82527 communications controller).
- Support for the development of special-purpose BSP functions.

The kernel itself has many more features (compared with the Nanokernel, that is). Whether one is designed in-house or bought-in, it should provide a set of operations (or 'primitives') relating to:

(a) System set-up and special functions.

- Initialise the OS (if not part of the BSP).
- Set up all special (non-kernel) interrupt functions.
- Start execution of the application programs.

(b) Process (task) scheduling and control.

- Declare a task.
- Start a task.
- Stop a task.
- Destroy a task.
- Set task priorities.
- Lock-out task (make it non pre-emptive).
- Unlock a task.
- Delay a task.
- Resume a task.
- Control the real-time clock (tick, relative time and absolute time functions).
- Control use of interrupts.

(c) Mutual exclusion.

- Gain control using semaphores (entry to critical region).
- Release control using semaphores (exit from critical region).
- Gain control using monitors.
- Release control using monitors.
- Wait in a monitor.

(d) Synchronization functions - no data transfer.

- Initialise a signal/flag.
- Send a signal/flag (with and without timeouts).
- Wait for a signal/flag (with and without timeouts).
- Check for a signal/flag.

(e) Data transfer without synchronization.

- Initialise a channel/pool.
- Send to a channel/write to a pool (with and without timeouts).
- Receive from a channel/read from a pool (with and without timeouts).
- Check channel state (full/empty).

(f) Synchronization with data transfer.

- Set up a mailbox.
- Post to a mailbox (with and without timeouts).
- Pend on a mailbox (with and without timeouts).
- Check on a mailbox.

(g) Dynamic memory allocation.

- Allocate a block of memory.
- Deallocate a block of memory.

In conceptual terms, the software of a typical microkernel-based system may be

modelled as in figure 10.7.

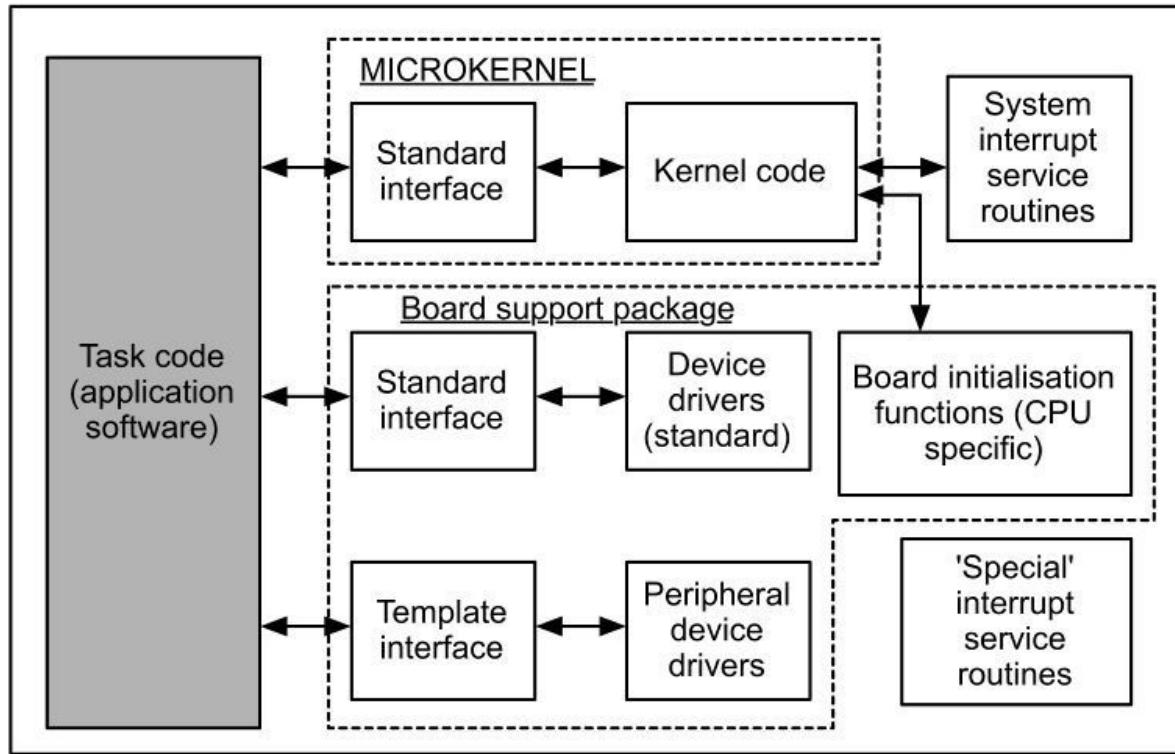


Figure 10.7 Software conceptual model - small microkernel-based system

10.5 A general-purpose embedded RTOS.

Figure 10.8 illustrates the hardware structure and software functions typically found in larger general-purpose embedded systems. From the figure it should be clear how the software functions relate to the hardware items. Note the following significant new features:

- Industrial networking, including standard systems (e.g. CAN, Fieldbus, etc.) and special-purpose designs.
- General purpose networking, with emphasis on Internet applications (e.g. Ethernet, ATM, etc.). Many embedded systems now have the ability to act as mini-web servers, so providing remote access facilities.
- Graphics-based user interfaces (PC-based GUIs and specialist designs).
- Long term persistent storage (disks and semiconductors).

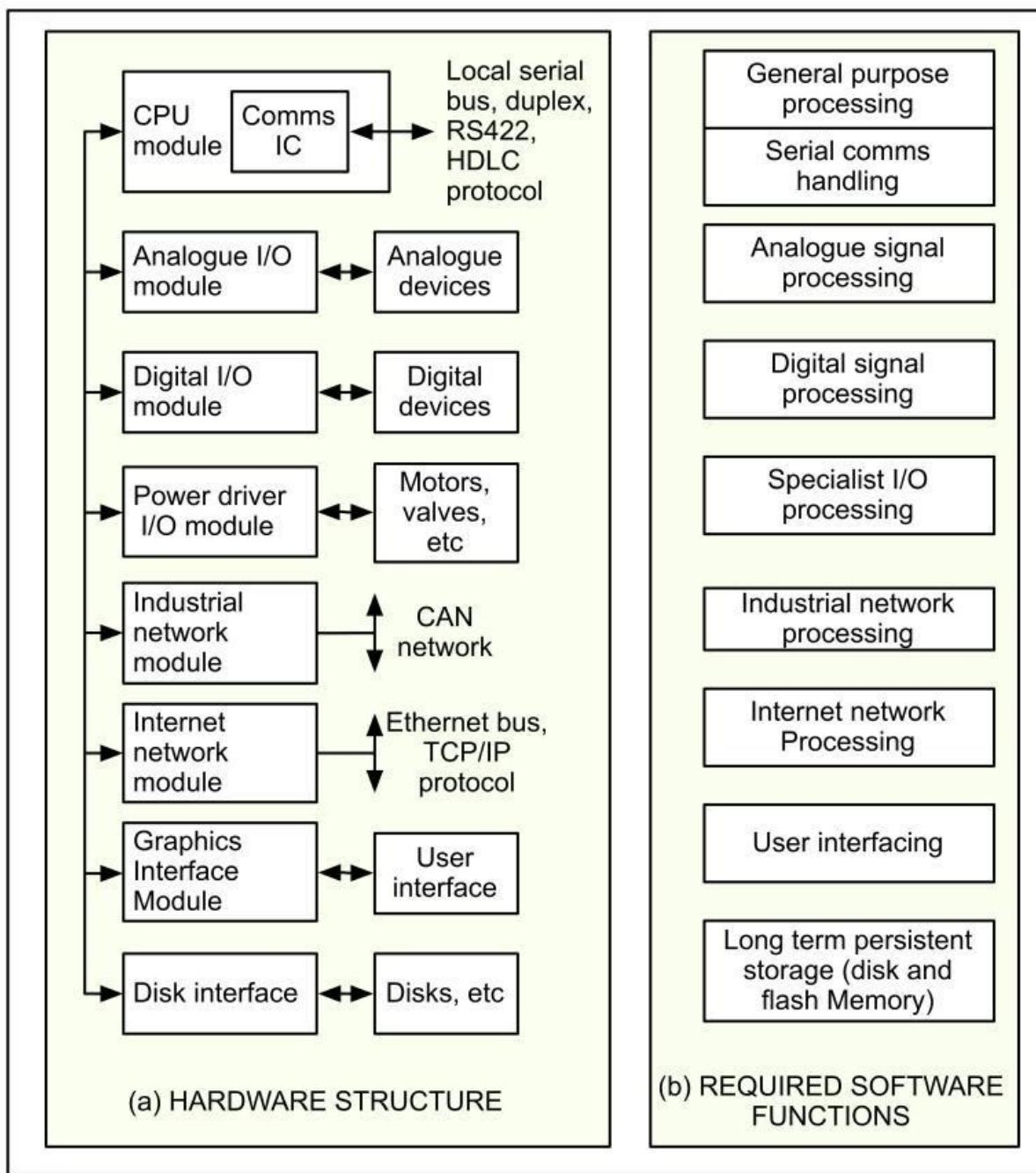


Figure 10.8 Typical large general-purpose embedded system: hardware structure and software functions.

The software conceptual model of Figure 10.9 is characteristic of these larger systems.

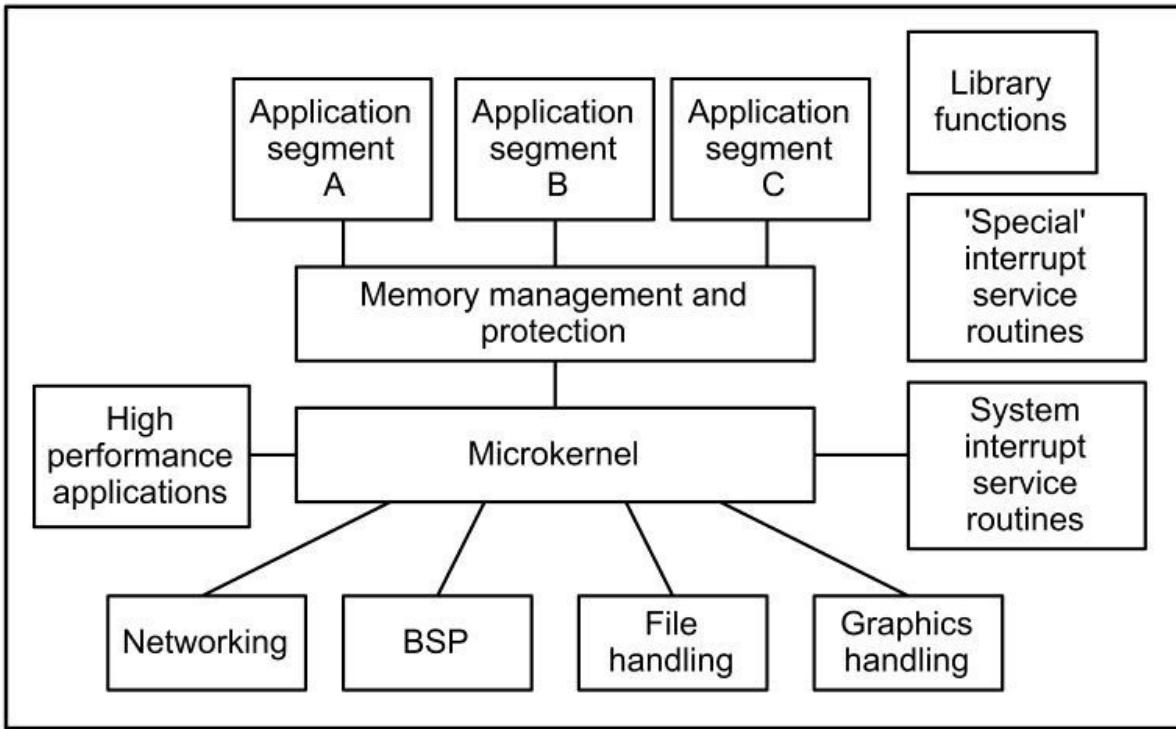


Figure 10.9 Software conceptual model - typical large GP system At its core is the microkernel, which acts as the overall orchestrator of events. Individual software packages are provided for network, device/peripheral, file and graphics handling. Application software is isolated from the microkernel via a memory management and protection layer. In its simplest form this supports two modes of operation; kernel (protected) and other (non-protected). Critical systems use more secure techniques, adding an isolation barrier between the individual tasks. System interrupt routines are those which interact with the kernel itself (e.g. tick timer, peripheral devices, etc.). In contrast, special interrupt-driven routines are allowed to completely override kernel operations. These are normally reserved for use by highly time-critical functions. Applications which are less time critical but still must deliver a high performance often interface directly to the kernel.

A major support facility is that of a set of library functions, usually based on standard C and C++ facilities. However, to be able to use these safely in embedded systems, full re-entrancy is essential (some OS vendors have developed fully re-entrant versions of the standard libraries just to meet this requirement).

Clearly, the network, file handling and graphics handling software sub-systems are not part of the RTOS itself. However, many vendors provide these as an integral part of their RTOS package, designed for ease of use by the application

software.

Review

You should now:

- Know what core functionality an RTOS must provide.
- See why extra functionality is needed to satisfy more complex requirements.
- Understand how you can use interrupts to provide simple quasi-concurrency (multi-tasking) in embedded applications.
- Understand the reasons for and advantages of having clear separation between application, service, initialization and RTOS software.
- Know what a HAL is and what it does.
- Know what a nanokernel-based RTOS is and what services it provides.
- Know what a microkernel-based RTOS is and what services it provides.
- Understand the code organizations of the nanokernel and the microkernel RTOSs.
- Know what a BSP is and what it does.
- Appreciate the range of software functions found in larger general-purpose embedded systems.
- Understand how these relate to the RTOS kernel.

Chapter 11 Performance and benchmarking of RTOSS

The objectives of this chapter are to:

- Introduce the topic of benchmarking as a way to measure run-time performance.
- Explain the difference between computation performance and OS performance.
- Describe how the different benchmarks exercise computer systems.
- Show where and how overheads are incurred during program execution.
- Describe the specifics of OS testing using representative and synthetic testing.
- Show that real multi-tasking applications are best evaluated using synthetic test methods.

11.1 Introduction.

This may sound like a mantra, but it's still worth repeating. Computers in real-time systems must deliver the right answers in the right order and at the right time. And lest you should ever forget, these are determined by system, not computer, requirements. To meet these objectives software must have deterministic behaviour. More precisely, we must guarantee to deliver behaviour that is correct both functionally (functional correctness) and temporally (temporal correctness).

The use of real-time operating systems has significantly simplified the implementation of large, complex software systems. In particular it has made it easier to produce designs that are functionally correct. Unfortunately, the same cannot be said for timing correctness; if anything the reverse is true. And that can raise serious problems for the designer of embedded systems, especially hard/fast ones.

The core mechanism in a multitasking structure is the RTOS. Where time behaviour is critical, we would dearly love to employ one whose performance is fully predictable (from hereon, 'performance' is shorthand for 'time performance'). And this leads to the following questions:

- What information do we need?
- From where can this be obtained?
- How can we use such information to define the performance of the system?

It seems reasonable to assume that information produced by RTOS vendors will provide the answers. Unfortunately these may be less helpful than you might think, for a variety of reasons, including:

- Comparison problems.
- Hidden overheads.
- Timing methods.
- System architectures.

Let us take these in turn.

Point one: If we wish to compare products, then it's essential that we compare like with like. This may not be as straightforward as it should be, on two counts. First, different vendors may use the same terminology but actually mean

different things. One example is 'context switch'. This seems straightforward enough. But it may or may not include a rescheduling decision. Second, the items measured may not in fact be directly comparable (as from two vendors, 'Interrupt Latency' and 'Interrupt Dispatch Time').

Point two: The operations defined are in fact only part of the whole story, thus misleading the reader. For example, to use a particular kernel function, it may first be necessary to perform some 'prologue' operations. In practice this means that extra code must be added to the application program. Yet the overhead of such operations may be excluded from the vendors timing data.

Point three: Timing measurements can be carried out in a number of ways. Some are less precise than others, leading to greater error margins. It is unusual to find vendors explaining, in detail, how their timing information is gathered.

Point four: Most data sheets specify the CPU type (and clock speed) to which the timings apply. However, the architecture of the processor system can also affect timings. Regrettably, details of the test environment are rarely provided with the performance information.

Having said this, however, we have to live with reality. In most cases timing data can be found only in documentation produced by RTOS suppliers. What we need, therefore, is the knowledge to allow us to interpret, assess and use such data.

There is no doubt that many software engineers think this topic is not especially relevant to 'real' systems. Now, this might be true for soft systems that employ powerful processors. For hard fast systems, especially resource-constrained ones, it most certainly isn't the case. This was clearly demonstrated some years ago by Steve Toeppen and Scott Ranville at the Ford Research laboratory, in a project for 'RTOS Evaluation and selection criteria for embedded automotive powertrain applications'. One of the core tests was the assessment of RTOS time overhead (and I do stress overhead) on specific processor platforms. In one set of tests 10 RTOSs were evaluated; under a worst-case load condition the lowest overhead was 23%, the highest 53%. This last figure is staggering. It means that the OS used more than half the available processor power!

11.2 Measuring computer performance - Benchmarking.

11.2.1 Introduction.

Benchmarking is a technique used to establish the performance of computer systems. In benchmark tests a system carries out a set of defined test tasks; the time taken to do these is then used as a measure of its performance. We say that the system has been benchmarked.

Two major areas of computer performance are of interest to the designer: computation performance and OS performance, figure 11.1. Benchmarking of computing functions is well established, having been used for many years. In contrast those benchmarks concerned with OS performance are newer and not widely applied. However, we can learn much from these to help us build our own assessment frameworks. First, though, let us have a brief look at computation performance benchmarks to see their relevance to real-time computing.

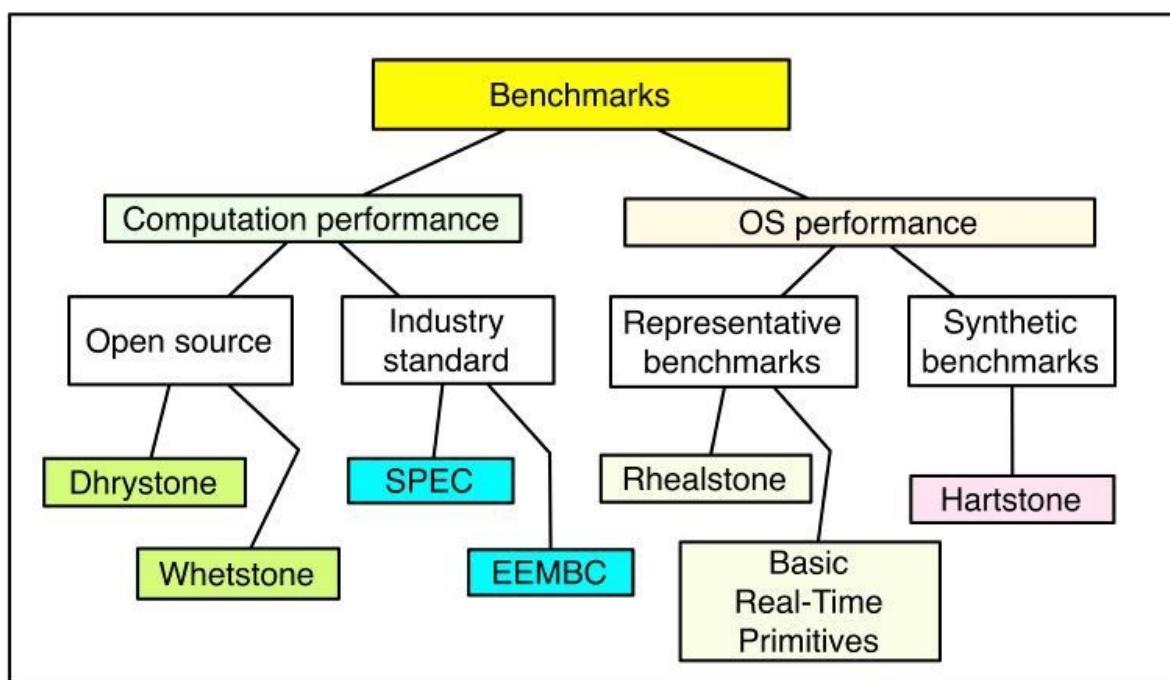


Figure 11.1 Some performance measures of computer systems

11.2.2 Computation performance benchmarks.

The computation benchmarks discussed here fall into two categories: open source and industry standard. They were originally devised for general-purpose systems, to provide measures of the performance of specific computer architectures. The results obtained are heavily dependent on system features such as the CPU, maths coprocessors, cache organization, memory structure etc. Many specific measures have been devised, some well-known ones being:

- Dhrystone.
- Whetstone.
- SPEC (System Performance Evaluation Corporation).
- EEMBC (Embedded Microprocessor Benchmark Consortium).

The basic testing mechanism for all four is similar. Systems execute a standard set of programs, timing data being collected during the testing. Performance measures are based on these recorded run-times.

In some cases the collective data set is used to calculate a single performance index for the computer system.

(a) Dhrystone.

Dhrystone benchmarking is intended to provide performance metrics that can be applied to systems programming. It does this by executing a set of artificial programs that are typical of real computing needs (in other words the programs don't do anything useful). Performance figures depend on two factors: hardware platform and programming language. Exercises involve data and data types, assignments, control statements, procedure and function calls and the like. The rate at which the test suite executes - measured as Dhrystones per second - is used to generate the widely-used MIPS figure (millions of instructions per second). Strictly this is a measure of performance against an industry-standard reference computer, the VAX 11/780, not a raw count of instructions. The VAX is considered to be a '1 MIPS' machine.

To get a MIPS figure for a computer a specified piece of work is carried out, the time taken for this being recorded. This is then compared with the time taken by the VAX do the same job. So, for example, if the test machine is 100 times faster than the VAX, it is rated as 100 MIPS. Just to give a feel for this, ARM quote the following: 'The maximum performance of the ARM7 family is 0.9 Dhrystone VAX MIPS per MHz.'. These figures were obtained using specified test conditions and hardware configuration.

This benchmark does not use floating-point operations.

(b) Whetstone.

Whetstone has its roots in scientific numerical computing. It is geared to measuring floating-point (FP) performance, programs being typical of those found in real FP applications (e.g. matrix inversion, tree searching, etc.).

Typically a test set of 'Whetstone' instructions are executed for a fixed number of iterations, the run time being recorded. From this data the execution rate can be calculated in Millions of Whetstone Instructions Per Second or MWIPS (note that it was originally measured in Kilo-WIPS, KWIPS).

(c) SPEC.

SPEC measures were established to overcome the perceived narrowness of the Dhrystone and Whetstone test suites. Tests are derived from real applications (e.g. circuit design with Spice2g6, simulation with Doduc, etc), and results are used to establish the throughput of the system. The benchmarks are specified in terms of a geometric mean based on the average of all individual test times (a normalized figure).

(d) EEMBC.

EEMBC benchmarks are particularly relevant to our work as they:

- Are specifically intended for assessing embedded systems performance and
- Evaluate how well systems execute specific functions.

They cover a wide range of applications, as noted in the following abstract from their website:

EEMBC develops embedded benchmark software to help system designers select the optimal processors. EEMBC organizes this software into benchmark suites targeting telecom/networking, digital media, Java, automotive/industrial, consumer, and office equipment products.

The tests themselves are defined in a set of software benchmark data books, each one covering a particular technology area. For example, the AutoBench book is intended for use by the auto industry, and lists 15 tests, including 'Angle to Time Conversion', 'Fast Fourier Transform' and 'Pulse Width Modulation'.

Now for a most important point. These figures are normally used to provide a comparative measure of performance, not an absolute one. In spite of this they can be of great help in choosing a processor. Unfortunately they don't give much idea as to how well your overall application will perform. Moreover, nothing is said vis-à-vis the OS.

11.2.3 OS performance.

From earlier work it should be clear that many factors determine RTOS performance other than pure computation times. Examples include scheduling techniques, interrupt handling, context switch times, task dispatching, etc. As a result any useful benchmark test must exercise these features, either directly or indirectly.

OS benchmarks fall into two broad categories: representative and synthetic. In simplistic terms they can be considered as low-level and high-level tests.

Representative benchmarking sets out to provide performance figures for specific RTOS functions such as:

- Task management calls (create task, suspend task, lock scheduling for example).
- Memory management calls (e.g. get a memory block, extend a memory partition, etc.).
- Interprocess communication calls (post to mailbox, send to channel and the like).

This sort of material is usually provided by RTOS vendors, though test methods are product-specific. Several standards have also been proposed, including the RheaLstone and Basic Real-Time Primitive benchmarks. More will be said on these shortly.

Synthetic benchmarks, in contrast, set out to obtain a measure of RTOS responsiveness and throughput. This is done by varying the workload on the system by changing, for example:

- Task numbers and types.
- Task qualities.
- Scheduling algorithms.
- Run-time mix of tasks.

Proposed methods include the Hartstone benchmark, discussed later.

11.3 Time overheads in processor systems.

This section could be loosely retitled 'Where does all the time go to?' To answer that you need some understanding of what happens within the processor system as it executes code. You could, of course, argue there is little value in having such information; as an application designer you have no control over these activities. Now, it is true that you cannot change detailed activities. However, the way in which these are used in a multitasking design may significantly impact on performance. To appreciate this only a broad - essentially conceptual - understanding of certain basic operations is required.

A good starting point is the register structure of a simple microcontroller (very simple indeed, figure 11.2). Here memory and I/O address details are handled within the address register set. Processor status is maintained by the flags register, whilst a set of general-purpose registers supports all remaining functions.

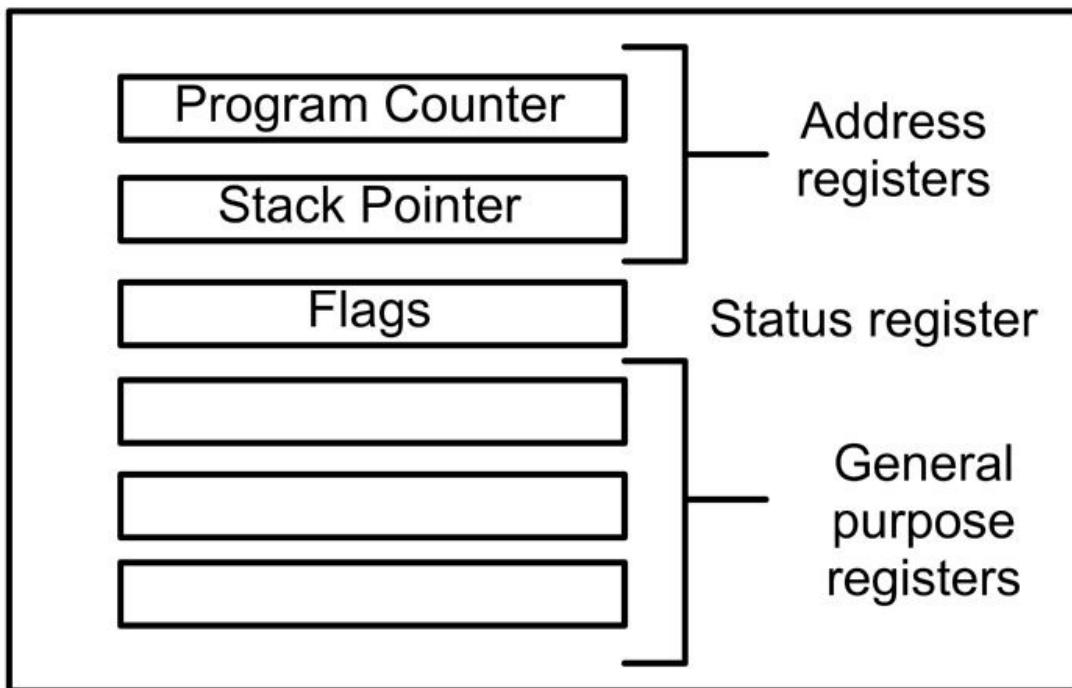


Figure 11.2 Register structure - simple microcontroller

The program counter (PC) - an essential part of any CPU - contains the address of the next instruction to be executed. It is fundamental to and forms an integral

part of the Von Neumann computer architecture. A second essential register is the stack pointer, SP. This holds the address of the top of the stack (a memory store area, organized on a last-in first-out basis, and located in RAM). Note that register naming varies between processor types.

We are now in a position to look at the problem of program overheads. To start with, consider assembly-level programming, the lowest practical source code level. A tiny sequential program that executes all of its statements is unlikely to incur an overhead. However, as soon as subroutines are used, this is no longer the case. Take the situation shown in figure 11.3(a). Here the code of the processor main program has within it a call to a subroutine 'SubX'. For the processor to be able to branch to the subroutine (and to begin executing it correctly), it must be supplied with the address of its first instruction. The 'Call SubX' provides this information: it also initiates transfer from the main program to the subroutine itself. The processor must also be informed when the subroutine finishes; this is provided by some form of return instruction.

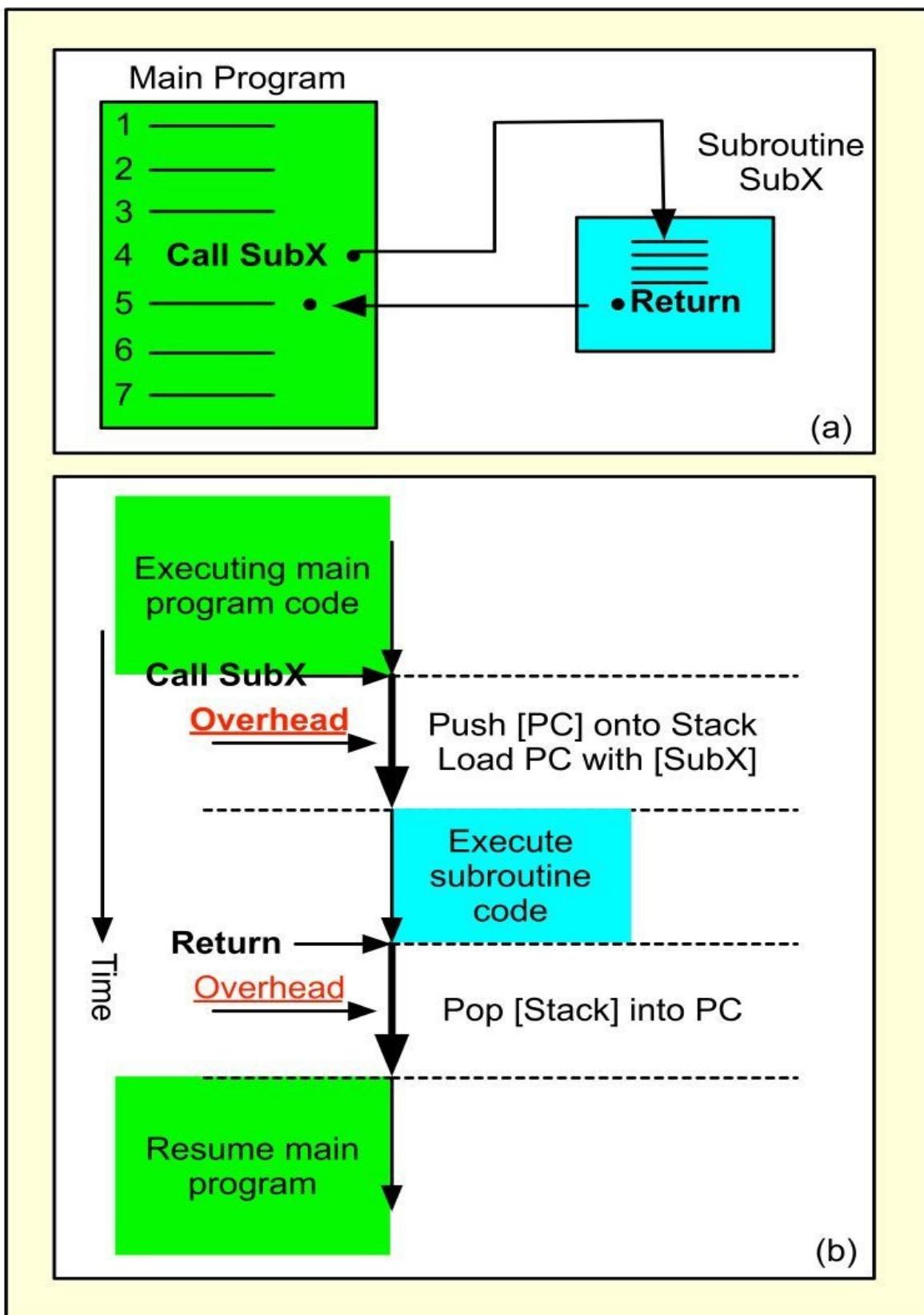


Figure 11.3 Subroutine call - assembly-level operation

As shown in figure 11.3(a), the call instruction is located at line 4 of the source code. Once the processor has loaded the instruction the PC will automatically increment; it now shows the address of the instruction defined in line 5.

Operations then proceed as follows, figure 11.3(b):

- (i) The contents of the PC (i.e. the address of the next executable instruction of the main program) is stored on the top of the stack.
- (ii) The PC is loaded with the address of the first instruction of the subroutine.
- (iii) Execution of the subroutine commences, finishing when the return instruction is met.
- (iv) The contents of the top of stack are loaded into the PC.
- (v) Execution of the main program resumes with the instruction specified by the PC.

It can be seen that by using a subroutine the processor has been forced to carry out extra operations. These, as far as the application is concerned, don't perform any useful work; they do, though, consume processor time.

A more complex situation arises when the subroutine is required to act on data 'passed' to it by the main program (these data items are the subroutine 'parameters'). The most general-purpose technique uses the stack as the communication mechanism, illustrated in figure 11.4.

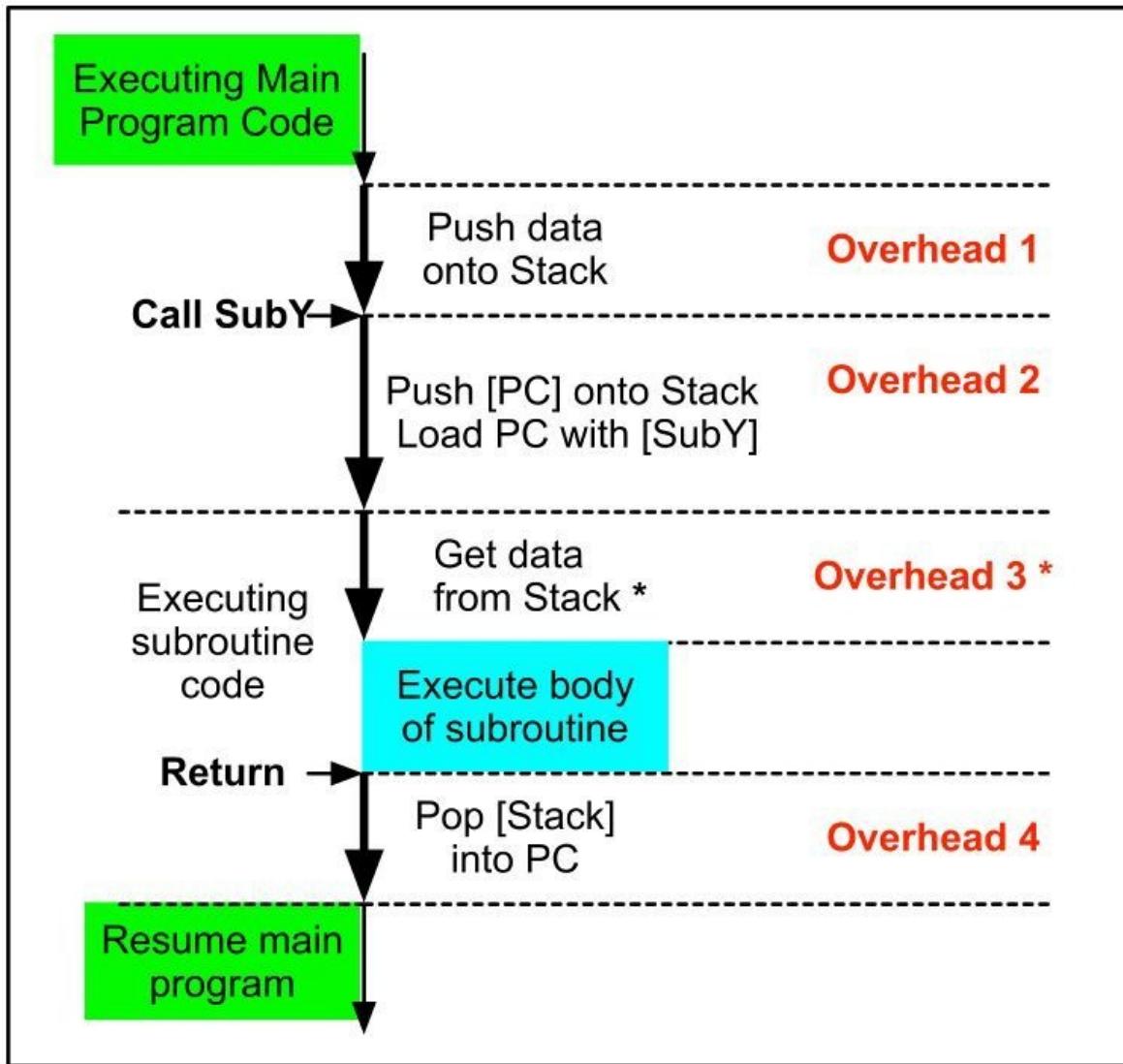


Figure 11.4 Subroutine call with parameters

Here, prior to making the call, parameter information is pushed onto the stack (overhead 1), followed by PC operations (overhead 2). On starting the subroutine the first action is to retrieve the parameter data, overhead 3 (note that this overhead isn't always present - many processors can work directly with stack data). Quite clearly these operations form yet more overheads.

How important is this extra loading? This is a difficult question to answer; much depends on the processor type. However, in time-critical applications, it is always worth developing general guidelines for use by programmers. For instance, in the example above, the following metric was produced for one widely-used 8-bit microcontroller:

Overhead - basic subroutine: 1.0 unit of time.

Overhead - parameter passing operation, one data byte: ...1.5 units.

Overhead - basic subroutine having one data byte parameter: ...2.5 units.

At this point you may be wondering what this has to do with RTOS work. Well, many (if not most) kernel operations are based on subroutines that use parameter passing techniques. Thus the number of parameters, and their data types, have a significant effect on system overheads (hence pass-by-value generally produces much greater overheads than pass-by-reference). Moreover, composite data types (arrays, for example) often consist of a large number of individual data items. Finally, some prologue code has to be produced to set up the parameter information prior to calling the subroutine. Depending on the application the overhead may be significant. If the times quoted by a vendor do not include such factors your performance estimates could be rather wide of the mark.

Although assembly language is used for RTOS applications, this tends nowadays to be restricted to handling detailed (CPU-level) functions. In general the majority of commercial designs provide high-level language interfaces to the OS features. Unfortunately, from a performance point of view, programmers aren't always conscious of the accompanying overheads. For example, consider the use of a simple function call (ComputeSpeed), figure 11.5.

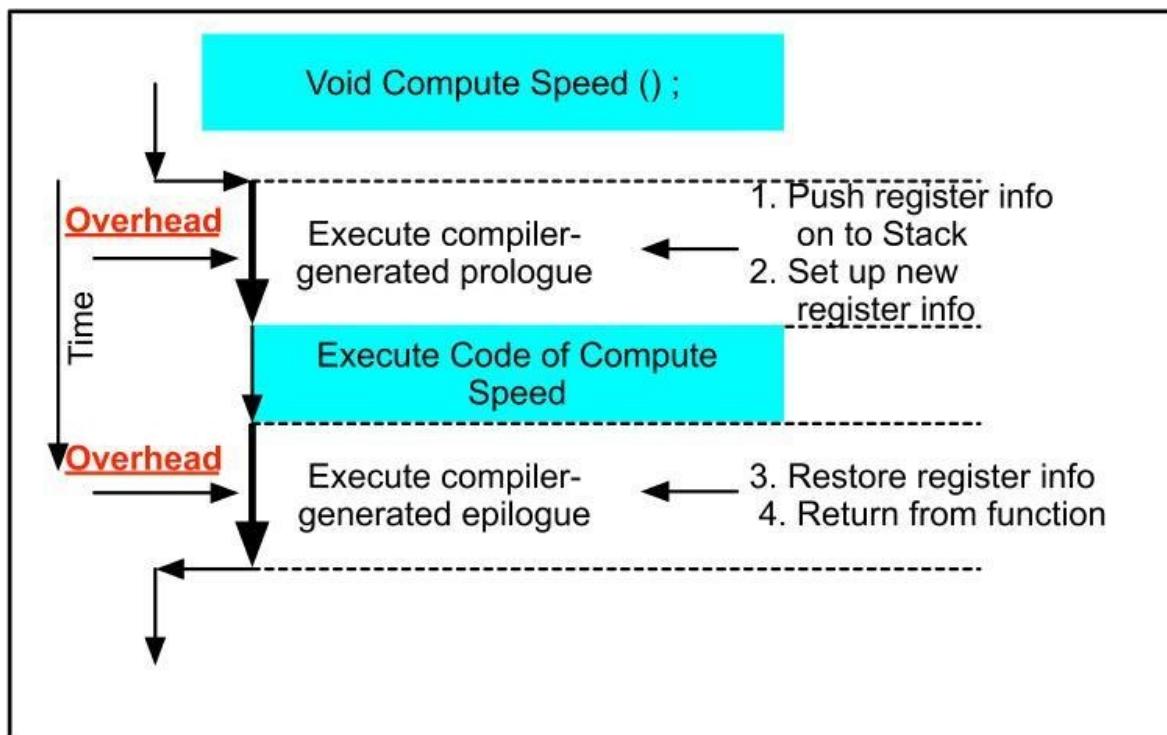


Figure 11.5 Function call - HLL operation

Unlike assembly language operation, the source code gives no indication of its associated overheads. However, the similarity between the overheads shown here and those of figure 11.2 are clear enough. Should the function have parameters, figure 11.6, then extra overheads are incurred similar to those shown in figure 11.4.

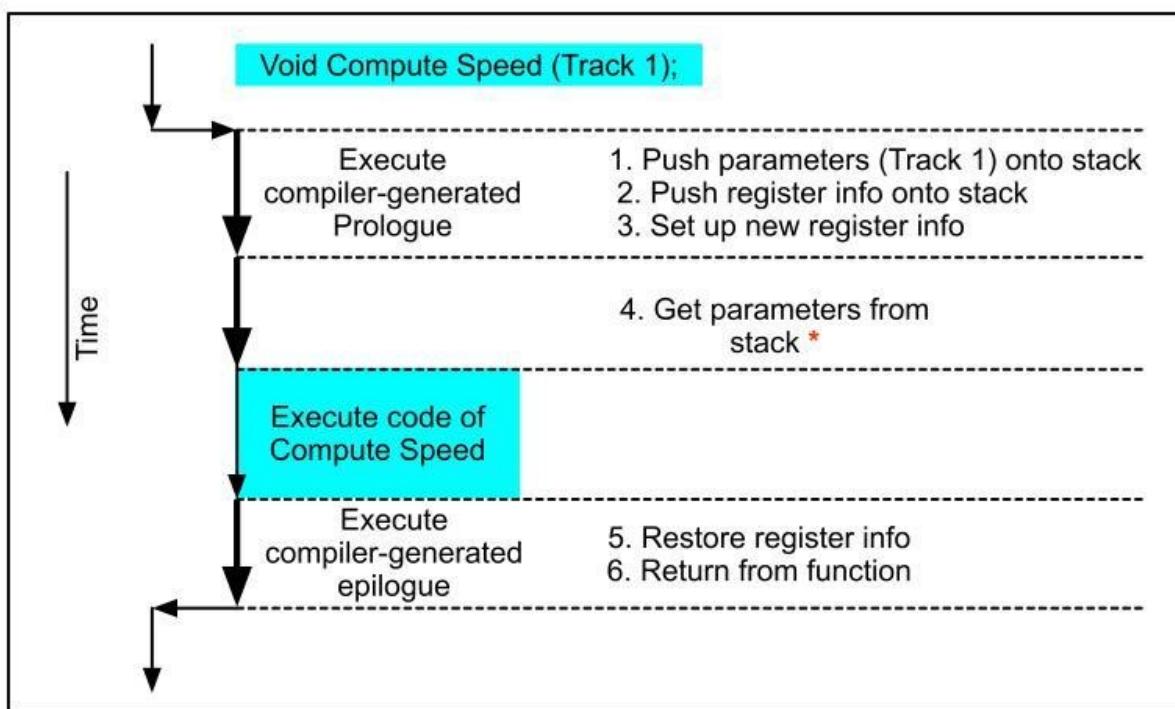


Figure 11.6 Function call with parameters

Now to turn to another important component of real-time operating systems: the interrupt. Interrupts are used for various reasons, two major functions being to:

- Enable the processor to deal with external aperiodic events.
- Provide accurate timing for system operation.

We'll look at both of these, with emphasis on the overhead baggage associated with such operations. The first, for clarity and simplicity, will not involve any operating system facilities. Assume, for example, that the processor is executing a background loop when an external device signals a hardware-generated interrupt, figure 11.7. Assume also that interrupts are enabled. When the processor receives an interrupt signal it continues normal processing until its current instruction is completed. At that point it acknowledges the interrupt (a hardware signal), and begins to save essential register information to the stack. It

now obtains information identifying the interrupt type, then branches ('vectors') to the related interrupt service routine (ISR). Note; the time between receipt of an interrupt and subsequent branching to the ISR is called the interrupt latency. This is the best guide to just how quickly a system can respond to external events.

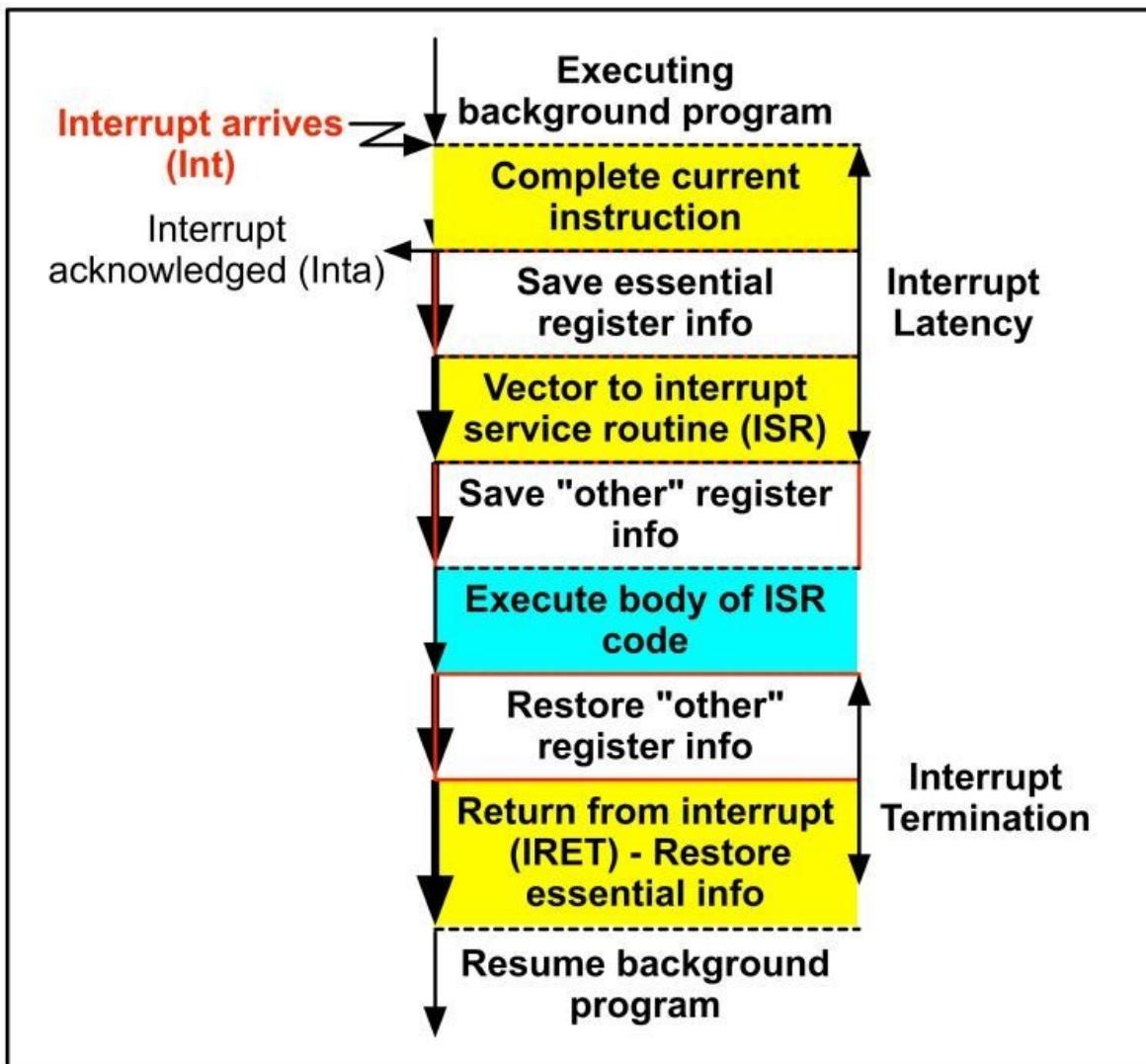


Figure 11.7 Interrupt-driven operation

One of the first statements in the ISR saves the contents of other registers (normally this means saving the contents of all other registers unless there is a very clever compiler at work). Following this the body of the ISR is executed, this leading into an interrupt termination stage. First, all 'other' register information is restored. Second, a 'return from interrupt' instruction causes the

processor to retrieve the essential register information from the stack; it then resumes execution of the background program.

Figure 11.7 identifies all the overheads related to this operation. Comparing these with a function call, figure 11.5, it can be seen that these differ in two ways: interrupt vectoring and the saving of register data. Both are important issues.

The time taken to vector to an ISR can range from the insignificant to something quite substantial. Where, for example, a processor has a set of interrupts, each interrupt signal forces a branch to a specified address. As the ISR will (should) have been placed there by the compiler, vectoring takes a trivial amount of time. The other extreme is met where systems use a single interrupt line that can be activated by various I/O modules. Extra time is needed to identify the interrupting source and to transfer control to the appropriate handling routine.

Saving of register data may appear to carry similar overheads to that of the function call. However, one factor makes a significant difference: the amount of control the programmer has over program execution. In normal sequential programming the programmer decides what function to use and when to use it. As such, the compiler will ensure that only minimal register information need be saved. By contrast, with interrupts, the programmer has no idea when they are going to appear. The interrupted program could be doing anything; in the worst case all registers could contain essential information. As a result all register data must be saved (and, of course, subsequently restored). Which, it might be added, can take a significant amount of time.

Please note that these comments apply to mainstream microprocessors and microcontrollers. There are types specifically designed to minimize multitasking overheads by using extra on-chip hardware. For example, some modern processors have extra sets of registers to hold system context information. Each set is allocated to an interrupt vector, and 'shadows' the structure of the CPU registers. When an interrupt occurs, the general-purpose register set of the CPU is switched out, being replaced by the shadow set. At the end of the ISR, system context is retained in the shadow register set. This arrangement leads to a massive speed-up in context switch times.

One important use of interrupts is to generate the tick time signal. Assume that tasks are executed round-robin fashion, with rescheduling decisions made at tick-time by the ISR (figure 11.8). Two outcomes are possible at each tick. In the first, figure 11.8(a), no rescheduling is needed; the current task is reinstalled. In the second, figure 11.8(b), a reschedule does take place. Here the current task is

suspended, being replaced by a different one. It can be seen that this incurs extra overheads, specifically time to:

- (i) Save the context information (which may involve more than just the register data) of the current task.
- (ii) Reorder the task schedule.
- (iii) Restore the context information of the new task.

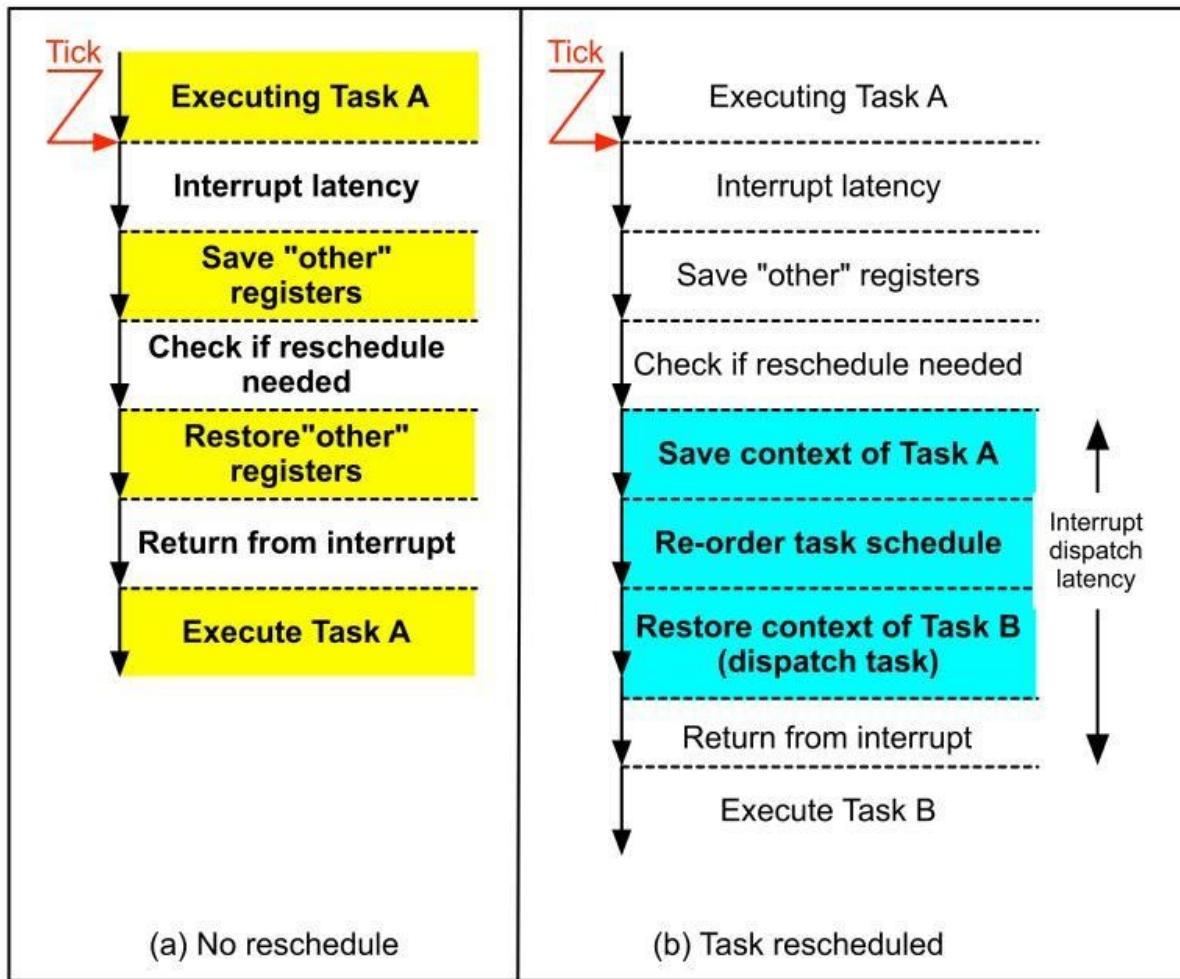


Figure 11.8 Tick-driven scheduling

Clearly, figures obtained by measuring a context switch time that doesn't involve rescheduling are the more flattering ones. Be aware of this.

Another measure sometimes used is called 'interrupt dispatch latency'. To explain this, let us revisit figure 11.8(b). It can be seen that the final operation in the rescheduling process is to return from the interrupt. This action loads the CPU registers with the context data of Task B. In such cases the time that elapses

between 'Check if reschedule needed' and 'execute Task B' is an important measure; it shows how quickly a new task can be set running after completing the body of the ISR. This is our interrupt dispatch latency time (but be aware; there doesn't seem to be a standard definition for the completion point of the ISR code).

Some RTOSs use the interrupt mechanism to provoke a context switch. Others embed the kernel functions within a monitor (say), and rely on function calls to achieve the same ends. The time taken for task switching in such circumstances is sometimes called the 'task switch latency'.

11.4 OS Performance and representative benchmarks.

As stated earlier, representative benchmarks provide performance figures for specific functions. Unfortunately, there is one major (and I do mean major) problem here. Unless all benchmarking is carried out on a uniform well-defined hardware platform cross-comparison of figures may be meaningless. Having said that, it is rare to find such information available. In reality our options are quite limited. First, though, recognize that if accurate figures are required, RTOS testing must be done in the application environment. But that is likely to be done after an RTOS has been acquired. Prior to this the main (and perhaps only) source of OS performance data is that produced by the vendors themselves. Such information can be used only as a general guide. Even so, provided it is very carefully scrutinised and evaluated, it can be quite useful.

The basic set of representative benchmarks is that shown in figure 11.9.

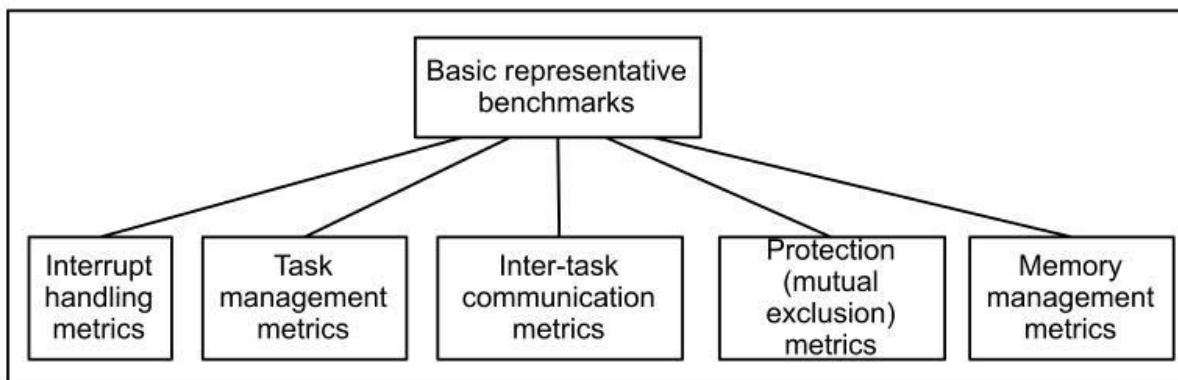


Figure 11.9 Representative benchmark metrics - basic set

Specific data relating to these can be presented in many ways, usually being RTOS dependent. However, some standard tests have been proposed for this benchmarking purpose, including the:

- Basic real-time primitives benchmark and
- Rheatstone real-time benchmark.

The factors taken into account by these benchmarks are given in figure 11.10. Although these tests haven't been widely used, they provide a good basis for the development of company-specific benchmarking techniques. A brief description follows.

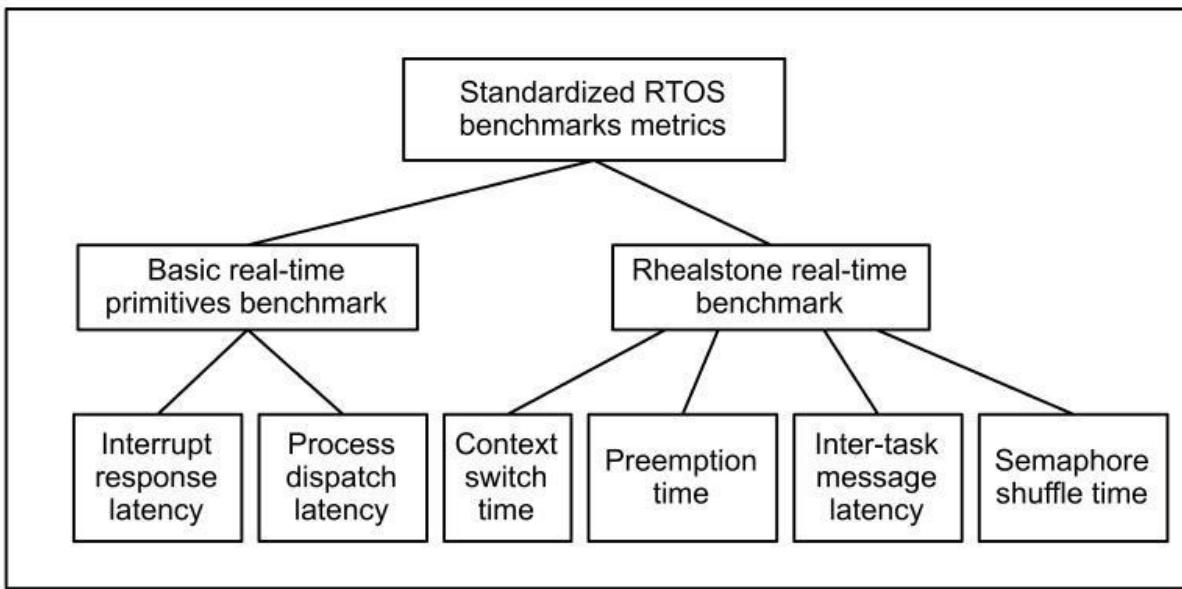


Figure 11.10 Standardized representative benchmark metrics

(a) Interrupt response latency: is the elapsed time from when the kernel receives an interrupt until execution of the first instruction of the ISR. Results are given as min, max and average.

(b) Process dispatch latency: is the time needed for the system to respond to an external interrupt (event) and produce a task reschedule. It assumes that:

- A task is suspended, waiting for the event to arrive.
- The processor is executing a task that has lower priority than the suspended one.
- In response to the interrupt, the suspended task is awoken, replacing the lower-priority one.

More precisely, it is the time between an interrupt arriving and the execution of the first instruction of the suspended task. Results are given as min, max and average.

(c) Context switch time: is the average time to switch between two tasks of equal priority.

(d) Pre-emption time: is the average time for a high-priority task to pre-empt a running lower-priority task.

(e) Inter-task message latency: is the average latency within the kernel when a data message is sent from one task to another.

(f) Semaphore shuffle time: is the average time delay between a task asking for a semaphore and it actually receiving it. It is assumed that, at request time, the semaphore is currently held by another task. The shuffle time excludes the run-time of the holding task.

Establishing metrics for your OS is to be heartily recommended. It does, however, require proper test facilities, personnel, time and money. What do you do though if you don't have such luxuries? There is little choice but to rely on data provided by the RTOS vendor. And the next question is how best to use this information.

Frankly, it is most unlikely that a product would be selected on the single criterion of speed; other factors such as cost, pedigree, technical support etc. are also very important. Thus performance information is probably most useful when deciding which OS features to use and when to use them. Following on here are a set of performance figures for a mainstream microprocessor, measured as part of a research program. All times have been normalized, the context switch time being given the value of 1 (one). Why pick on this as the reference measure, you may ask. The reason is that many inexperienced developers seize on this as the measure of kernel performance. This, it will become clear, is somewhat naive.

To begin with, take some task management metrics, figure 11.11.

TASK MANAGEMENT METRICS	RELATIVE TIME
Do context switch	Normalized - 1
Create a task	3
Delete (kill) a task	2.7
Suspend a task	0.9
Resume a task	0.84
Delay a task	1.4
Change priority	1.14
Get task status	0.48

Figure 11.11 Task management overheads

The first thing that stands out here is the relatively long time taken to create and delete tasks. This should tell you that in fast systems the use of dynamic tasks (i.e. ones that come and go) is going to lead to substantial overheads. By contrast, where tasks are static, creation is a once-only activity; moreover it normally takes place during initialization. Observe next the amount of time taken to delay a task. If you think about what actually happens this shouldn't surprise

you. Also, changing a task's priority is no trivial matter. Moreover, it is possible that a number of tasks may have their priorities changed at the same time.

A second set of extremely important metrics (from a performance point of view, that is) concerns inter-task communication, figure 11.12.

INTERTASK COMMUNICATION METRICS	RELATIVE TIME
Mailbox - Post (no rescheduling)	0.42
Mailbox - Post (rescheduling)	1.28
Mailbox - Pend (no rescheduling)	0.38
Mailbox - Pend (rescheduling)	1.72
Channel - Send (no rescheduling)	0.44
Channel - Send (rescheduling)	1.54
Channel - Get (no rescheduling)	0.5
Channel - Get (rescheduling)	1.54

Figure 11.12 Inter-task communication overheads

These figures are based on the time taken to transfer a single data pointer through the relevant component.

Two points stand out here. First, inter-task communication times are not something to be ignored when doing performance calculations. Second, when rescheduling takes place as a result of task-interaction, the overheads are quite significant.

Similar comments can be made regarding the use of semaphores, figure 11.13.

PROTECTION (MUTUAL EXCLUSION) METRICS	RELATIVE TIME
Semaphore - Create	0.42
Semaphore - Delete	0.44
Semaphore - Check	0.38
Semaphore - Wait (no rescheduling)	0.48
Semaphore - Wait (rescheduling)	1.62

Figure 11.13 Mutual exclusion overheads

Finally, activities related to the dynamic handling of processor memory (figure 11.14) are also important sources of overhead.

MEMORY MANAGEMENT METRICS	RELATIVE TIME
Get a block	0.48
Create a partition	0.96
Extend a partition	0.9
Return a block	0.58

Figure 11.14 Memory Handling Overheads

It should now be clear to you why performance degrades rapidly as more tasks are added to a design. As the number of context switches increase, the overhead rises at a linear rate. Worse still, the need for extra communication and protection facilities leads to geometric-like increases in the overheads. You have been warned.

The preceding data, useful as it is, can be used only as a guide to performance. Figure 11.15 typical of ISR latency test results, shows exactly why this is the case. It can be seen that there is wide range of latency times, the worst being approximately twice as long as the best. Figures for the other metrics can also have a wide spread in value. For example, a set of measurements was produced for the MicroC/OS operating system running on an Intel MCS251 processor. These showed that semaphore and mailbox operations had maximum/minimum time ratios in the order of 7:1. What all this means is clear; it is impossible to predict precisely the actual performance of a multitasking (OS) based design.

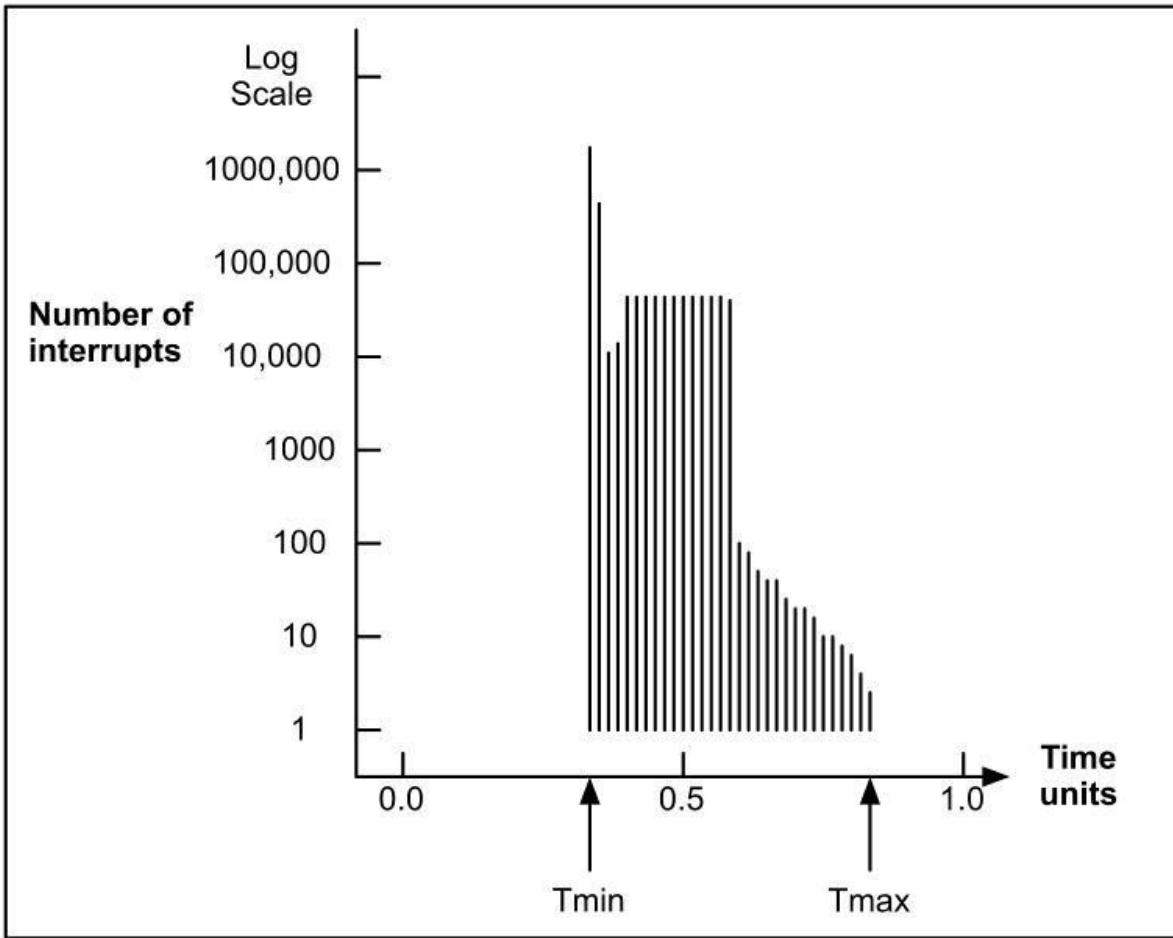


Figure 11.15 ISR Latency test results

So where do we go from here concerning the performance issue? A pragmatic approach is needed, based on data we can obtain. Recommendations are as follows:

- Soft-slow systems: OS performance isn't usually a significant factor. Average times are sufficient if performance prediction is required.
- Hard-slow systems: Similar comments apply.
- Soft-fast systems: Average times are satisfactory. Variations between actual and predicted performance are unlikely to be upsetting.
- Fast-hard systems: For these, worst-case maximum timings must be used (essential for critical applications).

There is a fast-hard sub-category that can tolerate occasional failures to meet deadlines, sometimes referred to as 'firm deadlines'. It's up to you how you handle these as so much depends on specific applications.

11.5 OS Performance and synthetic benchmarks.

11.5.1 Overview.

The information provided by representative benchmarks is best used as a design guide; it helps us to produce efficient, effective and high-performance multitasking systems. Unfortunately the very specific nature of the data involved limits the usefulness of the benchmarks. What we have is a set of individual, isolated measures relating to individual RTOS features. But real systems use combinations of these, their interactions being multiple and varied. Remember, the fundamental question facing us is 'will the system meet its deadlines?'. That is, will the combination of task code and RTOS functions execute sufficiently fast for our needs? There is only one way to truly answer that; build and test the product. The drawback with this method is that all information is gathered as a post-mortem activity. By the time we've found out that the system has a dog-like performance it's a little bit late in the day. What we actually need is a general predictive benchmark that:

- Is not application-specific.
- Gives general guidance on the structuring of multitasking designs.
- Helps identify time-critical operations.
- Indicates, for a specific kernel, ways to achieve optimum performance.

One important step in developing such benchmarks was 'Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications'. Although it hasn't been widely used, the basic ideas are sound and well thought out. They are promoted here as a way of generating company-specific guidelines for developing multitasking designs.

The Hartstone technique is based on four key components (figure 11.16):

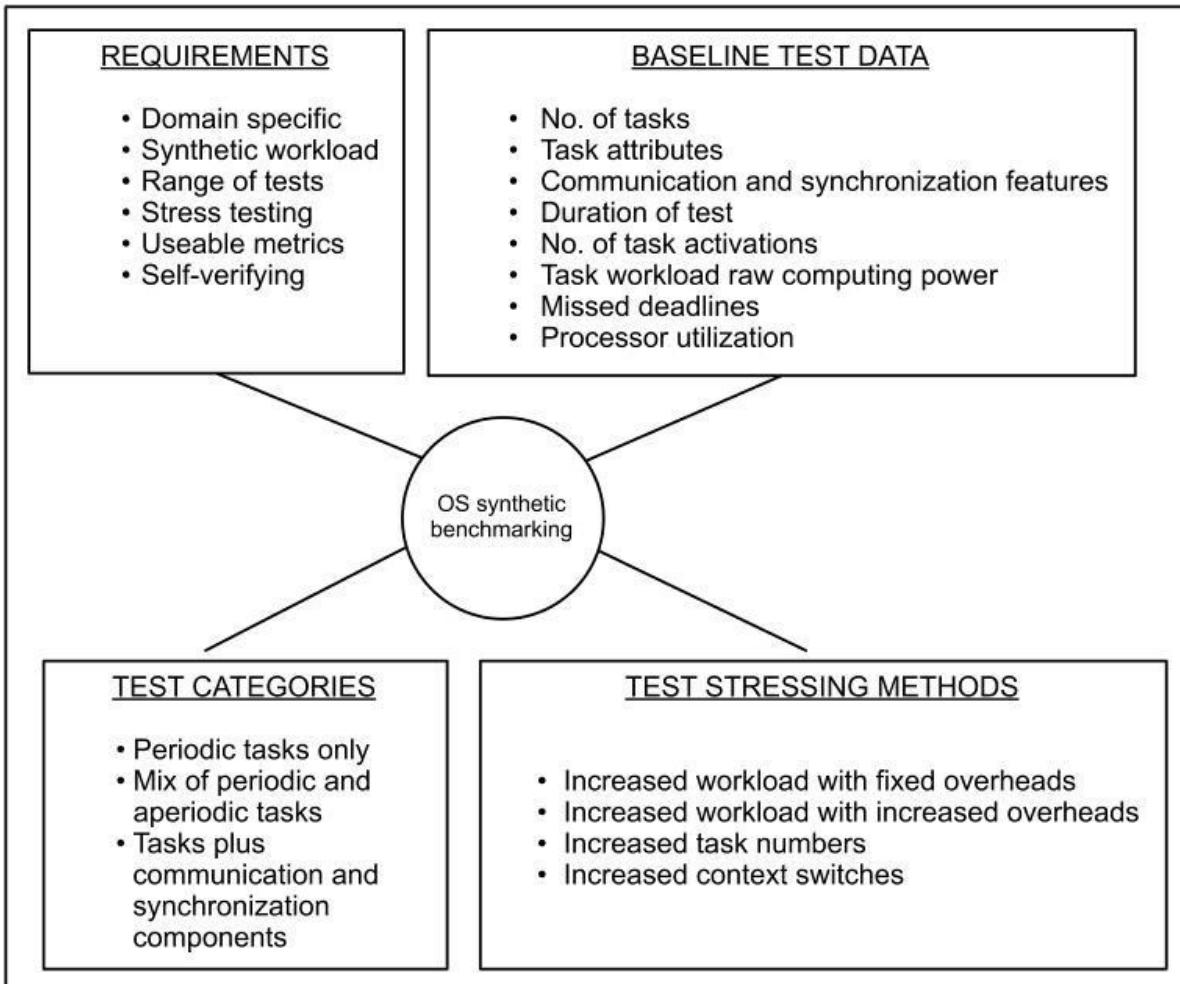


Figure 11.16 Key components of RTOS synthetic testing

- Basic requirements.
- Test categories.
- Baseline (reference) test data.
- Test stressing methods.

Let us take each of these in turn and examine them in more detail.

11.5.2 Basic requirements.

(a) Domain specific: All testing must be relevant to the application domain. For real-time (especially embedded) systems this includes exercising:

- Both periodic and aperiodic task execution.

- Interrupt-driven actions.
 - Task synchronization and mutual exclusion features.
- (b) Synthetic workload: The chosen workload should be typical of that found in the application domain.
- (c) Range of tests: Tests used must range from simple to complex, exercising an appropriate set of functions.
- (d) Stress testing: It must be possible to stress the performance of the system to identify its limits (expressed in terms of missed deadlines).
- (e) Useable metrics: Figures obtained should be able to distinguish between useful work and OS overhead. These should provide a base from which performance estimates can be developed.
- (f) Self-verifying: Each test should be complete in its own right, generating values for all result parameters.

11.5.3 Test categories.

This section defines in general terms the nature of a set of tests applicable to real-time systems. These are intended to show how specific multitasking features impact on system performance. However, in most cases relevant information can be gleaned only by cross-comparing two or more tests (one of the reasons for using a suite of tests). Test categories (based on, but not exactly the same as, Hartstone) include:

- Periodic tasks (harmonic frequencies).
- Periodic tasks (non-harmonic frequencies).
- Aperiodic and periodic tasks.
- Communication and synchronization involving periodic tasks.
- Communications and synchronization involving aperiodic and periodic tasks.

(a) Periodic tasks (harmonic frequencies).

Here tasks are periodic, their repetition rates having a whole number (integral) relationship. This could be simple (e.g. 10 Hz., 20 Hz., 30 Hz. etc.) or logarithmic (1 Hz., 2 Hz., 4Hz., for example). A good reason for using this as the

starting point is that such task arrangements:

- Are the easiest to schedule (typically using a simple cyclic scheduler) and
- Produce good performance figures.

As a result, designers frequently use this as a design criterion in the tasking structure. Results from this test category are just about the best that can be obtained for any particular RTOS.

(b) Periodic tasks (non-harmonic frequencies).

While a harmonic task relationship is desirable, real-world requirements often dictate otherwise. For example, sample rates in closed-loop control systems are related to issues such as bandwidth, responsiveness and stability.

Where tasks end up being non-harmonic, a more complex scheduler - one carrying extra overheads - must be used. In practice such scheduling is much more commonplace than the simple cyclic technique. Consequently the results generated by this test group are useful as a practical best performance baseline.

(c) Aperiodic and periodic tasks.

Where fast responsiveness with low overhead is required, interrupt-driven aperiodic tasking must be employed. Such tasks can be described in statistical terms only, but frequently have to meet specific response times. These tests highlight the interference effects - in terms of missed deadlines - of random inputs on predefined schedules. As such they give a good guide to the percentage of processing time that may be safely allocated to periodic tasking.

(d) Communication and synchronization involving periodic tasks.

Both factors lead to a degradation of performance, especially if priority inversion comes into play. Their impact is more easily seen when evaluated in the context of a set of periodic tasks (by comparing the results with those of test group (a)). The tests also allow the designer to evaluate various alternatives for task communication and synchronization.

(e) Communications and synchronization involving aperiodic and periodic tasks.

These tests bring into play all the features already covered, thus thoroughly

exercising the RTOS functions. With only a small amount of tailoring they can be used for the testing of actual real systems.

11.5.4 Baseline (reference) test data.

For each set of tests a baseline must be established, to be used as a reference point for stress testing. The following items are relevant to the baseline configuration:

- Number of tasks, their attributes and task loading of the processor.
- Communication and synchronization features.
- Test duration, number of task activations, processor utilization and missed deadlines.
- Raw computing power.

The last item, raw computing power, is a measure of system throughput when using simple sequential processing (i.e. without multitasking). In the Hartstone suite this is measured in Kilo Whetstone Instructions Per Second (KWIPS). Each task load can be expressed in KWI per task run. To calculate the total loading per task simply multiply this figure by the task frequency. Summing the results for all tasks gives the total loading on the processor. From this the processor utilization can be calculated. For example, consider the data shown in figure 11.17, which was collected from one specific test:

Baseline test: Periodic tasks (harmonic frequencies only). Test duration: 1 second. Raw Computing Power: 1500 KWIPS.			
Task	Frequency	Task Load	Task Total Load/Sec
Task 1	80 Hz.	1 KWI	80 KWIPS
Task 2	40 Hz.	1	40
Task 3	20 Hz.	20	400
Task 4	10 Hz.	10	100
Total task loading/sec = $(80 + 40 + 400 + 100) = 620 \text{ KWIPS}$			
Processor utilization = $620/1500 = 0.413 = 41.3\%$			

Figure 11.17 Specimen data - RTOS synthetic test

It makes sense that in the baseline tests all tasks should meet their deadlines. Stress testing can then be applied to see the conditions that lead to deadlines being missed.

11.5.5 Test stressing methods.

Test stressing sets out to see what changes in the baseline test conditions take a system to its limits (this is defined to be the point at which tasks begin to miss deadlines). The methods outlined here set out to assess the sensitivity of a design to increases in:

- Workload.
- Overheads (i.e. overheads incurred during the test run).
- Task numbers.
- Context switch rates.

Wherever possible only one parameter is varied, the others being kept constant (in some cases this just isn't possible - results have to be assessed with this in mind). In all cases the most important result is the processor utilization (U) when deadlines are missed. The collective set of U values can be combined to give an overall performance metric.

(a) Increased workload with fixed overheads.

The purpose of this test is to see how the system responds to increases in task workload only.

Constants: Number of tasks; task periods.

Variables: Task load (all tasks) - gradually increased from the baseline value.

(b) Increased workload with increased overheads.

The objective here is to assess how OS overheads impact on system performance. This can be deduced by comparing the results obtained with those of (a).

Constants: Number of tasks; task loads.

Variables: Task frequencies and loads (all tasks) - gradually increased from the baseline value.

(c) Increased task numbers.

This test also sets out to see how OS overheads affect system performance. However, compared with (b), it seeks to answer two further questions. First, are context switch times dependent on task numbers? Second, how sensitive is the system to increases in synchronization and communication functions?

Constants: Task frequencies; baseline task loads.

Variables: Number of tasks; synchronization and communication components - gradually increased from the baseline value.

(d) Increased context switch rates.

This is an attempt to examine the effects of high context switching rates on system performance. As such it increases the overhead while leaving the workload constant.

Constants: Number of tasks; task loads; task periods (except the test task).

Variables: Test task *frequency* - gradually increased from its baseline value.

The simple technique used here does lead to an increase in workload. To

minimize this it is essential to allocate a very low load (say < 1% of task total) to the test task.

Obviously the detail will depend on the nature and scope of the baseline tests.

Review

You should now:

- Appreciate what benchmarking is and what it sets out to achieve.
- Understand, in general, the principles and practices of Dhrystone, Whetstone, SPEC and EEMBC benchmarking.
- Realize the significant difference between the benchmarking of computation performance and that of OS performance.
- Understand what leads to overheads in both sequential and concurrent software.
- Be familiar with the detailed overheads incurred by task management functions.
- Realize that inter-task communication, mutual exclusion and memory handling activities incur (sometimes significant) overheads.
- See clearly why, where computer performance is a key design factor, the number of tasks should be minimized.
- Know what information is produced by OS representation benchmarks.
- Understand what RTOS synthetic testing is, why we use it, what it sets out to achieve and what results it produces.
- Be competent to produce a test strategy based on synthetic testing.

Chapter 12 The testing and debugging of multitasking software

The objectives of this chapter are to show:

- Why and how we test multitasking software.
- Why testing concurrent software is different from testing sequential software.
- Precisely what testing is needed and what the results tell us.
- Why you really should invest in development environments that support the testing of multitasking software.
- How to use the facilities of development environments.

12.1 Setting the scene.

So, you've arrived at a point where design and coding is finished and all code compiled correctly. Your next step? To check how the software behaves in the target system. Now, let's make two assumptions. First, you've been able to test the code of each task as a normal sequential unit and all seems correct. Second, you don't have any special tools to help you test concurrent (multitasking) software.

You could be tempted to apply the 'big-bang' method; download all the code, start program execution and pray everything works correctly. All we can say is 'not a good idea'. If nothing else, the history of failed projects will show you how poor such an approach is (but hasn't stopped people doing it). No, what is needed is an incremental testing technique, starting from the simplest base level.

Implicit in this is that all the hardware is working correctly and all initialization is complete (it is essential that these conditions are met, otherwise it's a bit pointless testing the application software). Your test strategy needs to be carefully planned before you load application code into the target. What complicates the issue is that, in general, tasks are not isolated units; they interact with other tasks. This is where the tasking diagram shows its real value; the interaction components are visible. Using this information, you can devise ways to test each task in isolation by pre-setting interaction values (this is equivalent to 'dummy-stubbing' in normal sequential programs). And if the tasks are dynamically complex then, unless you are willing to spend an enormous amount of time testing, you must develop a suite of test programs. This also implies that your task code must be instrumented in some fashion, thus adding time and difficulty to the project.

Now let's fast-forward to the point where we're ready to carry out full system testing for the first time. There are three possible outcomes of such tests:

- (a) Everything works perfectly or
- (b) Everything seems to work perfectly but there are (or may be) hidden faults or
- (c) There are run-time problems (the 'gloom and doom' moment).

Given our primitive level of instrumentation, outcomes (a) and (b) look exactly the same to us of course. All that we can say is that the software behaviour appears to be correct, both functionally and temporally. But even so, there are many things that we don't know. For example:

- What is the actual run-time behaviour of the set of tasks?
- What are the precise task execution times (including variations from run to run)?
- What are the task utilization figures, both in the short term and also over extended periods?
- How is memory being used, and are there access violations?
- Is the stack usage within predicted limits?
- Are there hidden problems just waiting to cause you great angst at a later date?

Now consider if there are run-time problems:

- What are the problems?
- What are the causes of such problems?
- Which task or tasks are involved?
- Is it a design, implementation or run-time issue?
- What do we need to do to solve the problems?
- How can we be sure that they truly are solved?

If this doesn't convince you that good analysis and test tools will really help you then nothing will!

12.2 Testing and developing multitasking software - a professional approach.

The core message of the previous section is both simple and important; testing multitasking software is much more challenging than testing sequential programs. And never lose sight of the fact that our job is to deliver software that is:

- Functionally correct ('producing the right results').
- Temporally correct ('producing results at the right time').
- On schedule.
- Within budget.

The combined pressures of cost, time and reliability have produced a market requirement for RTOS-specific design and development tools (which RTOS vendors are delighted to fill). Multitasking software development, when using one of the more comprehensive toolsets, usually follows the process outlined in figure 12.1. Implicit here is that system and software design (to task-level) has been completed. Now the goals are to:

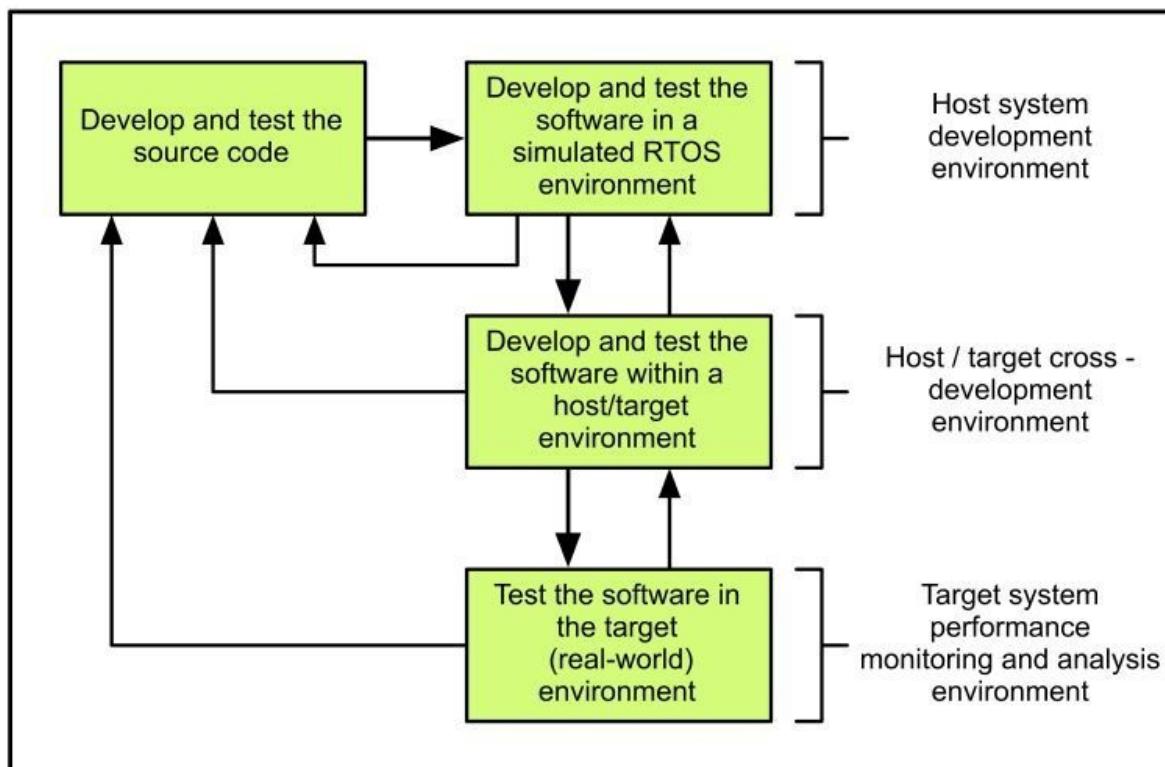


Figure 12.1 Real-Time multitasking software development process

- Produce, compile and test the source-code for each sequential program unit (e.g. task, ISR, passive object, etc.). This is carried out on the host development machine.
- Validate multitasking operation in a host-based simulated RTOS environment (please note; relatively few vendors offer this facility). As before, this work is done on the host.
- Incrementally transfer the software into the target system, developing any new software as and when required. Testing is carried out to ensure that each increment works correctly. Both host and target software facilities are needed to achieve these aims.
- Test the real-world behaviour of the software when it runs in the target system. Here the host facilities are used mainly for performance monitoring and analysis.

Linking all the component parts of the development process is the RTOS integrated development environment (IDE), figure 12.2.

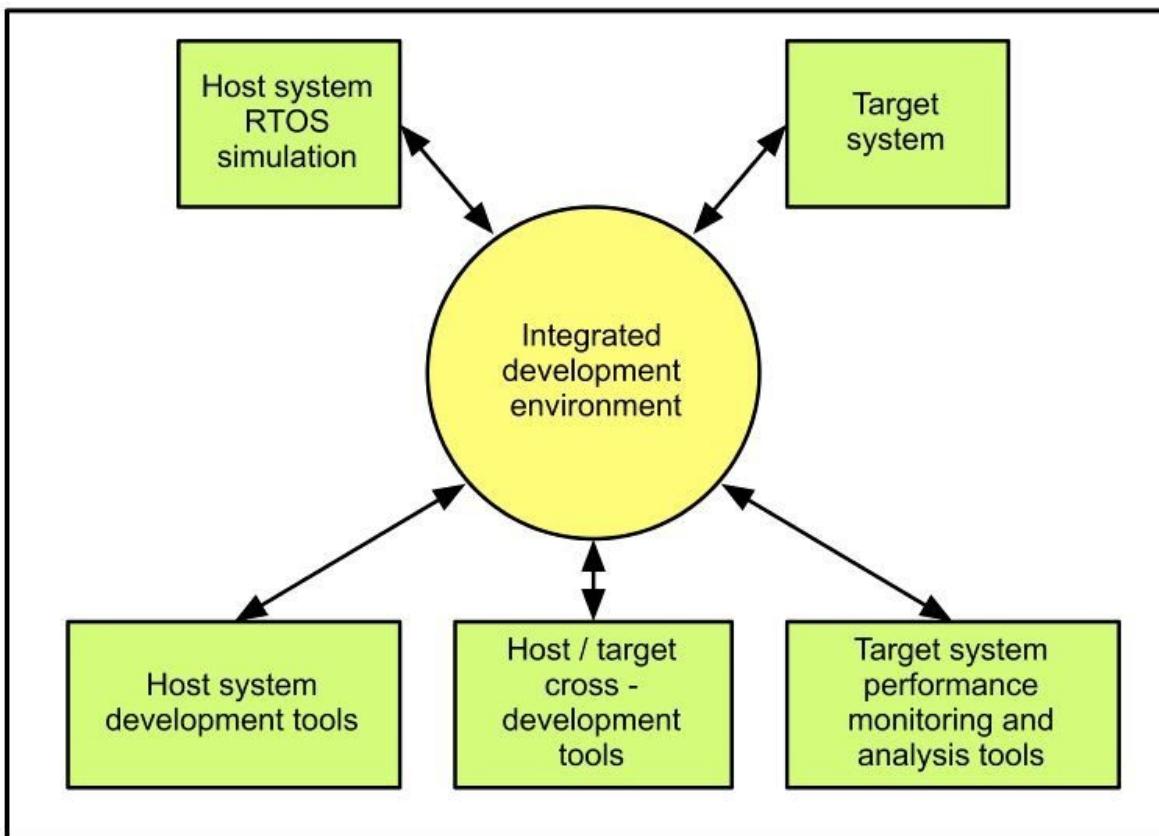


Figure 12.2 A multitasking software integrated development environment

In practice the designer normally interacts with the development facilities using familiar multi-window displays (usually Windows or Linux based). Three sets of development tools are provided within the IDE:

- Those that use host system facilities only (host system development tools).
- Those that use both host and target system facilities (cross-development tools).
- Those for performance monitoring and analysis of the target system (these use both host and target facilities).

Turning first to the host system development tool suite, figure 12.3. These tools fall into two major sub-groups: source code development and run-time analysis.

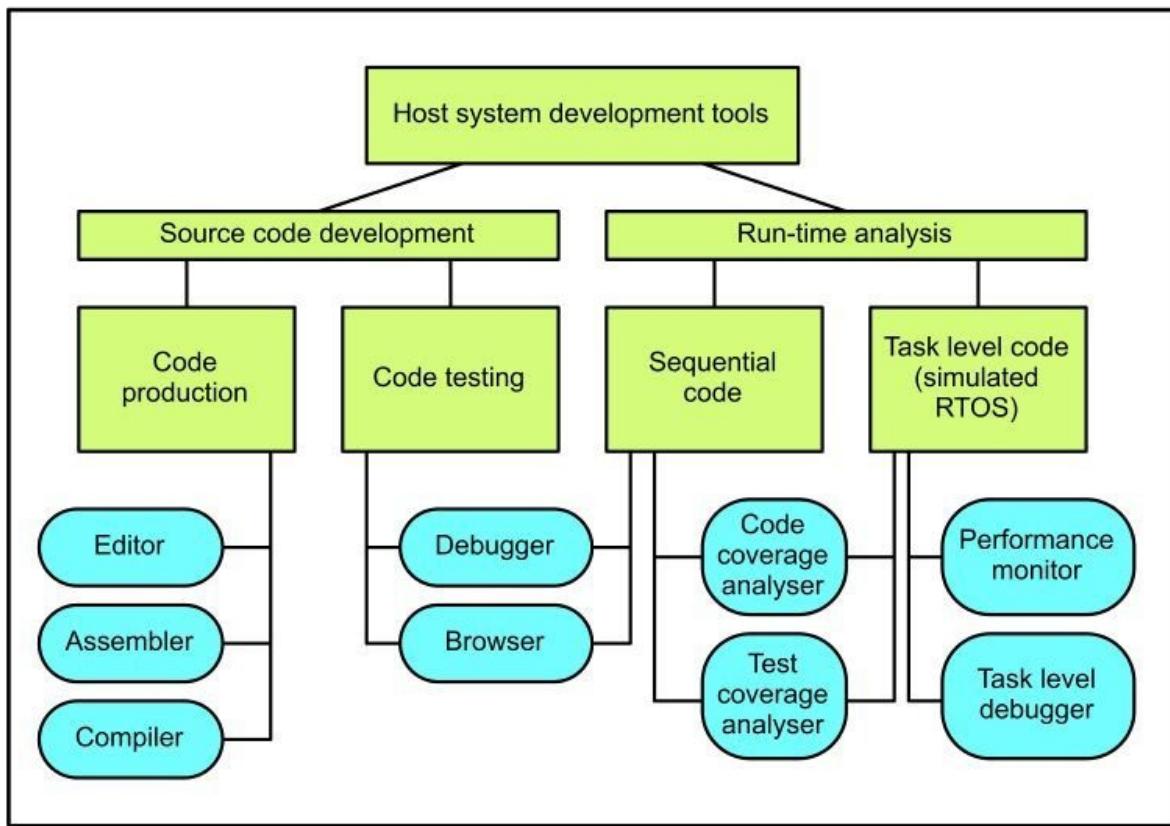


Figure 12.3 Host system development tool suite

Source code development has two component parts: code production and code testing. The first requires the use of editors, assemblers and compilers. Code testing, the second, is usually supported by source code debuggers and browsers. Facilities may also be provided for object code disassembly; much depends on

individual packages. Modern commercial IDEs normally include all these facilities; moreover the features are usually well integrated.

Run-time analysis is normally carried out at two levels. The first (and easiest) is evaluation of the sequential code of each task (there isn't really a clear split between this activity and that of source code testing; it depends on testing strategies). At this stage both code and test coverage analysis can be applied. The second level of analysis takes place at the task level, and includes debugging and performance monitoring. How much can be achieved in practice depends heavily on the quality of the RTOS simulator. A good simulator should include most of the monitoring and interaction features listed in figure 12.4.

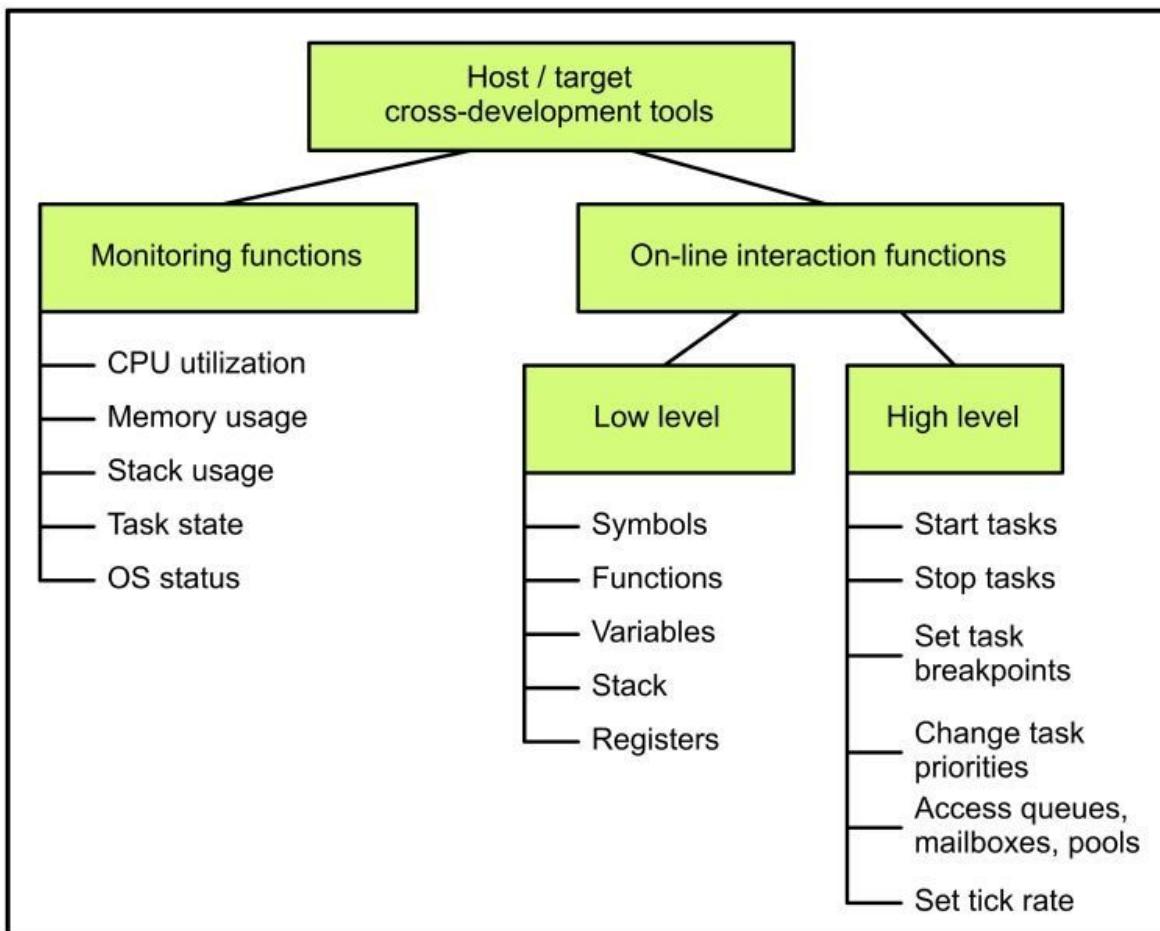


Figure 12.4 Host/target cross-development features

Here the monitoring functions allow the developer to view in depth:

- The behaviour of the target system as a whole.

- Execution of the application code.
- Resource usage (e.g. CPU utilization, memory, etc.).
- OS activities.

For debugging and test purposes a set of on-line interaction functions are supplied. The low-level ones apply to detailed items within the source code and the processor itself. Thus it is clear that the cross-development tools must be configurable; they need to take into account the programming language (and compiler) and CPU type used for the project.

High-level facilities enable to designer to develop and debug the application at the multitasking level; the points noted in Figure 12.4 speak for themselves.

The final stage of development concerns the test of the target system within its real-world environment. Its primary purpose is to ensure that the system really does meet its design aims when deployed into the field. From a practical point of view such testing will probably start with a laboratory mimic of the real-world system. During this the host and target systems are interconnected, data from the target being uploaded to the host (a monitoring function). Information can be displayed live or stored for later processing. It can be seen from figure 12.5 that two distinct issues need to be addressed: RTOS behaviour and application performance. These are two separate but inter-related factors: changes to one are likely to impact on the other. Thus it is essential to evaluate them in an integrated way and not as separate, isolated topics.

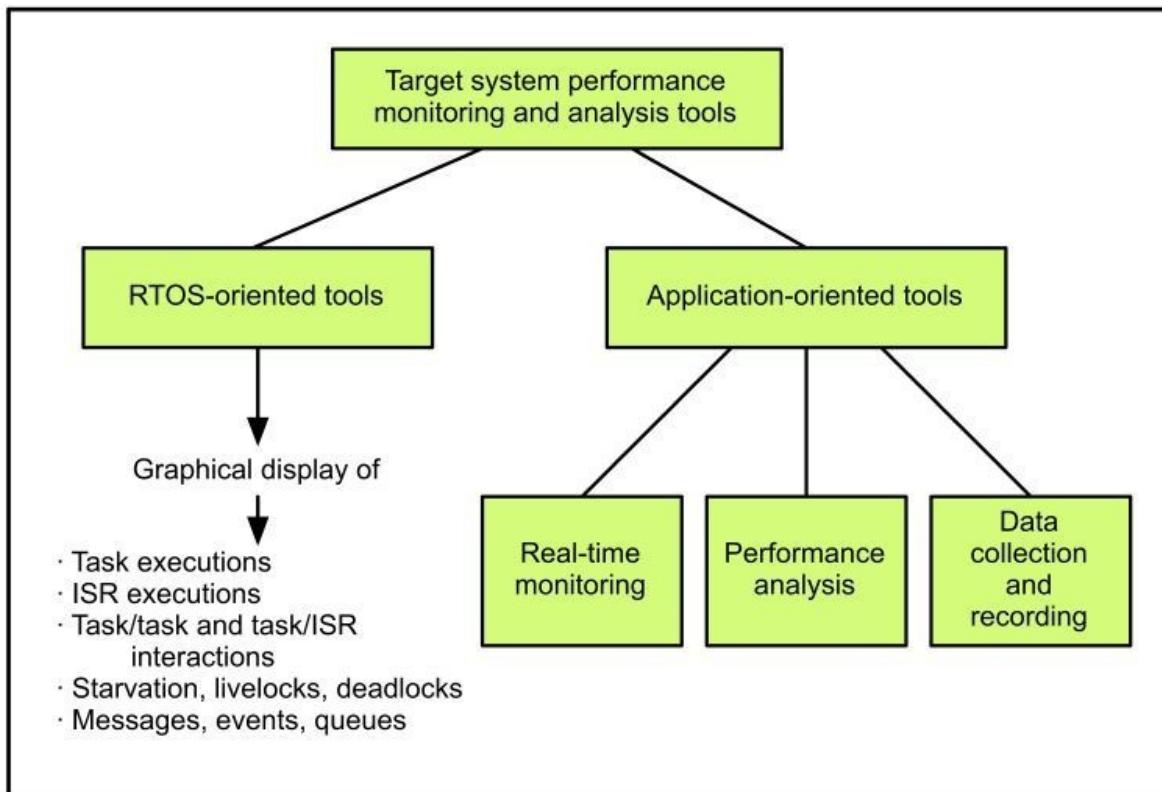


Figure 12.5 Evaluating target system behaviour

RTOS-oriented tools are fairly commonplace, information usually being displayed in graphical form (with text support, naturally). This shows the behaviour of the system from the perspective of the tasking model, focussing on:

- Timing behaviour of tasks and ISRs.
- Effects caused by task/task and task/ISR interactions (e.g. synchronization, pre-emption, etc.).
- Problems related to resource protection features (e.g. priority inversion, deadlocks, livelocks, etc.).
- Overheads and delays due to inter-task communication.

A self-evident comment: if the RTOS performance is poor then there is a good chance that the application performance will also be poor. However, if the RTOS performance is good, it doesn't follow that the application performance will also be good. It all depends on how well you've structured and designed the application software. And key to analyzing and evaluating application performance are the application-oriented tools defined in figure 12.5. In simple terms these allow us to probe the target with a very powerful, flexible 'logic

'analyser'. Thus all digitised real-time information is (potentially) available for display and analysis. There are many ways in which this can be used; in the end all will be determined by the aims of the designer and the capabilities of the tools.

12.3 In-target testing - practical tool features.

12.3.1 Overview.

What has been described so far might be called the 'core toolset'. Now while it would be nice to have a standard generic toolset this just isn't possible in the real world. The reasons are threefold.

First, to use them in a target system they need to be tailored to the:

- RTOS type.
- Microcomputer type.
- Programming language.

And this is really the bare minimum; more features may have to be taken into account (e.g. MMU, MPU, etc.).

The second factor is that the tool must have many details of the application design, including the identifiers and locations of:

- Tasks and their types (including ISRs).
- Mutual exclusion components: semaphores, mutexes and/or monitors.
- Intertask communication components: pools, flags, channels and mailboxes.
- Designated memory areas.

Finally, methods to collect and display target run-time information must be adapted to suit the run-time environment. And bear in mind; you can only trust the results provided the tool doesn't interfere with program execution. In other words, it must be non-invasive.

Most practical tools fall into one of the following categories:

- Those using dedicated control and data collection units.
- Those employing in-target (on-chip) data storage methods.
- Those using the data collection storage of the host system.

Remember, the whole point of this is to see precisely what's happening in the target system as it carries out its work. Simply put, does the software perform correctly, timely and safely? Does it meet its design specifications? Does it have any unwanted, unpredicted or unspecified behaviour? To answer these questions we need to observe the state of the software as it executes. And this really needs

to be presented in a way that is relevant, clear and easy to understand. Without a doubt the most effective method is to display information in graphical form; most engineers are very comfortable with this format. Some representative displays are shown in the following figures 12.6 to 12.10 (produced by the Tracealyzer tool, courtesy of Percepio AB).

In figure 12.6 the target system behaviour is shown, this including task occurrences, execution timing, system events, interrupts and user events.

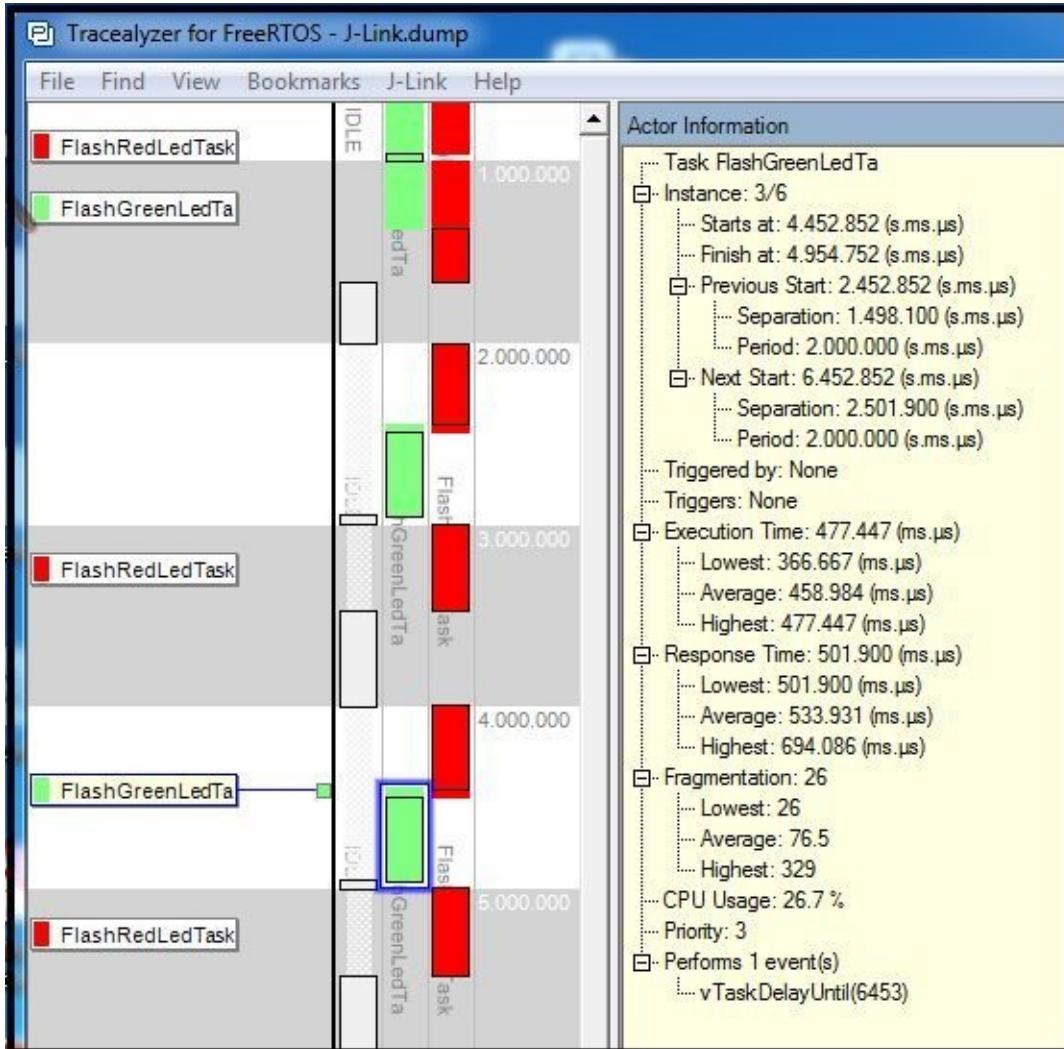


Figure 12.6 Example display - target system behaviour 1

An alternative view of target system behaviour is depicted in figure 12.7, the ‘horizontal trace view’.

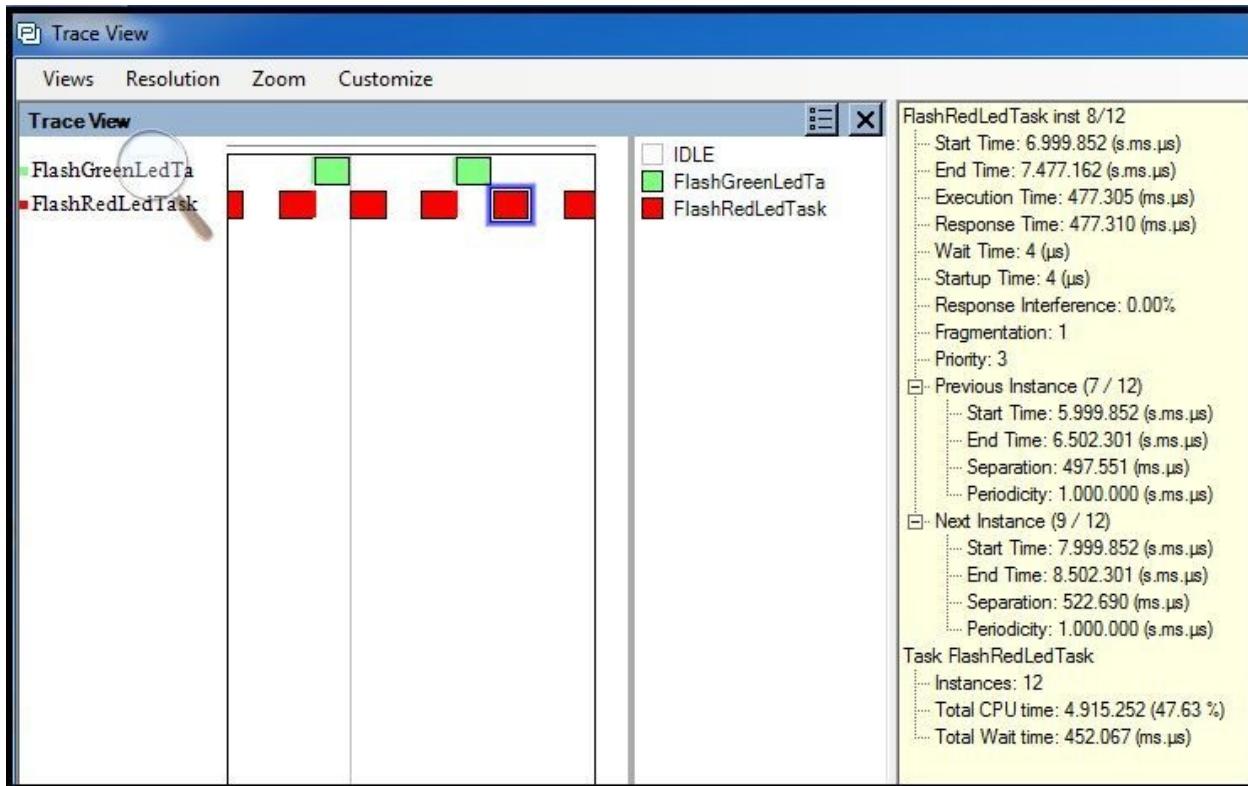


Figure 12.7 Example display - target system behaviour 2

For highly responsive systems a key performance measure is that of processor utilization or loading. Typically this is shown as a plot of CPU loading against time, figure 12.8. Observe that the plot also identifies the concurrent units that contribute to the overall work in each time slot.

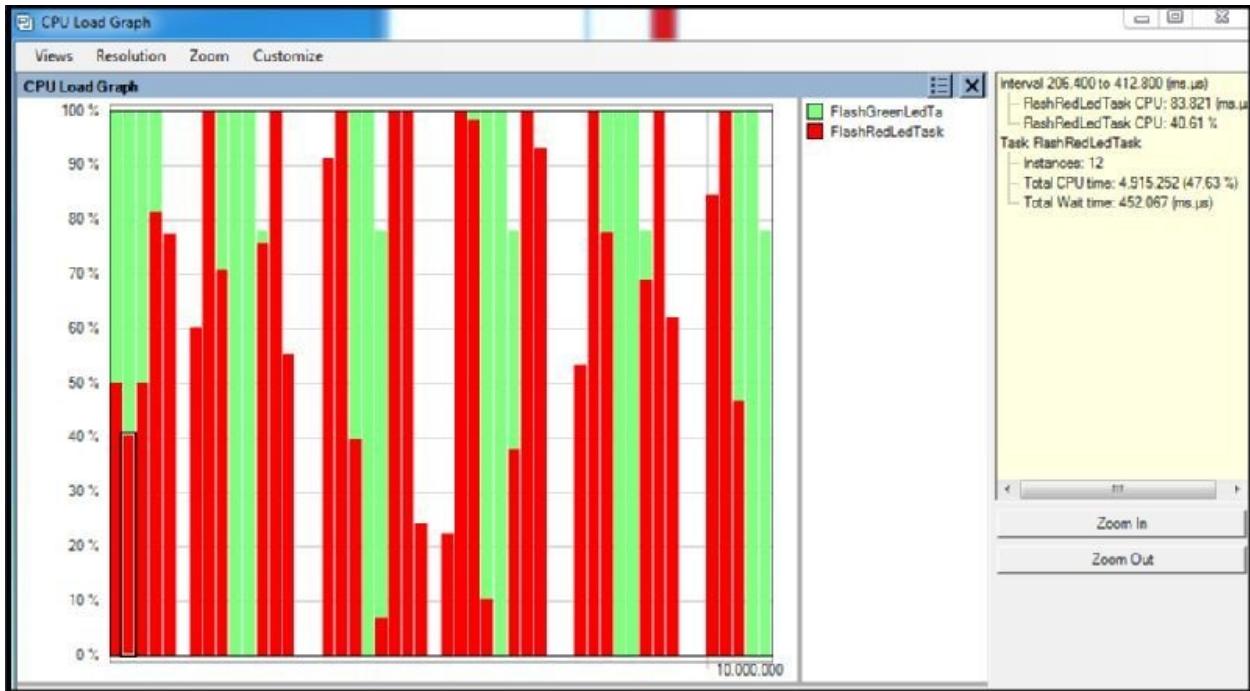


Figure 12.8 Example display - CPU load graph

It can also be useful to look at interactions with communication and protection components such as queues and semaphores. In this Tracealyzer tool these components are called 'kernel objects'; the interactions are known as 'events'. Figure 12.9 is an illustration of messaging information captured in a particular time frame. It is shown both as a list and as a time trace of messages being sent to one specific queue.

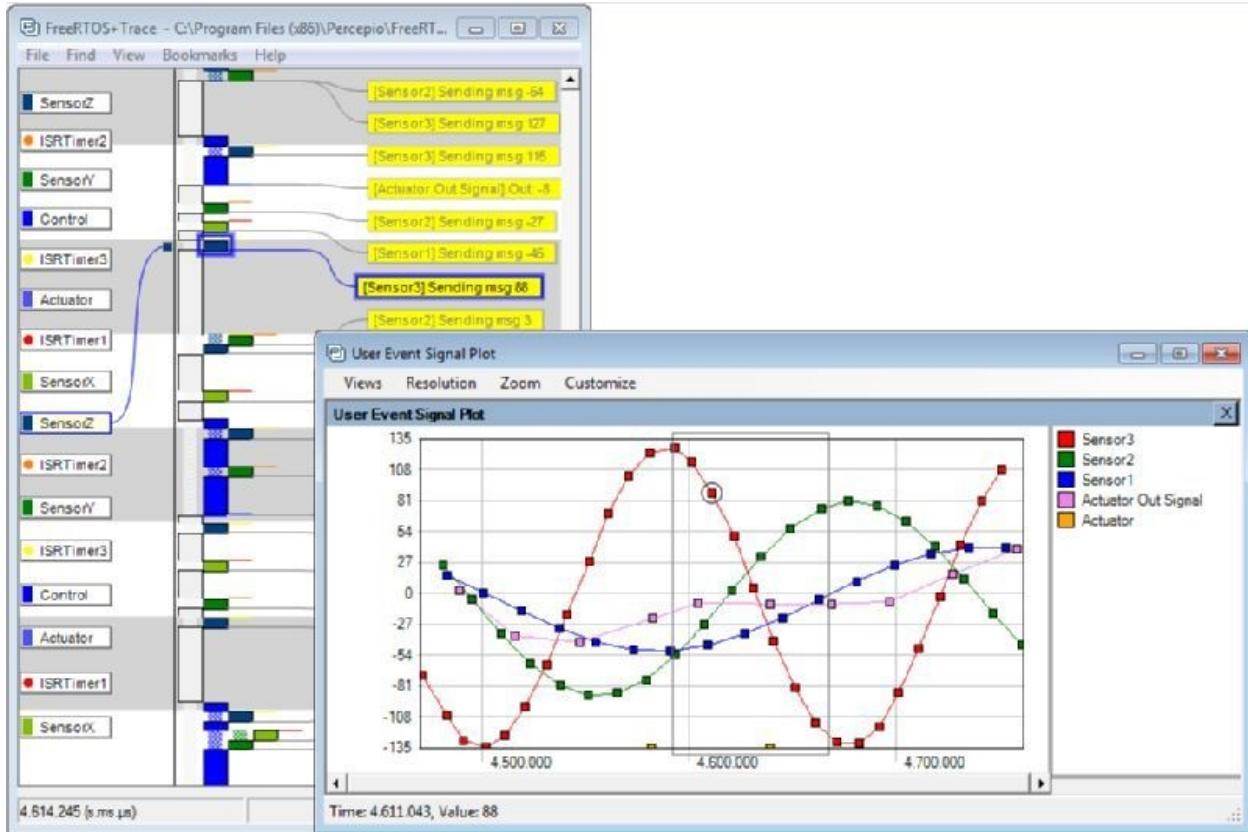


Figure 12.9 Example display - user event and signal plots

And finally, a view that can be of immense help when evaluating inter-task communications: the Communication Flow diagram, figure 12.10.

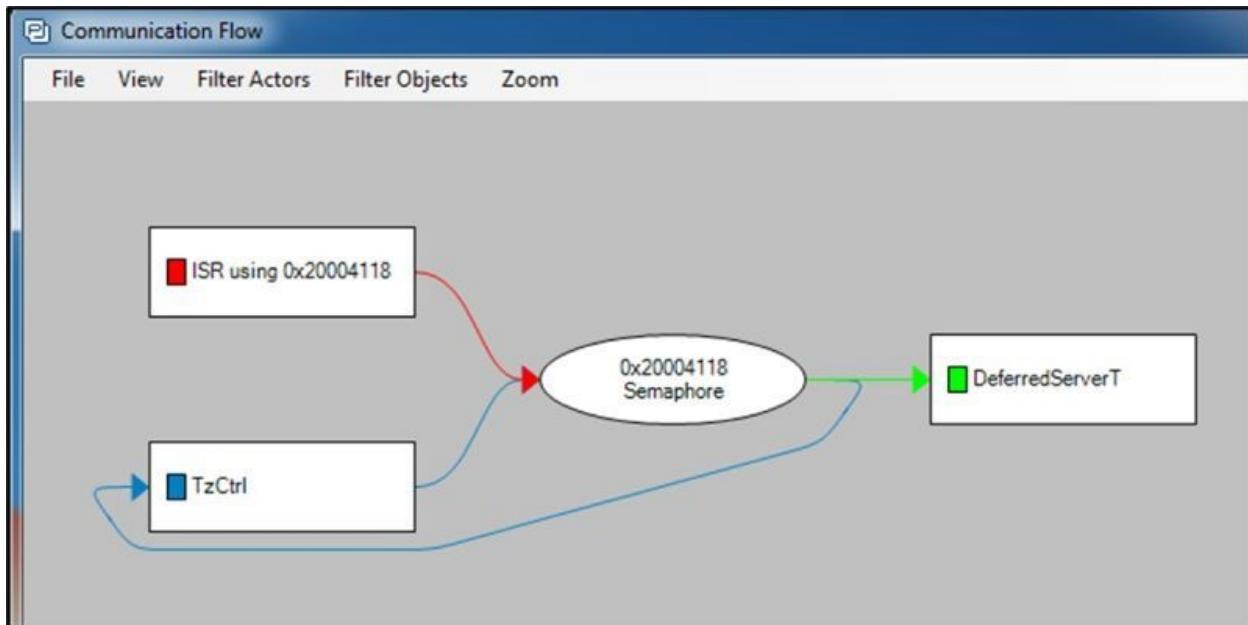


Figure 12.10 Example display - communication flow (part tasking diagram)

There are many commercial tools out there to help you with the testing and development of your multitasking design. In general RTOS-aware types produce information similar to that shown here but their detailed operation and display features vary considerably (as do the costs).

12.3.2 RTOS testing using dedicated control and data collection units.

Figure 12.11 shows the key parts of a system used to carry out in-system testing of multitasking software using a dedicated hardware unit; figure 12.12 is an example of a practical commercial unit.

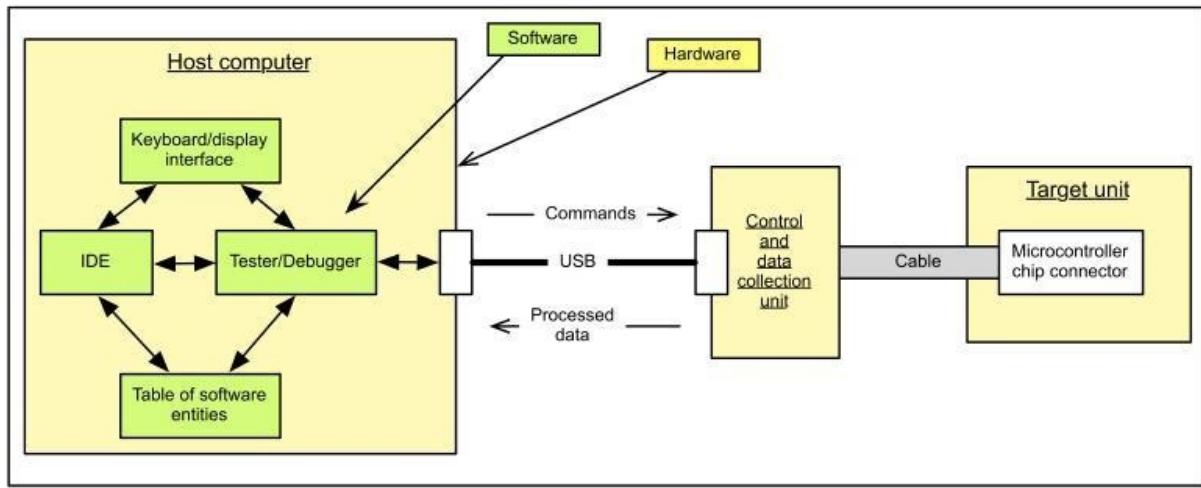


Figure 12.11 RTOS testing using a dedicated data collection unit



Figure 12.12 Example of a dedicated data collection unit - Green Hills Probe
Observe from these that the unit (the 'probe') is connected to the micro using clip-on methods.

In a typical test session the user sets up the required test conditions via the host HMI. This results in commands being downloaded into the probe, all actual run-time testing being handled by the probe itself. During the test run raw data is collected, stored and time-stamped by the probe. Later the processed data is uploaded to the host as and when required. In many cases a specialist software package, the 'Tester/Debugger' of figure 12.11, is used to handle all control, data acquisition, data manipulation and display operations. By using information stored in a table of software entities the analysis software is able to make sense of the raw data acquired from the target.

Such software is provided by the manufacturers of the probe unit as part of the complete tool package.

12.3.3 RTOS testing using on-chip data storage methods.

The approach used here is to collect run-time data during testing, store it in on-chip RAM and then transmit it to the host unit. A relatively low-cost external interface unit normally handles communication between the host and target units, figure 12.13.

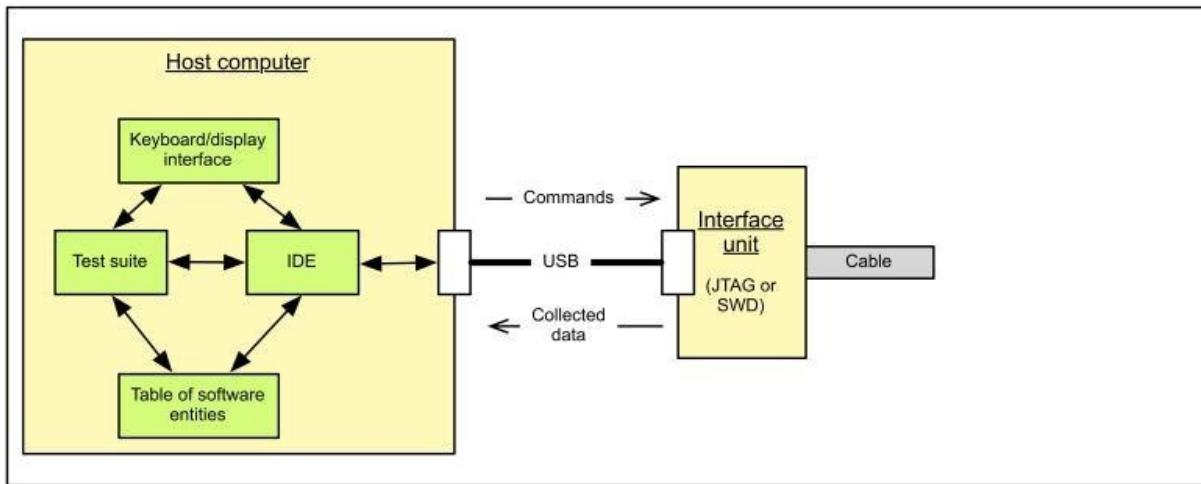


Figure 12.13 RTOS testing using on-chip data storage

Most interface units conform to the Joint Test Action Group (JTAG) IEEE 1149.x standard; many also support ARM's Serial Wire Debug (SWD) techniques. All intercommunication between devices is handled using IDE software, which is usually manufacturer-specific. The test suite is integrated with this software whilst target-based commands are compiled with the application code. In some cases the SWD software is used primarily to handle simple serial communication with the target. In others it provides full on-chip debug (OCD) facilities that extend those of the JTAG debuggers.

The overall test practices and principles are very similar to those of the dedicated collection unit. But note; the target RAM size limits the amount of information that can be collected and analysed at any one time.

Some standard microcomputer boards have integrated JTAG interfaces, providing very low cost test and debug facilities. Moreover, chips are available that can carry out full JTAG interfacing.

12.3.4 RTOS testing using host-system data storage facilities.

Strictly speaking this testing method relies on two factors: the provision of on-chip debug logic and the storage of collected data on the host system. The hardware needed to carry out this type of testing, figure 12.14, is exactly the

same as that of figure 12.13. The difference, of course, is that we don't use on-chip RAM to store the complete run-time data. Instead, this is collected under the control of the OCD logic and shipped out for storage on the host computer.

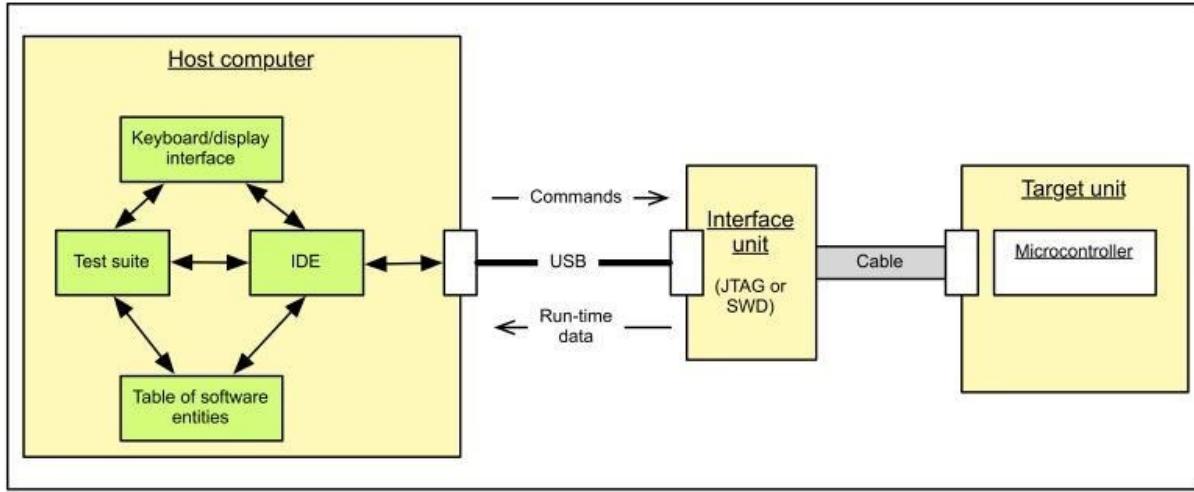


Figure 12.14 RTOS testing using host-system data storage facilities

With this arrangement only a relatively small amount of real-time trace data is stored in the target before being sent to the host. Now this has an important bearing on non-invasive real-time testing. It may be impossible to do this unless both debugging and data transmission are fast enough to keep up with full speed program execution.

The host computer IDE provides all the software to support communication between the host and target systems.

A brief comparison of the three methods is given in figure 12.15.

TECHNIQUE	Dedicated data collection units	On-chip data storage methods	Host system data storage facilities
ADVANTAGES	<ul style="list-style-type: none"> 1. Fully non-invasive. 2. Can collect large amounts of data. 3. Doesn't need on-chip facilities. 	<ul style="list-style-type: none"> 1. Fully non-invasive. 2. Lower cost method. 3. Simpler interfacing using either JTAG or USB serial techniques. 	<ul style="list-style-type: none"> 1. If the IDE includes debug facilities this can be a very low cost solution. 2. External interface unit not needed where on-chip interfacing is provided. 3. Minimal use of on-chip memory in many cases.
DISADVANTAGES	<ul style="list-style-type: none"> 1. Relatively expensive. 2. Number of target processors limited by probe availability. 3. Isn't always possible to <u>directly</u> access internal data via microcontroller external pins. 	<ul style="list-style-type: none"> 1. Uses target system RAM. This has two consequences: <ul style="list-style-type: none"> (a) Can be used only with targets having sufficient spare RAM. (b) Amount of data collected is limited by the RAM size. 2. May need OCD support. 	<ul style="list-style-type: none"> 1. Needs OCD support. 2. May be difficult to attain fully non-invasive operation unless: <ul style="list-style-type: none"> (a) Extensive on-chip storage is used or (b) Data trace signals can be collected and transmitted to the IDE in real-time.

Figure 12.15 Outline comparison of RTOS testing methods

12.4 Target system testing - some practical points.

12.4.1 Introduction.

Please note; what follows is an overview of the subject, not a detailed tutorial.

Start by asking a simple but important question; exactly what are we using the tool for? The answer is that it depends on what phase of work we're in, see figure 12.16. This, although a simplified view, does capture the key aspects of the process.

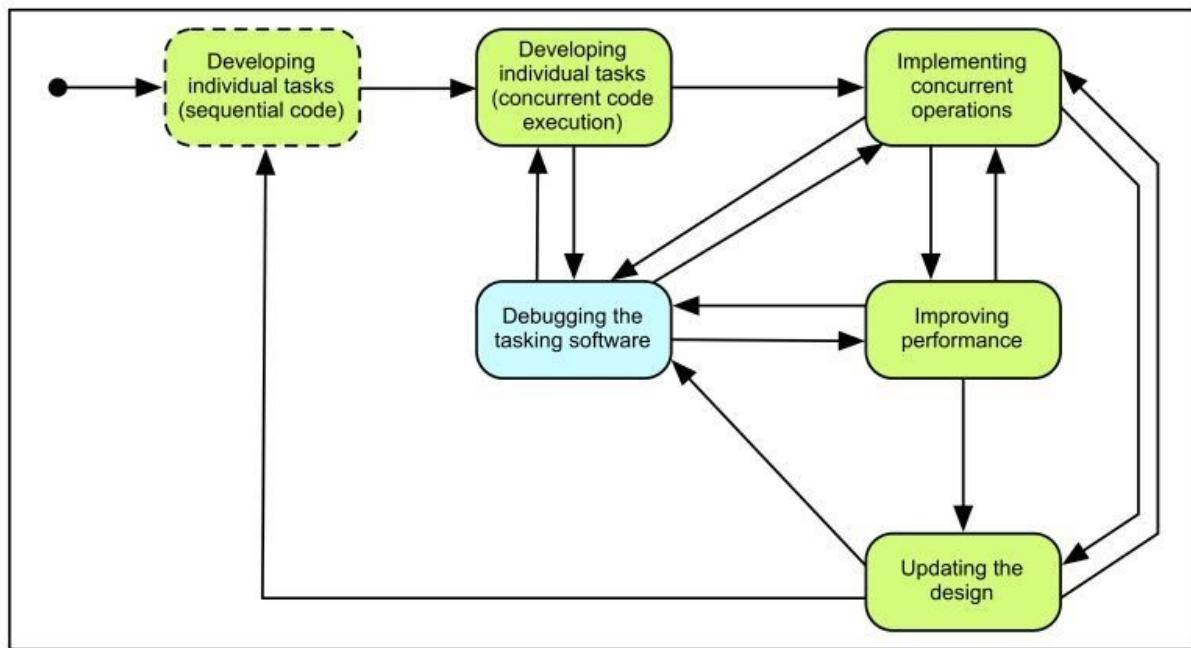


Figure 12.16 Test and debug work phases

So how does this work in practice? Consider the design depicted in the tasking diagram of figure 12.17, one that we'll use as a reference system. The purpose of this system is to control the power and torque delivered by two marine propulsion motors, these being defined by preset parameters. So let us now look at how we'd go about testing this system, starting with evaluating the concurrent operation of individual tasks.

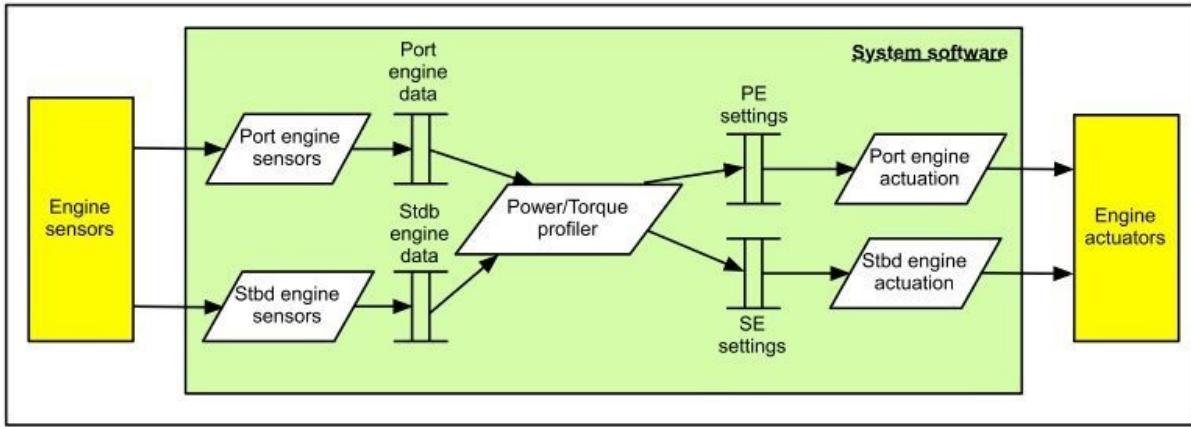


Figure 12.17 Reference system 1 tasking diagram

12.4.2 Testing the concurrency of individual tasks.

Although issues concerning the development of individual tasks have been highlighted earlier, three further points need emphasizing.

First, you should have done your utmost to eliminate bugs from the task code. This implies that rigorous testing, including static and dynamic analysis, has already been carried out. By doing this we can be pretty sure that later problems are due to multitasking aspects (although nothing is 100% certain with software!).

Second, if it's an internal task (such as 'Power/Torque profiler'), then functional testing to a high level of confidence can be carried out purely as a software activity. Here, for example, we could test this task by:

- First, loading test values into the engine data channels.
- Running the task under the control of the RTOS.
- Measuring the values delivered to the settings channels and
- Comparing these with predicted values.

Third, if it's a task that interfaces to the real world (e.g. 'Port engine sensors') testing must involve the use of external hardware. It is essential to do this; you must have total confidence in your software before you get anywhere near the real system. Now, an important question concerning the test hardware: how complex? Unfortunately there just isn't a cookbook answer to this; it all depends on the nature of the work. Basic non-critical systems can, for example, be tested using inputs derived from switches and potentiometers, outputs being delivered

to LEDs, bar-graphs, analogue meters and the like. For more complex systems it may be necessary to build complete test rigs that mimic the real system. Note that in many cases you can test systems using software-generated inputs. However, this should be seen only as a stepping-stone to hardware-based testing.

12.4.3 Implementing and testing concurrent operations.

When implementing and testing the concurrent software we have two aims in mind. First, we want to validate the overall functional behaviour of the system. Second, we wish to gather data concerning processor usage and time-related activities. Factors of interest include:

- Task execution timing and timing variations (in particular worst case execution time).
- Inter-task communication activities.
- Correct operation of mutual exclusion features.
- Effects due to inter-task interaction.
- CPU loading.
- Interrupt operations, both software and hardware generated.
- The effects generated by aperiodic signals (if relevant).

This is where you really do need a good RTOS-aware testing tool.

And now for a major tenet of RTOS testing; do not do it in an ad-hoc way (that is, don't make it up as you go along). You need to devise a test strategy before you implement the design; base this on a set of carefully considered test cases. Each case should be defined in terms of pre-conditions, goals and outcomes (post-conditions). Apart from this very little general guidance can be given; much depends on individual systems. One thing is clear though; testing must be as comprehensive as possible.

It is relatively straightforward to test the example system of figure 12.17. Now consider the system shown in figure 12.18. The fact that it has many more external devices, tasks and task components will clearly increase the testing effort. But this system is really a much more complex one as it involves significant dynamical behaviour, see figure 12.19.

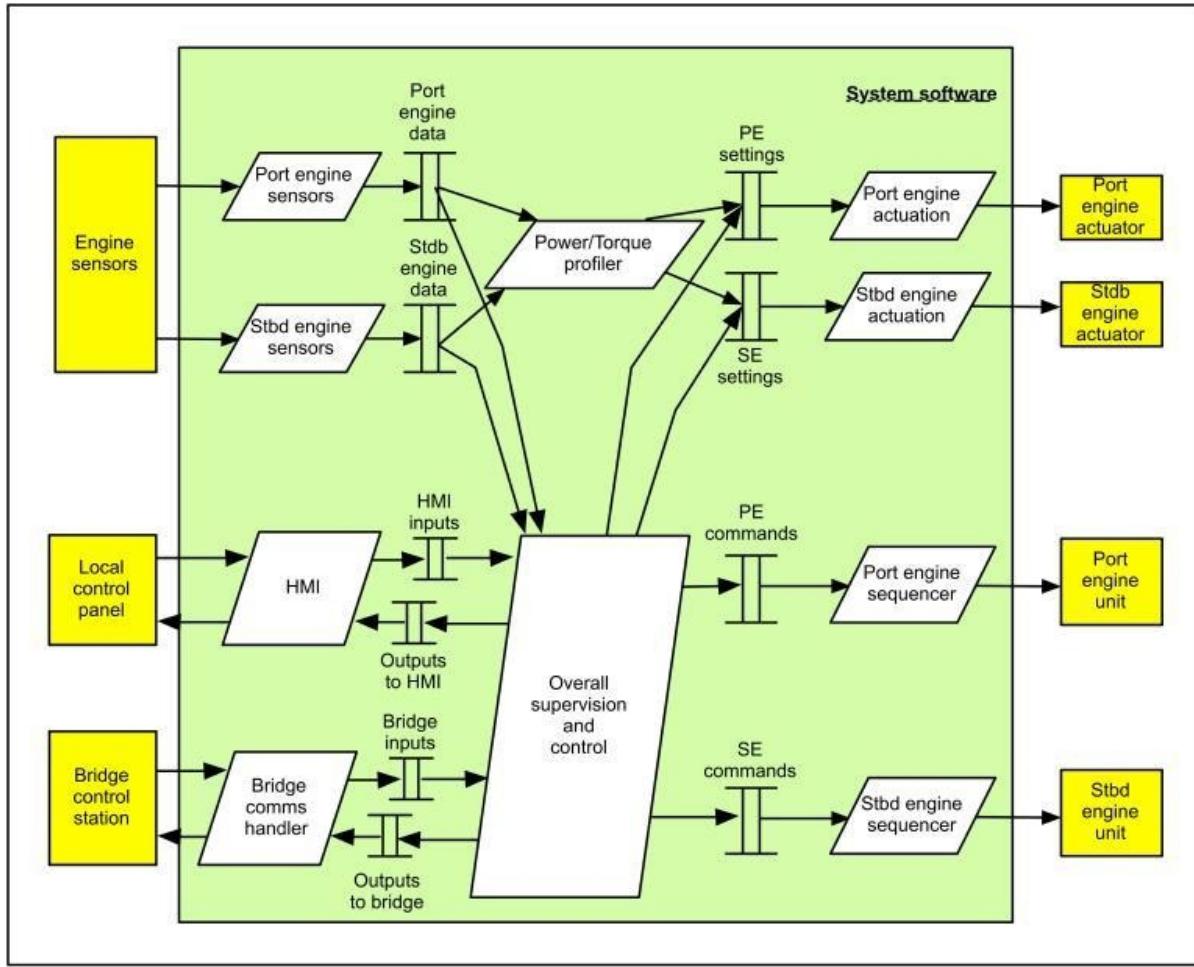


Figure 12.18 Reference system 2 tasking diagram

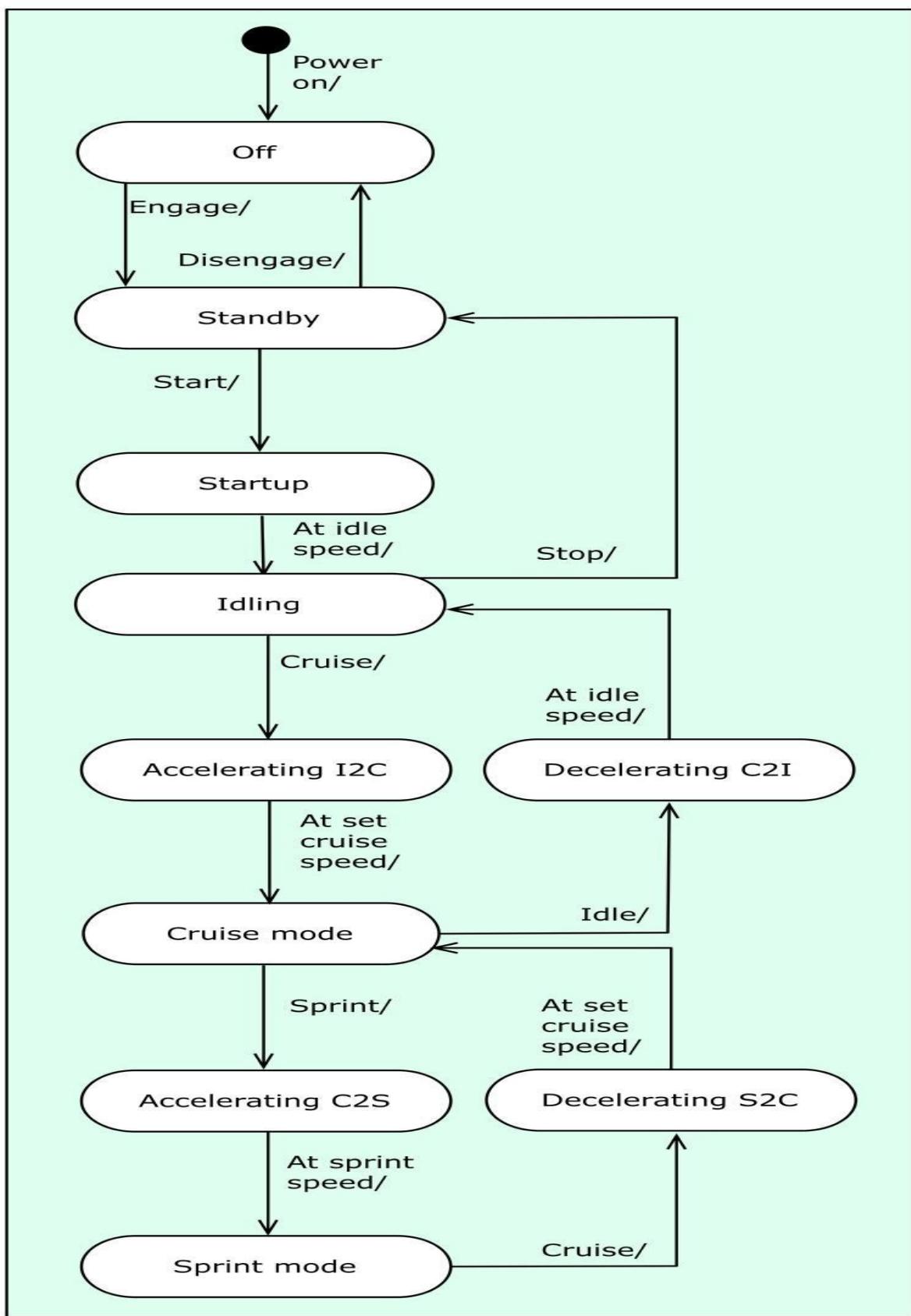


Figure 12.19 Overall supervision and control task state diagram - normal mode (part)

Here the dynamical behaviour of the Overall supervision and control (OSC) task is specified using a state diagram. And note, this describes only part of the overall dynamics; the full system is much more complex. Now what does imply for the test effort? It's going to be much more difficult and will not be done quickly! Observe that many, many logical decisions must be checked to confirm the functional and temporal correctness of the design.

Because of this complexity it makes good sense to do initial testing using software-generated signals. This can, if appropriate, be gradually extended to involve the external hardware devices. Eventually you can progress to full hardware-based testing techniques.

In this particular example we can exercise the OSC task software by:

- Forcing it to run through many different paths and
- Checking resulting responses, then
- Comparing these with predicted outcomes.

Conventionally this is done by the use of a test harness. However, rather than instrumenting the source code, a better solution might be to use a test task. The primary requirements of such a task is that it can:

- Stop and start the application tasks.
- Gather data from the various task communication components.
- Inject messages into queues.
- Control the data held in pools.
- Interrogate and control semaphores, mutexes and monitors.
- Enable the developer to control the test process via screen/keyboard units.

It is important that the test task itself makes minimal impact on the overall performance. Thus it should be run at a very low priority.

This approach to testing has two major advantages. First, it isn't necessary to instrument the application code. Thus the tested code is exactly the same as that deployed in the final system. Second, the test task software can be completely removed when testing has finished. Alternatively, the task itself could just be killed off (deleted), being resurrected (re-created) if needed for future work.

And don't forget; your in-target test tool is the key to the recording and display

of run-time behaviour.

During this phase problems may surface, which means that debugging must be carried out. Fortunately our work will be made much easier if we've followed the steps given earlier. Faults are unlikely to be caused by the behaviour of the individual tasks; the prime suspects here are issues relating to task interactions and timings. The debugging may turn out to be a long and difficult process; once again a good tool will make life easier and help you retain your sanity.

Review

You should now:

- Understand clearly the objectives of the testing of concurrent software and the procedures and tools used in such work.
- Know how to implement a professional testing regime.
- Understand the role and use of host, host-target and target system test and development environments.
- Appreciate the distinction between RTOS-oriented and application-oriented tools used in the monitoring and analysis of run-time code.
- Have a general appreciation of the nature and representation of the in-target run-time data collected by monitoring and analysis tools.
- Know that practical in-target tools fall into a number of categories and be able to choose the one most suitable for your development work.
- Be able to define a general test and debug strategy for your projects.
- Know why it essential to have tested verified and validated all code units (to the best of your ability) before running them in a multitasking environment.
- Understand why it can be very difficult to test and debug tasks that implement complex logic.
- Understand how a test task can significantly aid the test and debug process.

Chapter 13 Epilogue

This section brings together a number of items of more general interest but which don't belong to any one chapter.

13.1 Tasks, threads and processes

13.1.1 General aspects

Ah, the great debate; what are tasks, threads and processes? Well, if anybody tells you that they have THE correct definition, your response should be a somewhat cynical one. The reality is that there are multiple definitions out there in the software world, but no uniquely acceptable one. Partly the problem is due to the differing historical developments of mainframe, mini and microcomputers. Partly it is due to the fashionistas of the software world who regularly invent new words to replace perfectly good existing ones (summarized as 'old wine in new bottles').

Up to this point we've mainly considered the words task, thread and process to be synonymous. Now we'll broaden this out to provide a more general set of definitions, ones that work for me.

13.1.2 Code execution in an embedded environment - a simple-man's guide

First, a brief aside for those not so familiar with the details of processor operations: a simple description of program execution in a typical embedded environment. Here 'ROM' is used as a general term to denote non-volatile storage, 'RAM' denoting volatile storage. What follows applies to the great majority of embedded microcomputers, good enough for our explanation.

When an embedded micro is powered up the software must begin to execute without any operator intervention. This means, of course, that the program code must be held in ROM. And remember, we're describing operations at the electronic level. On start-up the processor automatically generates a specific memory address, which is where the first program instruction should be stored. Following this is a memory read, which loads the ROM information into the CPU. The microcode of the machine then interprets this to produce the desired processor operations. On completion of these the next instruction is obtained from memory, interpreted by the microcode and the necessary operations carried out. This cycle repeats itself as long as the unit is powered up.

The process of loading and storing the code into the micro has traditionally been called 'PROM programming', though sometimes it is called 'downloading'. And note; such code must be stored in the correct ROM locations, the 'absolute' addresses. This information is first obtained from the source code by compiling

it, absolute addresses being defined by the location process. The locator also specifies the address of the RAM space to be used by the program transient items (e.g. program variables, stack values, etc.). The purpose of the locator is essentially to translate the relative addresses generated by the compilation process into the absolute addresses used in the micro. Thus the programming information contains the program code, the values of program constant items and also the values of initialised data variables. This combination, when loaded into the micro, **is here** defined to be an 'application'.

13.1.3 Software activities, applications and tasks

A software activity is here defined to be a combination of a freestanding software unit and its processing hardware. More precisely, it denotes a sequence of code in execution. Now, this simple statement is quite profound as it implies that activities are essentially different to other software artefacts. Consider, for example, the system shown in figure E.1 (first described in chapter 1).

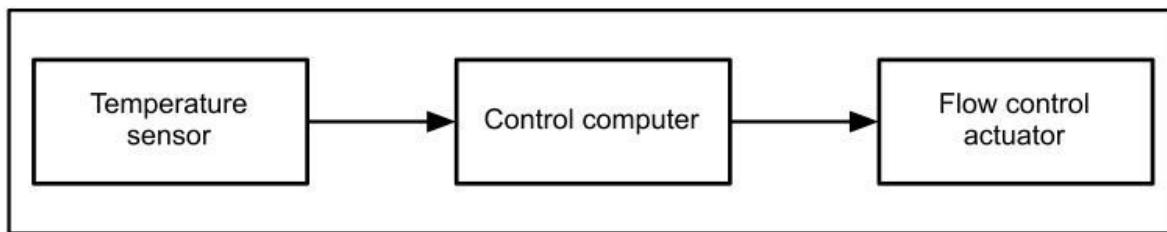


Figure 13.1 A simple processor-based real-time system

Assume that the corresponding program is formed as a single program unit, the essential aspects being (listing 13.1):

```

int const EngineTopTemp    = 700; /* 700 degrees centigrade */
int const LoopSampleTime   = 100; /* 100 milliseconds */
int EngineTemp;
int ConditionedEngineTemp;
int EngineTopTemp;
int ActuatorControlSignal;

void main (void)
{
    while(1) /* infinite loop */
    {
        MeasureEngineTemp(&EngineTemp);
        ConditionTempSignal(&EngineTemp, &ConditionedEngineTemp);
        ComputeControlSignal(&ConditionedEngineTemp, &EngineTopTemp,
                             &ActuatorControlSignal);
        SetActuatorPosition(&ActuatorControlSignal);
        DelayUntil(&LoopSampleTime);
    } /* end while loop */
} /* end main */

```

Listing 13.1 Source code of the control computer software

It's important to realize that this is only the code of the program. Even when it is compiled, linked and downloaded it still merely represents program code and associated data items, the *application*. Only when it begins executing does it have meaning from an activity point of view; it becomes an 'active' program unit. Another point of view is that an activity represents behaviour at the electronic level. Thus we can model the run-time system as shown in figure 13.2. where the active unit is represented by the parallelogram symbol.

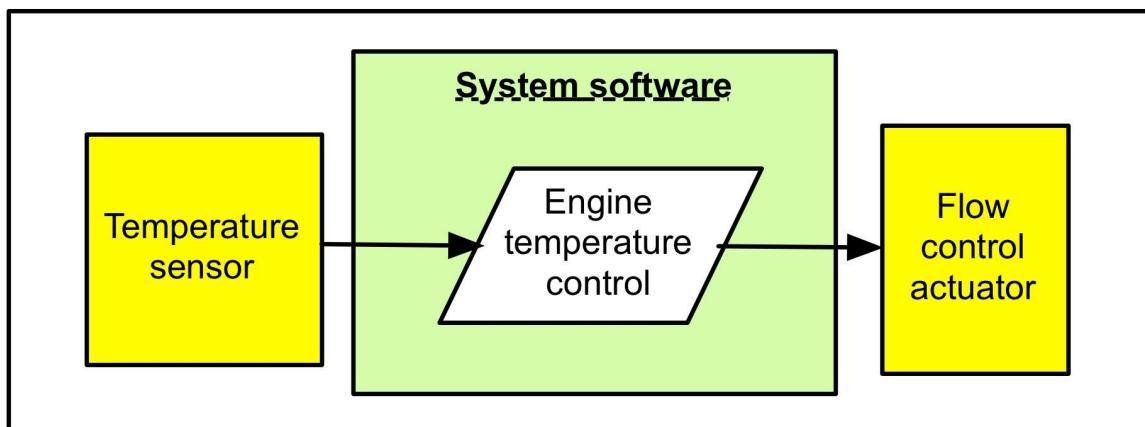


Figure 13.2 Activity model of the control computer software

Now consider the more complex multiple activity structure of figure 13.3. Suppose this was implemented on a five-processor multicomputer system without the use of an operating system. In this case we'd expect to have five programs (similar in structure to that of listing 13.1), each one being freestanding on an individual processor.

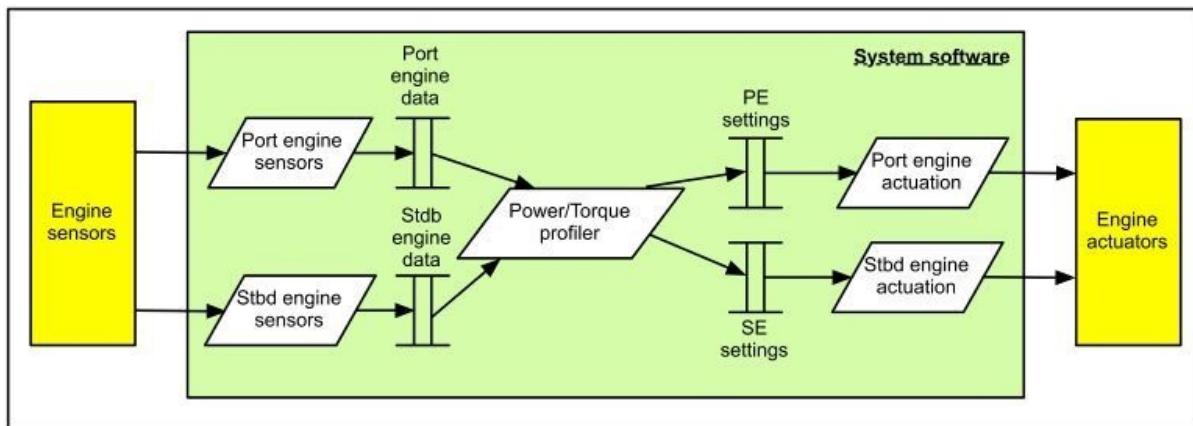


Figure 13.3 Example task-based design

Thus the software is formed as a set of cooperating concurrent units, each unit being an activity. However, the word 'activity' is not generally used in the software world; historically such constructs were called 'processes'. Moreover their structures and attributes were, in the early days, driven by the needs of large multi-user systems. These needs are really quite different from those of the real-time embedded world. It has been suggested that the suppliers of the first generation RTOSs used the word 'task' instead of 'process' to highlight this difference. Whether this is true or not, we'll keep things simple by *not* using the word process. So with this in mind you can see that figure 13.2 depicts a single task system, figure 13.3 showing a multiple task one.

Suppose that we decide to house all software on a single processor instead of a multiprocessor. Does this change things? Clearly all tasks must now run in time-shared fashion, using an RTOS and/or interrupts as enabling mechanisms. But note that the terminology is exactly the same. For simplicity we'll define this to

be the 'RTOS classical model'.

Now we come to something that causes some confusion: the 'thread'. This term came from the Multics and Unix operating systems work, which takes us back to the 1960's. However, it didn't enter the real-time embedded world until the late 80's when some vendors began to call the concurrent units threads instead of tasks. We've now reached a situation where the terms, for the classical RTOS model, are used synonymously. While it would be sensible to use just one name we have to live with the world as it is (and not as we'd like it to be). As long as you understand what the RTOS actually does it isn't really a problem.

13.1.4 Concurrency within tasks in a single processor system

Here we're concerned with the idea of running concurrent (strictly quasi-concurrent) units within tasks. In this discussion it's essential to realize that if a task is not running (e.g. is ready or suspended) its internal units cannot possibly run.

A crucial question; why would we even want to use concurrency within tasks? Well, the first thing to say is that in general you wouldn't do it in small embedded systems. The words pointless and overkill come to mind. So sensibly we're talking of using the technique in larger software systems. And in such systems a context switch save/restore is likely to involve much more than processor register information. We may, for instance, have to handle data relating to maths processors, MPU's, programmable devices, memory cards, database operations, etc. Hence the time and storage overheads could well be substantial.

Now, you can see this implies that we should minimize the number of tasks. Yet it may be better from a functional design aspect to go for a larger task set. Can we resolve this dilemma? In some cases we can, especially where concurrent units have strong functional and/or temporal cohesion. Here they may be grouped together to form what might loosely be called a super-task. These internal units we will call 'threads', the 'parent' being a task (this structure we'll define as the 'RTOS task/thread model'). As the threads are highly cohesive (they are all closely related), thread switching should have minimal context switch overhead.

Few commercial RTOS's implement the task/thread model. Further, thread scheduling tends to be simple, typically based on run-to-completion semantics.

13.1.5 Running multiple applications

From the previous work you can see that as far as the target system is concerned:

- An application denotes information carried in the target that is needed to execute the software. Thus it is a static item; it does not represent an active code unit.
- Tasks and threads are active code units.
- The code of an application can define either a classical or a task/thread RTOS model.

So what we've seen so far are actually single application models. However, in that context the application in itself isn't especially important; we know it's there but don't do anything with it. So we don't bother to model it. However, this all changes when we get to designs that consist of a set of applications (we've previously touched on this subject in chapter 6, 'Memory usage and management').

Up to this point all single application designs have provided a secure wall between concurrent units by using 'space partitioning' (i.e. memory space separation). With this, each task/thread has a defined memory area for storing its permanent and transient information, using ROM and RAM. These areas are separate and distinct, where necessary being policed by an MPU. Note though; if dynamic memory allocation is used tasks may well share RAM space. However this is not a problem provided secure memory allocation methods are used (as described earlier).

Let us now consider a situation where we might decide to structure our software as a set of applications. Normally we would use this approach in the design of large software systems, especially one that carries out a number of distinct jobs. For example, the software of a radar-controlled weapons system might be organized as shown in figure 13.4.

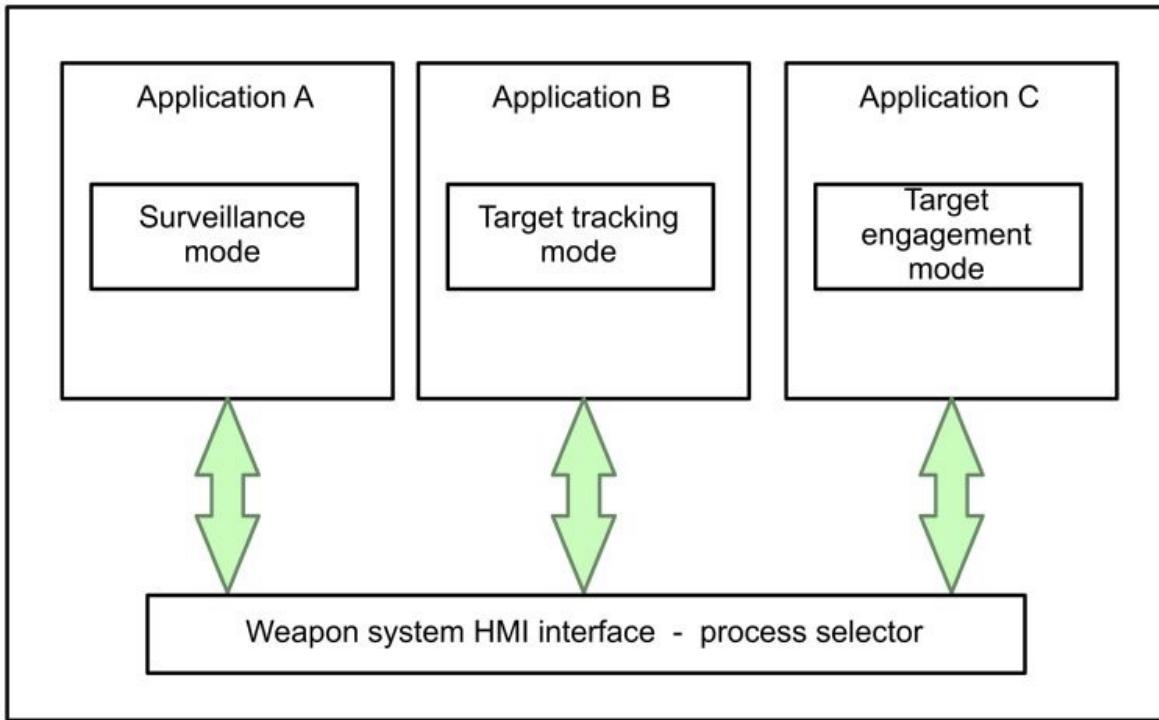


Figure 13.4 Running multiple processes

A key point is that one, and only one, application can be loaded onto the processor at any one time - they are truly mutually exclusive (this does not, of course, preclude them from sharing information). In the example here the processor would, by default, be loaded with the Surveillance mode application. Changes to the other modes are made in response to selections made by the system operator.

In modern embedded systems the applications are likely to be housed in a PROM store or a ramdisk. To run an application its code and data are first downloaded to RAM, all subsequent operations being RAM-based. This is done for reasons of speed; RAM operations are much faster than those of ROM. Normally the stored information uses logical addressing (see chapter 6) and so an MMU is needed to translate these to the physical ones. This combination we will call the 'RTOS application model'.

13.1.6 Summary

A brief summary of the types and key attributes of the three RTOS models is

given in figure 13.5.

RTOS model →	Classical model	Task/Thread model	Application model
Defining feature →	Single application software structure containing multiple concurrent units.	Single application software structure containing multiple concurrent units ('parents') which may themselves contain concurrent units ('children').	Multiple application structure. Each process may be structured as a classical or a task/thread model.
Terminology used for concurrent units. →	Task and thread are synonymous.	'Parent' == Task. 'Child' == Thread.	Depends on the RTOS structure used within the tasks. No new terminology.
Usage →	Extremely widely used.	Limited use in RTE's but supported by POSIX.	Found usually in the larger software systems.

Figure 13.5 Summary of RTOS models

It would be naïve to expect that everybody will agree with these definitions. So be it; we're not out to save the world. What is important to understand is that the structures defined in figure 13.5 are what you'll find in real-time embedded systems. If you wish to use different terminology, well, that's your choice. Just define what you mean and then be consistent in your usage.

13.2 Running multiple diverse applications - time partitioning

We are now in a position to look at how we can run multiple applications on the same processor where each application has its own OS. Key to this is the separation of software units using time partitioning (not to be confused with task time-sharing), figure E.6. Here we have three applications, each one being run in a specific time slot by a partition switcher (often called a partition scheduler). Each partition is allocated a specific amount of run-time, called a minor frame time. During this period only its application can run. Partitions are activated in sequence, the sequence being repeated cyclically at fixed time intervals defined as the major frame time. Note the similarity to the mechanism of rate group scheduling.

Another significant advantage of this technique is that we are free to use different RTOS's in each application, as depicted in figure 13.6. This allows us to mix critical, non-critical, bespoke and commercial-off-the-shelf RTOS's in a single computer. As a result we can run quite diverse applications on the same processor with a very high degree of security. It is, in fact, the basis of critical RTOS-based implementations on the A380 airbus.

As shown here a single application is executed in each time partition. However it is perfectly practical to run multiple applications in a single time slot.

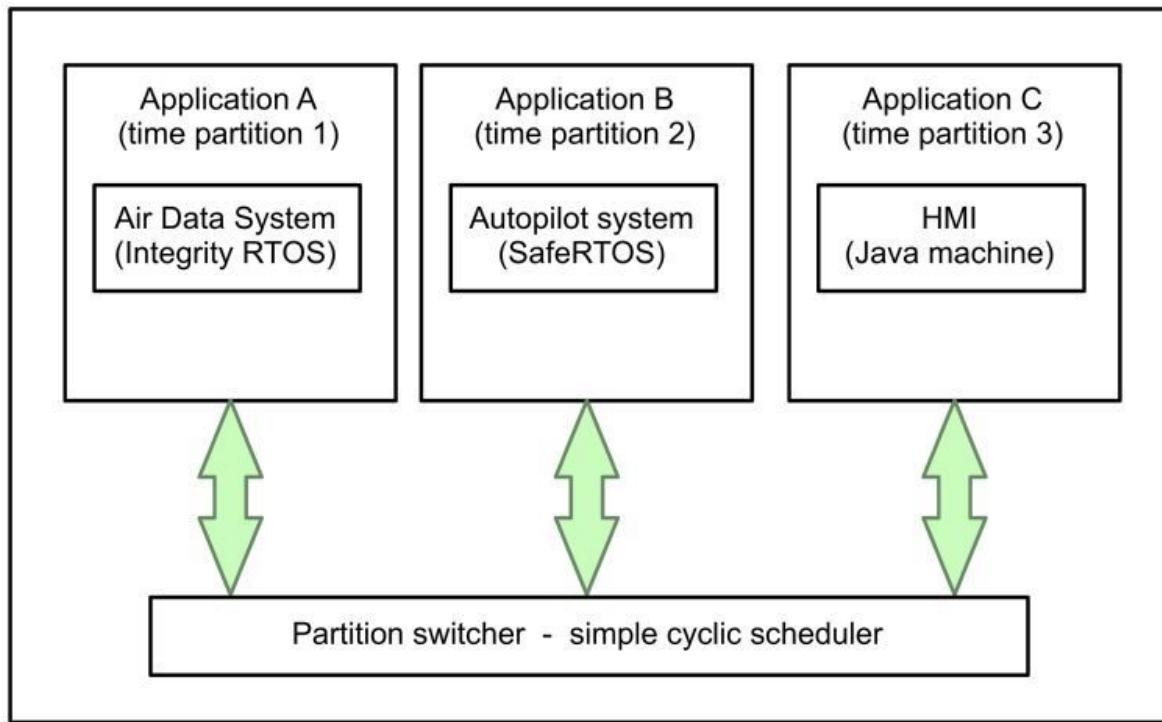


Figure 13.6 Time partitioning of multiple applications

13.3 RTOS's vs. general-purpose OS's.

I've often been asked what the difference is between a real-time operating system and a general-purpose (GP) one. This used to be quite clear-cut but, with improvements to GPOS's, has become less so. As a generalization the key differences lie in the following areas:

- Size (amount of ROM and RAM needed).
- Suitability of scheduling policies (at the very minimum a priority pre-emptive scheme is needed).
- Provision of priority inheritance schemes.
- User-accessible interrupt functions (ability to develop own ISR's).
- Provision of high-resolution timers (in some cases sub-millisecond operations need to be supported. Generally a high performance classical RTOS will provide a resolution down to 1ms).
- Provision of memory protection mechanisms (i.e. MPU or MMU).
- Provision of timeouts on operations for increased security of operation (e.g. waiting on semaphore).
- Efficient and timely aperiodic event handling.
- Ability to interface to real-world devices

However, you really need to assess your own specific needs against those of GPOS's; not all features are needed in all systems. A useful guide to categorizing embedded systems has been produced in the IEEE POSIX standards (specifically, Portable Operating System Interface Standard IEEE 1003.13). This has defined four types that are often found in real-time embedded systems, figure 13.7.

<u>POSIX Profile</u>	<u>Threads</u>	<u>Multiple applications</u>	<u>File system</u>	<u>Hardware model</u>
<u>Profile 51</u> Minimum real-time system	Yes	No	No	Small deep-embedded system – single processor, no MMU, no <i>user</i> I/O device.
<u>Profile 52</u> Real-time controller system	Yes	No	Yes	Special-purpose controller system – no MMU, file storage may be in RAM (e.g. ramdisk), user I/O device(s).
<u>Profile 53</u> Dedicated real-time system	Yes	Yes	Yes	Large embedded system – one or more processors, may have MMU.
<u>Profile 54</u> Multi-purpose real-time system	Yes	Yes	Yes	Large real-time system – one or more processors, may have MMUs, networking, user I/O devices, mix of real and non-real-time tasks, interactive user tasks.

Figure 13.7 POSIX real-time profiles

13.4 Bibliography and reference reading material.

One of the problems with referenced topics is that as time goes by they often become less relevant and/or out of date. Much useful material can now, of course, be found on the internet. The problem here though is that web sites are often shut down or the material just isn't updated. So to provide accessible, relevant and focussed information we have decided to put it on the Lindentree website, www.lindentreeuk.co.uk.

Index

A

[Absolute addresses](#)

[Abstract processor](#)

[Abstract task](#)

[Accumulated execution time](#)

[Active program unit](#)

[Aperiodic functions](#)

[Application programs](#)

[Application tasks](#)

[Application-level code](#)

[Asymmetric multiprocessing \(AMP\)](#)

[Asymmetric multiprocessor](#)

[Asymmetric multiprocessor - example software allocation](#)

B

[Background processing](#)
[Bad block management in solid-state drives](#)
[Base-level tasks](#)
[Benchmark - Dhrystone](#)
[Benchmark - EEMBC](#)
[Benchmark - Hartstone](#)
[Benchmark - SPEC](#)
[Benchmark - Whetstone](#)
[Benchmarking](#)
[Benchmarks - computation performance](#)
[Benchmarks - representative \[1\] \[2\]](#)
[Benchmarks - Synthetic](#)
[Benchmarks - synthetic](#)
[Benchmarks, representative - context switch time](#)
[Benchmarks, representative - inter-task message latency](#)
[Benchmarks, representative - interrupt response latency](#)
[Benchmarks, representative - pre-emption time](#)
[Benchmarks, representative - process dispatch latency](#)
[Benchmarks, representative - semaphore shuffle time](#)
[Benchmarks, synthetic - baseline test data](#)
[Benchmarks, synthetic - basic requirements](#)
[Benchmarks, synthetic - test categories](#)
[Benchmarks, synthetic - test stressing methods](#)
[Binary semaphore](#)
[Blocked state \(task\)](#)
[Board Support Package \(BSP\)](#)
[Bound multiprocessing \(BMP\)](#)
[Busy-wait state](#)

C

[Cache memory](#)
[Calendar time recording](#)
[Circular buffer](#)
[Circular dependency](#)
[Closely-coupled multicomputer](#)
[Context \(of task\)](#)
[Context switch time](#)
[Context switching](#)
[Cooperative scheduling](#)
[Coordination - condition flag groups](#)
[Coordination - definition](#)
[Coordination - simple condition flag](#)
[Coordinator software](#)
[Counting semaphore](#)
[Create Task API](#)
[Cycle time](#)

D

[Dangling pointers](#)
[Data pool](#)
[Data processing based structuring of software](#)
[Data queue](#)
[Deadlock avoidance](#)
[Deadlock example](#)
[Deadlock preconditions](#)
[Deadlock prevention](#)
[Deadlock prevention - allowing request pre-emption](#)
[Deadlock prevention - controlling resource allocation](#)
[Deadlock prevention - simultaneously sharing resources](#)
[Deadlock prevention strategies](#)
[Deadlocks](#)
[Deadlocks - overcoming](#)
[Deferred server](#)
[Deployment code model](#)
[Dispatcher](#)
[Distributed software designs](#)
[Distributed systems - communication and timing](#)
[Distributed systems - federated structure](#)
[Distributed systems - integrated structure](#)
[Distributed systems - physical organization](#)
[Distributed systems - proxies](#)
[Distributed systems - safety](#)
[Distributed systems - software architectural issues](#)
[Downloading to target system](#)
[DRAM](#)
[Due time](#)
[Dynamic memory allocation - problems](#)

E

[EPROM](#)

[Event flags](#)

[Execution engine](#)

[Executive \(RTOS\)](#)

F

[FIFO scheduling](#)

[Flash memory](#)

[Functional structuring of software systems](#)

G

[Garbage collection](#)

[GP embedded system - library routines](#)

[GP embedded system - system interrupt service routines](#)

[GP embedded system- special interrupt service routines](#)

H

[Hardware Abstraction Layer \(HAL\)](#)

[Head of queue](#)

[Heap memory](#)

[Heterogeneous AMP](#)

[Hold and wait](#)

[Homogeneous AMP](#)

I

[Idle task](#)
[Initialization code](#)
[Inter-task communication in distributed systems](#)
[Inter-task communication overheads](#)
[Interrupt details and overheads](#)
[Interrupt dispatch latency](#)
[Interrupt latency](#)
[Interrupt-based execution engine](#)
[Interrupt-based multitasking](#)
[Interrupts and the tick](#)
[Intertask communication](#)

J

Jitter - task

K

Kernel

L

[Latency](#)

[Laxity](#)

[Locator](#)

[Logical software design model](#)

[Loosely-coupled multicompiler](#)

M

[Mailbox](#)

[Mask PROM](#)

[Memory access - simple control mechanism](#)

[Memory allocation/deallocation - secure methods](#)

[Memory blocks - dynamic memory allocation](#)

[Memory control block](#)

[Memory device wear-out](#)

[Memory devices - bit flipping](#)

[Memory devices - check bits](#)

[Memory fragmentation](#)

[Memory leakage](#)

[Memory management overheads](#)

[Memory partitions - dynamic memory allocation](#)

[Memory protection registers](#)

[Microkernel](#)

[Microkernel - data transfer functions](#)

[Microkernel - dynamic memory allocation functions](#)

[Microkernel - Mailbox functions](#)

[Microkernel - mutual exclusion functions](#)

[Microkernel - scheduling and control functions](#)

[Microkernel - synchronization functions](#)

[Microkernel - system set-up and special functions](#)

[Microkernel use in large GP embedded system](#)

[MIPS](#)

[Mixed-mode multiprocessing](#)

[MMU](#)

[Monitor](#)

[Monitor - simple](#)

[MPU](#)

[Multicore - anonymous resource](#)

[Multicore - dedicated resource](#)

[Mutex](#)

[Mutual exclusion \[1\] \[2\]](#)

[Mutual exclusion mechanisms](#)

[Mutual exclusion overheads](#)

[MWIPS](#)

N

[Nanokernel](#)

[Nanokernel - key subprograms](#)

[Network support - virtual circuit](#)

[NVRAM](#)

O

OTPROM

P

[Partition scheduler](#)

[Partition switcher](#)

[Periodic functions](#)

[Periodic operation \(Tp\)](#)

[Periodic server](#)

[Physical code model](#)

[Polling control](#)

[Polling of events](#)

[Pool \(data\)](#)

[Pre-emptive scheduling](#)

[Priority - dynamic](#)

[Priority - static](#)

[Priority ceiling protocol](#)

[Priority inheritance protocol](#)

[Priority inversion](#)

[Priority pre-emptive scheduling](#)

[Process descriptor](#)

[Process, software](#)

[Program overheads](#)

[PROM programming](#)

Q

[Quasi-concurrency](#)

R

[Rate groups](#)
[Rate groups - major cycle time](#)
[Rate groups - minor cycle time slots](#)
[Rate groups - processor loading in time slots](#)
[Re-entrancy \(of code\)](#)
[Ready queue](#)
[Real-world interfaces](#)
[Reschedule function](#)
[Residual time](#)
[Resource manager](#)
[Round-robin scheduling](#)
RTOS application model [\[1\]](#) [\[2\]](#)
[RTOS classical model](#)
[RTOS task/thread model](#)
[Run-time model](#)
[Run-to-completion semantics](#)

S

[Scheduling - cooperative](#)
[Scheduling - deadline monotonic](#)
[Scheduling - Highest Response Ratio Next \(HRRN\)](#)
[Scheduling - Shortest Job First \(SJF\)](#)
[Scheduling - tasks](#)
[Scheduling, dynamic - computation time](#)
[Scheduling, dynamic - deadline](#)
[Scheduling, dynamic - earliest deadline](#)
[Scheduling, dynamic - laxity](#)
[Scheduling in AMP systems](#)
Scheduling in BMP systems [\[1\]](#) [\[2\]](#)
[Scheduling in mixed-mode systems](#)
[Scheduling policies - categories](#)
[Scheduling, static - heuristic](#)
Scheduling, static - Rate Monotonic Analysis (RMA) [\[1\]](#) [\[2\]](#) [\[3\]](#)
[Scheduling, static - Shortest Job First \(SJF\)](#)
Scheduling, static - Shortest Response Time (SRT) [\[1\]](#) [\[2\]](#)
[Scheduling timing](#)
[Semaphore limitations](#)
[Set-and-test instruction](#)
[Simple cyclic scheduler](#)
Software activity [\[1\]](#) [\[2\]](#)
[Software allocation - symmetric and asymmetric multiprocessors](#)
[Software application](#)
[Solid-state drives - fixed media](#)
[Solid-state drives - removable media](#)
[Spare time \(Ts\)](#)
[SRAM](#)
[Suspend-wait state](#)
[Symmetric multiprocessing](#)
[Symmetric multiprocessing \(SMP\)](#)
[Symmetric multiprocessor](#)
[Symmetric multiprocessor - example software allocation](#)
[Synchronization - bilateral rendezvous](#)
[Synchronization - definition](#)
[Synchronization - unilateral](#)

Synchronization signals

System responsiveness

T

[Target system testing - RTOS-oriented tools](#)
[Task - forced release](#)
[Task - priority profiles](#)
[Task - ready list](#)
[Task - suspended list](#)
[Task abstraction](#)
Task arrival time (Ta) [\[1\]](#) [\[2\]](#)
[Task blocking](#)
[Task control block \(TCB\)](#)
[Task deadline \(Td\)](#)
Task execution time (Te) [\[1\]](#) [\[2\]](#)
[Task management overheads](#)
[Task response time \(Tr\)](#)
[Task self-release](#)
[Task starvation](#)
[Task suspension](#)
[Tasking model](#)
[Testing - debugging and performance monitoring](#)
[Testing - host system development tool suite](#)
[Testing - in-target toolset](#)
[Testing - JTAG](#)
[Testing - non-invasive](#)
[Testing - OCD](#)
[Testing - RTOS host/target cross-development tools](#)
[Testing - RTOS Integrated Development Environment \(IDE\)](#)
[Testing - run-time analysis](#)
[Testing - SWD](#)
[Testing - target system performance monitoring and analysis tools](#)
[Testing, multitasking - key measurements](#)
[Testing, multitasking - test task](#)
[Testing multitasking software - goals](#)
[Testing using dedicated hardware devices](#)
[Testing using host-system data storage facilities](#)
[Testing using on-chip data storage methods](#)
[Testing, visual display - CPU load graph](#)
[Testing, visual display - messaging information](#)

[Testing, visual display - reverse-engineered tasking diagram](#)

[Testing, visual display - run-time behaviour](#)

Tick [\[1\]](#) [\[2\]](#)

[Tick handler](#)

[Tick-driven scheduling](#)

[Time delay generation](#)

[Timed cyclic scheduling](#)

U

Utilization [\[1\]](#) [\[2\]](#) [\[3\]](#)

V

[Virtual machine](#)
[Virtual processor](#)

W

[Wear levelling in solid-state drives](#)