Chapter 1

Introduction to real-time systems

1.1 Chapter overview

The role of real-time software grows larger and larger, and in a competitive marketplace any marginal improvement in usability or performance, provided by more effective software, will give a significant sales advantage. This introductory chapter tries to outline the source of some of the problems which programmers and engineers are likely to encounter and provides a set of guidelines for identifying potential real-time systems by looking at a number of their characteristics. It also introduces associated key ideas through example applications which, at this early stage, may be more helpful than offering abstract principles.

1.2 Real-time systems development

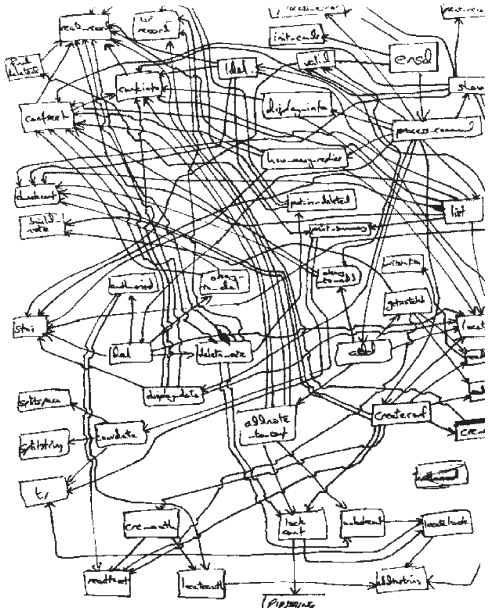
Real-time processing normally requires both parallel activities and fast response. In fact, the term 'real-time' is often used synonymously with 'multi-tasking' or 'multi-threading', although this is not strictly correct: small real-time systems, as used in dedicated equipment controllers, can perform perfectly adequately with just a simple looping program. Indeed, the period I spent developing commercial embedded systems taught me that such simplicity of design has much merit, and with the massive increase in processor speeds, it is now possible to use such crude software schemes for a much wider range of applications. As long as the response is *good enough*, no further complexities need be introduced. But, if a large number of different inputs are being monitored by a single processor, or the input data streams are complex and structured, the simple polling loop approach will prove inflexible and slow, and a multi-tasking solution will be required. Whatever style of implementation is chosen as appropriate, the need remains to deal with several concurrent activities over a period of time.

Real-time systems often seem like juggling

1.3 System complexity

A lot of the problems encountered with any software development involve 'complexity management'. Good practice, prior experience and team work are essential factors in achieving a successful outcome. Problems often appear impossible until they are subdivided, then each component part becomes much more manageable. Real-time software suffers from the same set of problems as traditional DP (Data Processing) applications, but adds the extra dimension of time to confuse the developer. To help in the preliminary analysis and design work, a rigorous method, which can be understood by all the team members, should be adopted. This will provide discipline and guidance. The main reason for undertaking design activity is to arrive at some well-structured code. Design without a subsequent implementation is mostly a futile activity. If you follow a good design technique, appropriate questions will emerge at the right moment, disciplining your thought processes.

A design method should provide intellectual guidance for system partitioning as well as documentation standards to ensure you record your decisions and supporting rationale. Without an effective method, you could end up with the complexity of a bramble patch, as illustrated opposite.

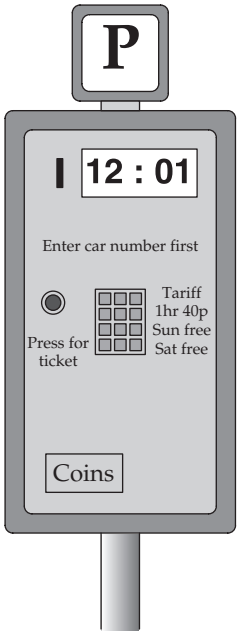


A *very* preliminary design schema illustrating complexity (thanks to Les Carleton)

Perhaps surprisingly, suitable alternatives for real-time systems design are not very numerous. In this text we have selected: Structured Analysis/ Structured Design (SA/SD), Concurrent Design Approach for Real-time Systems (CODARTS), Finite State Methods (FSM), and Object-Oriented Design (OOD) for study. The actual tools used to solve problems clearly constrain the set of solutions available, and so the choice of design method is vital.

1.4 Microprocessors and real-time applications

We are all familiar with real-time applications, they surround us in our everyday lives. Vending machines, mobile phones, alarm systems, washing machines, motor car engine controllers, heart monitors, microwave ovens, point-of-sale terminals, all operate courtesy of an embedded microcontroller running dedicated software. Before microprocessors appeared in the late 1970s, such functionality, in as far as it was possible, was conferred by electronic circuits often built using 7400 series TTL logic packs. Each application required a completely different circuit to be designed and manufactured. This was not an attractive prospect for equipment suppliers who were struggling to control their expanding warehouse stock levels, inflated by the gush of new silicon products. The arrival of embedded software, which allowed many different applications to share the same hardware, was most welcome. The term



A familiar real-time application

'real-time' is also used in the USA to describe on-line terminal services such as ATMs (Automatic Teller Machines, or cash dispensers), database enquiry, and on-line reservation and payment systems. Recently the term 'responsive system' has been introduced to further distinguish such computer-based applications. The list expands as technology elaborates. In practice, all computer systems have some aspects which are relevant to real-time programming and so the specific skills presented in this text are in great demand.

1.5 Definition of a real-time system

Although there is no clear dividing line between real-time and non-real-time systems, there are a set of distinguishing features (listed below) which can assist with an outline classification schema to identify real-time applications.

- *Timing* The most common definition of a real-time system involves a statement similar to 'Real-time systems are required to compute and deliver correct results within a specified period of time.' Does this mean that a *non-real-time* system such as a payroll program, could print salary cheques two years late, and be forgiven because it was *not* a real-time system? Hardly so! Obviously there are time constraints on non-real-time systems too. There are even circumstances in which the

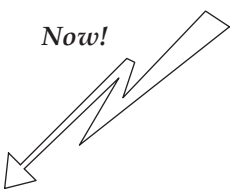
- Specified limit on system response latency
- Event-driven scheduling
- Low-level programming
- Software tightly coupled to special hardware
- Dedicated specialized function
- The computer may be inside a control loop
- Volatile variables
- Multi-tasking implementation
- Run-time scheduling
- Unpredictable environment
- System intended to run continuously
- Life-critical applications

Outline real-time categorization scheme

early delivery of a result could generate more problems than lateness of delivery. A premature newspaper obituary could sometimes create as much havoc as an early green on a traffic light controller.

Response time sensitivity

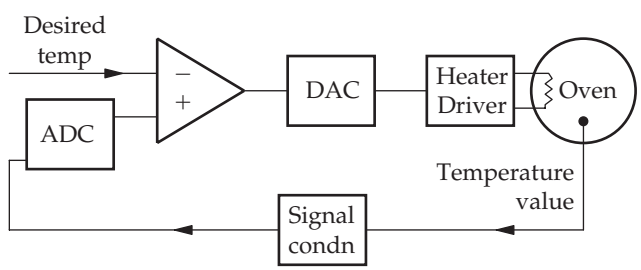
- *Interrupt driven* After the requirement for maximum response delay times, the next characteristic of real-time systems is their involvement with events. These often manifest themselves in terms of interrupt signals arising from the arrival of data at an input port, or the ticking



Event-driven pre-emption

of a hardware clock, or an error status alarm. Because real-time systems are often closely coupled with special equipment (a situation that is termed 'embedded') the programmer has also to gain a reasonable understanding of the hardware if the project is to be a thorough success. Once again, however, the demarcation between traditional data processing and real-time systems is not easy to draw because *event-driven* GUI interfaces are so widely used within all desktop applications.

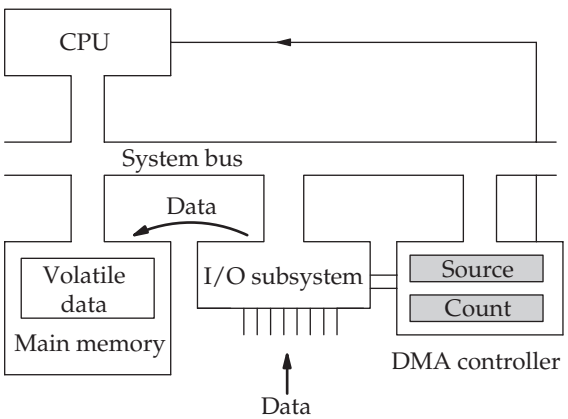
- *Low-level programming* The C language is still favourite for writing device drivers for new hardware. But because high-level languages, including C, do not generally have the necessary instructions to handle interrupt processing, it has been common for programmers to drop down to assembler level to carry out this type of coding. Because ASM and C are classified as low-level languages by many programmers, who may be more familiar with database systems and windowing interfaces, it has been suggested as a distinguishing characteristic of real-time programmers that they prefer to use low-level languages. This can be seen as somewhat misleading, when the real-time high-level languages Modula-2 and ADA are taken into consideration.
- *Specialized hardware* Most real-time systems work within, or at least close beside, specialized electronic and mechanical devices. Unfortunately, to make matters more difficult, during development these are often only prototype models, with some doubt surrounding their functionality and reliability. This is especially true for small embedded microcontrollers which may even be required to perform as critical component parts within a feedback control loop. The oven power controller illustrated below could employ an integrated microcontroller to monitor the oven temperature and adjust the electrical power accordingly. Such applications place a heavy responsibility on the programmer to fully understand the functional role of the software and its contribution to the feedback delay which governs the system response. Code may have to run synchronously with the hardware or other software systems, such as when telephone transmissions are sequenced 8000 times a second to maintain acceptable voice quality. Very often this leads the programmer



Feedback control loop for specialized hardware

into other disciplines: electrical theory, mechanics, acoustics, physiology or optics. Real-time programmers rarely have a routine day.

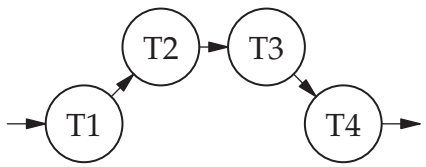
- *Volatile data I/O* Another special issue for real-time software concerns 'volatile data'. These are variables which change their value from moment to moment, due to the action of external devices or agents, through interrupts or DMA. This is distinguished from the situation where input data is obtained from a disk file, or from the keyboard under program control. The most common example encountered by real-time programmers involves input channels which operate autonomously to bring in new values for memory variables when data arrives at an input port. The software must then be structured to check for changes at the correct rate, so as not to miss a data update.



Volatile variables with a DMA controller

- *Multi-tasking* Real-time systems are often expected to involve multi-tasking. In this situation, several processes or tasks cooperate to carry out the overall job. When considering this arrangement, there should be a clear distinction drawn between the static aggregation of groups of instructions into functions for compilation, and the dynamic sequencing of tasks which takes place at run-time. It has already been suggested that full multi-tasking is not always necessary, but it can be positively advantageous to programmers in simplifying their work. It is also widely accepted that many computer systems have become so complex that it has become necessary to decompose them into components to help people to understand and build them. In the traditional data processing field, for example, the production of invoices from monthly accounts requires several distinct operations to be carried out. These can be sequenced, one after the other, in separate phases of processing. With

real-time systems this is rarely possible; the only way to partition the work is to run components in parallel, or concurrently. Multi-tasking provides one technique which can assist programmers to partition their systems into manageable components which have delegated responsibility to carry out some part of the complete activity. Thus, multi-tasking, although generally seen as an implementation strategy, can also offer an intellectual tool to aid the designer.

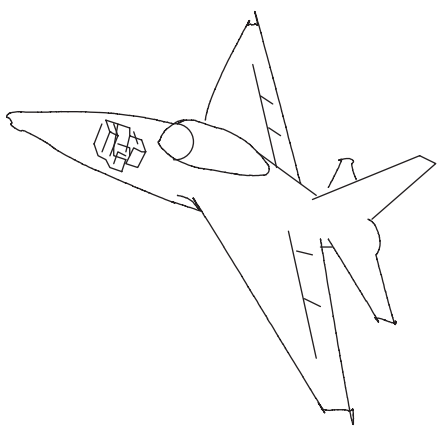


Component sequencing

- *Run-time scheduling* The separation of an activity into several distinct, semi-autonomous tasks leads to the question of task sequencing. In traditional DP applications the sequence planning is largely done by the programmer. Functions are called in the correct order and the activity is completed. But for real-time systems this is only half the story. The major part of sequencing takes place at run-time, and is accomplished by the operating system through the action of the scheduler. It is as if the sequencing decisions have been deferred, it is a kind of 'late sequencing', to draw a parallel with the established term 'late binding', used with regard to code linking. This is perhaps the most interesting feature of real-time systems. The manner in which the various activities are evoked in the correct order is quite different from that of a traditional DP system which normally relies on the arrival of data records from an input file to sequence the functions, and so it is predetermined and fixed.
- *Unpredictability* Being event driven, real-time systems are at the mercy of unpredictable changes in their environments. It is just not feasible to anticipate with 100 per cent certainty all the permutations of situations which may arise. In my experience, the worst offenders are actually the human users, who seem totally unable, or unwilling, to understand what the designer intended. Any choice offered by a menu or sequence of YES/NO alternatives will soon reveal unexpected outcomes during field trials. The exact ordering or sequencing of all the functions which deal with these interactions has to be decided at run-time by the scheduler, giving much more flexibility in response. Considerable effort is now put into extensive simulation testing in order to trap as many of these bugs as possible, even before the designs are released.

Unpredictability

- *Life-critical code* Although not always the case, real-time systems can involve serious risk. A failure to run correctly may result in death or at least injury to the user and others. Such applications are becoming more and more common, with the aircraft and automobile industries converting their products to 'fly by wire' processor technology. This removes from the driver/pilot all direct, muscular control over the physical mechanism, relying entirely on digital control systems to carry out their commands. The burden of developing real-time, life-critical software, with all the extra checking, documentation and acceptance trials



Life risking applications

required, may raise the cost beyond normal commercial projects, of similar code complexity, by an astonishing factor of 30. Most real-time applications are intended to run continuously, or at least until the user turns off the power. Telephone exchanges, for example, contain millions of lines of real-time code, and are expected to run non-stop for 20 years. The increasing use of embedded microprocessors within medical monitoring and life-support equipment, such as radiological scanners and drug infusion pumps, makes consideration of software reliability and systems integrity even more urgent. Some research effort has been expended in devising a method to formally prove correct a computer program, much in the same way that mathematicians deal with algebraic proofs. So far, the products resulting from this work have not generated much commercial interest.

1.6 Programming structures

It is now well accepted that computer programs can all be broken down into three fundamental structures:

- Linear sequences of instructions
- Iterative loops of instructions
- Branches guarded by selection statements

But as indicated above, the *sequencing* of real-time code is not straightforward. In addition, multi-tasking code requires two more structures:

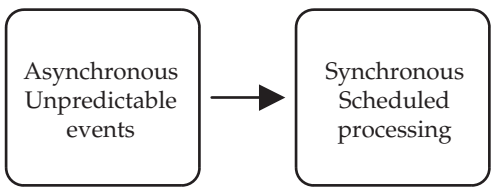
- Parallel or concurrent instructions
- Critical groups of exclusive instructions

More structures in real-time programs

While all DP systems may benefit from utilizing parallel or concurrent coding, it is rarely *essential*, as it frequently is in the case of real-time systems. This formally indicates the increased complexity that arises when working in the real-time field.

1.7 Response latency

There is also an interesting contradiction in citing 'minimum response delay' (latency) as the key factor when characterizing real-time systems. For example, when using more sophisticated real-time executives (RTE), the full response to a Receiver Ready or Transmitter Ready (RxRdy or TxRdy) interrupt is often deferred in order to balance the processing load. Thus the executive attempts to impose its own processing schedule on all the activities, which can actually result in a delayed response. This could be seen as transforming unpredictable, asynchronous demands into scheduled, synchronous processing.



Rapid response compromised for processing efficiency

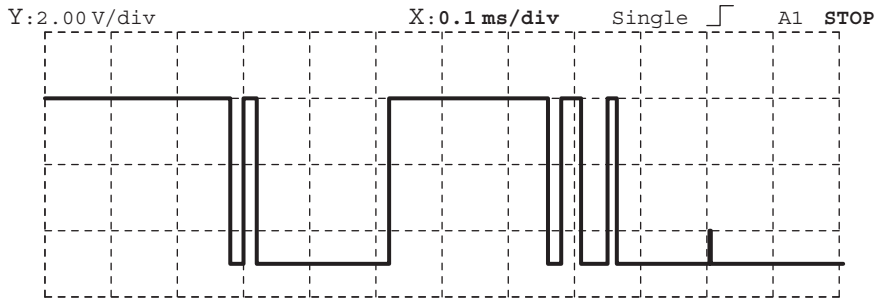
1.8 Relative speeds

1.8.1 Polling an input too fast

An important factor that needs to be clearly understood by newcomers to real-time programming is the vast disparity in speed between the modern, electronic computer and the human, physical world. Whereas even a slow microcontroller will zip through instructions at a rate of 10 million per second, humans can rarely handle a keyboard at two key strokes per second. The problem is due more to the *relative* speeds than their absolute values. Such an enormous disparity in speed leaves programmers in quite a quandary, since the voracious processing capacity of a modern CPU demands to be fed at all times!

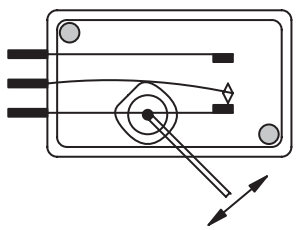
Consider the oscilloscope trace below, which shows how the output voltage changes when a microswitch is closed. The contact bounces for a period of up to one millisecond (1 ms, one thousandth of a second) before finally settling down. Humans are not aware of this high speed dithering, but a computer, sampling an input port one million times a second, can wrongly record that the switch has been turned on and off several times when it has only been pressed once.

Such errors often show up in 'monitoring and counting' systems and may lead to the use of more expensive optical or magnetic switch units which do not suffer from contact bounce.

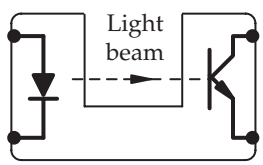


Voltage from a key switch showing a contact bounce of nearly 1 ms

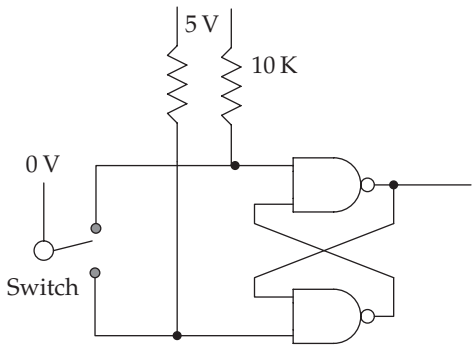
Alternatively, extra logic gates can be included to eliminate the effects of contact bounce as shown below. But perhaps the best solution is to deploy some debouncing software. This can subject the incoming, raw signals to low-pass filtering, at no extra expense. We will return to this issue in Chapter 4 with an example system.



Mechanical switch



Optical switch

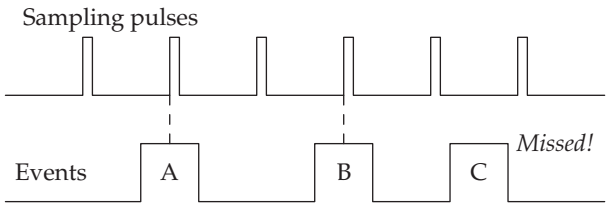


Debouncing logic

1.8.2 Polling an input too slowly

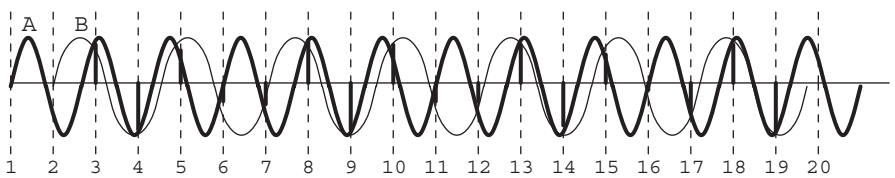
It scarcely needs to be said that if a computer checks an input too infrequently it runs the risk of missing an occasional event, such as a counting pulse. To avoid this happening, it is common to require the sampling rate to be at least twice as fast as the mean pulse frequency. If the system has to detect a pulse occurring no more often than every 10 ms, the port should be checked at least every 5 ms (200 times a second). Sometimes the input events are recorded on a hardware latch in an attempt to reduce the required sampling rate.

However, this still runs the risk of losing an event when a following event overruns the previous one before the software reads and clears the earlier event from the latch.



Sampling too infrequently

The term 'aliasing' is used to describe a similar situation which occurs when an analogue signal is sampled too slowly. If the input signal contains frequencies above *half* the sampling rate, the sampled version of the signal will appear to contain frequencies not in the original signal. Look closely at the figure below. The original signal ('A') is printed with a thick line and shows 12 cycles (∩∪). The sampling points are shown as dashed lines, with the captured values as thick vertical bars. Notice that there are fewer than the minimum two samples per signal cycle. There are only 20 samples in 12 cycles, whereas there should be at least 24. Now reconstruct the signal using only the sample values. The resulting synthesized wave ('B') is drawn with a thin line. 'A' and 'B' are not the same. This artifact is called aliasing and is avoided by filtering all the high frequency components from the original signal before sampling occurs. The maximum frequency threshold of half the sampling rate is referred to as the Nyquist limit. You may be familiar with old Hollywood films, where stagecoach wheels appear to turn backwards because the movie cameras ran too slowly.

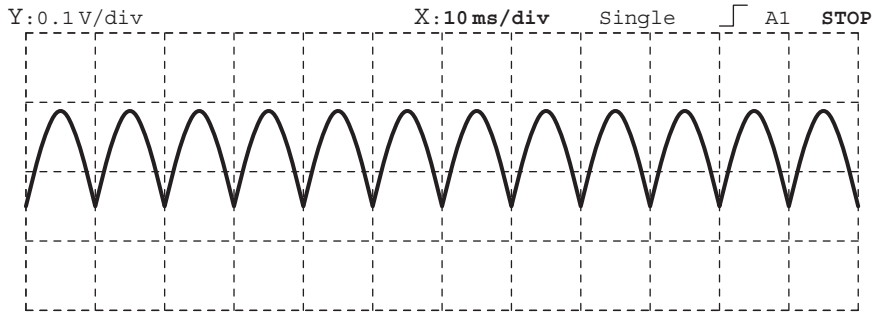


Aliasing error through sampling too slowly: only 20 sample points in 12 cycles

1.8.3 Light sensing

Another example problem, illustrating the relative timing issue, involves the use of light sensors. Normal office lighting relies on fluorescent tubes. These

actually flicker very strongly at 100 Hz. The human eye is normally insensitive to flicker rates above 40 Hz, but a surface inspection computer could easily be confused by large variation in illumination. If the program is periodically reading a value given by a photodiode, the exact moments when the samples are taken would have more influence on the result than the darkness of the surface being scanned. If the polling is carried out fast enough, say 5 kHz, the 100 Hz brightness modulation would get averaged out. Once again, the timing of computer activity is critical to obtaining a correct result.

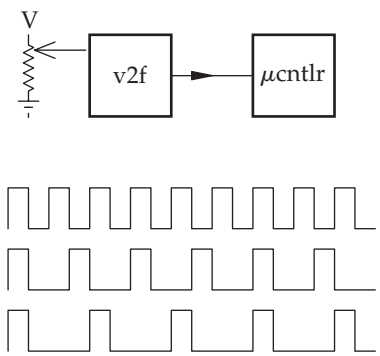


Voltage from a light sensor showing 100 Hz mains flicker

The application areas described above, switch scanning, pulse detection and light sensing, show that calling input routines too *frequently* or too *infrequently* can both generate artifacts which can blossom into serious errors.

1.9 Software timing

Another problem for programmers involved with real-time systems is the need to understand more exactly what the compiler is creating. With desktop systems it is now commonplace to write and run, with little attention being



Voltage-to-frequency converter

paid to the final executable code. There are circumstances where this optimistic disregard may lead to difficulties. A commonly used environmental monitoring arrangement involves a transducer being interfaced to a voltage-to-frequency converter (thanks to Laurence O'Brien for sharing this lop-sided bug with me). The cost advantage of not using an ADC interfaced to a serial transmission link is the prime motivation. With a V2F unit, the transducer analogue voltage is converted to a pulse frequency code: the larger the voltage, the higher the frequency; the lower the voltage, the lower the frequency. The computer only has to dedicate a single bit input port to accept the information in serial mode. However, there remains the problem of converting this pulse frequency code into normal integer format. For an HLL programmer the following code might appear attractive. It runs and offers the beguiling appearance of success, but it entails an interesting bug related to the code generated by the compiler. Unfortunately, the time spent in the two opposing arms of the `IF/ELSE` structure is not matched. So with an actual 50/50 situation, the results would not come out as 50/50, because of the dwell time bias. This can be checked by reversing the code and running both versions back to back. Inspecting assembler code listings from the compiler will also reveal the discrepancy.

```
loop for 100 msec {                          loop for 100 msec {
   if (input_bit) <------------------------>    if (!input_bit)
       hcount++;                                    hcount++;
   else                                          else
       lcount++;                                    lcount++;
}                                             }
temp1 = tempX*hcount/                         temp2 = tempX*hcount/
      (lcount+hcount)                               (lcount+hcount)
```

1.10 High speed timing

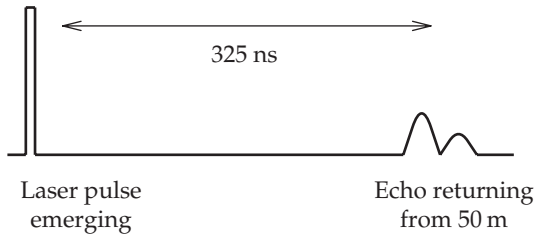
Perhaps an example would now be useful of the opposite situation, when processors simply cannot run fast enough. Consider a laser range-finder, intended for use in civil surveying, or more ominously for battlefield targeting. It works by preparing a pulse laser for firing, emitting a pulse of light, waiting for the reflected echo to return, and, by timing the duration of the flight, calculating the distance travelled.

The speed of light is 3×10^8 m/sec.

For a target 20 km away, the pulse of light will travel 40 km (4×10^4 m).

$$\text{So } \textit{time taken} = \frac{\textit{distance}}{\textit{speed}} = \frac{4 \times 10^4}{3 \times 10^8} = 1.3 \times 10^{-4}\,\text{s} = 130\,\mu\text{s}$$

If the item being surveyed is only 50 m distant, the time of flight will be reduced to 325 ns.

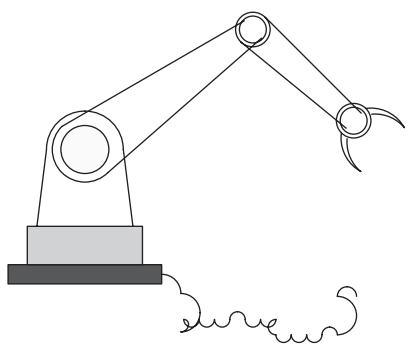


Light travels very fast!

Thus the timing mechanism must be able to cope with the range 0.3–150 μs. Instructions executed on a 500 MHz, dedicated processor could maximally complete instructions every 2 ns, with the code running from LI cache. However, any disturbance to the instruction fetch/execute pipeline sequence, such as cache elimination, task swapping, interrupts, or even conditional branches in the code, would reduce the instruction rate considerably. Therefore, the only reliable timing method for this application is to employ a high speed hardware counter which is cleared down and restarted when the light pulse leaves the laser, and stopped when the echo returns. The number captured is a measure of the distance travelled by the laser pulse, there and back. Only a close interplay of software and hardware can provide the solution to this problem.

1.11 Output timing overload

There is a similar set of timing problems facing the programmer when dealing with periodic, or cyclic, output data. A typical example involves the control of motors. These may be used in continuous mode, to turn wheels at a desired speed, or to position a unit and hold it against a varying resistant pressure.



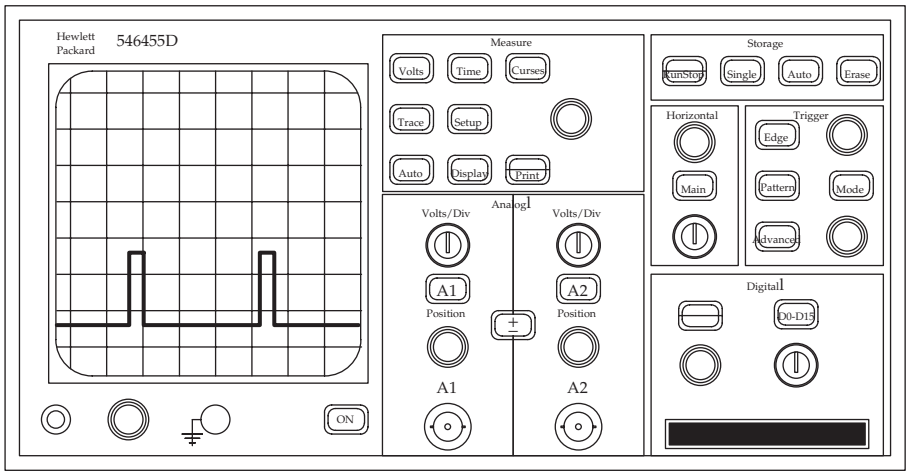
Motor drive problems

Both situations may involve responding to sensors providing feedback information. There are several different types of motor available, each with its own special area of application: stepper, DC servo, universal AC, induction AC, and synchronous AC. DC servo and stepper motors are most commonly controlled with microprocessors; the latter we will meet again in Chapter 2. Both DC servo and steppers can provide rotation and dynamic positioning. Stepper motors in particular require accurately timed sequences of pulses to control their speed and direction.

Microprocessor-based controllers can handle such a problem by holding pulse pattern tables in memory and accessing the entries in sequence at the correct rate. Another interesting type of positioning servo motor is supplied by Futaba for model makers. It also uses a digital pulse input to specify the required angular position of the motor shaft. Commonly, a 2 ms pulse will indicate a central, neutral position, a 1.5 ms pulse sets the shaft to $-45°$ and a 2.5 ms pulse sends the shaft to $+45°$. Unfortunately, unlike the stepper motor, the positioning pulses need to be repeated every 20 ms, to refresh the controller. This is quite a problem for a processor when several positioning units have to be serviced simultaneously, as is the case with a robot arm. Arranging for five timing pulses to be dispatched every 20 ms, with an accuracy of 50 μs, really does benefit from some special hardware support.

1.12 Debugging real-time systems

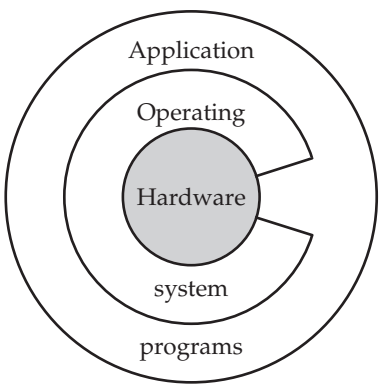
When debugging real-time code extra difficulties emerge, such as the impossibility of usefully single stepping through doubtful code, or reproducing elusive, time critical input situations. Inserting a neanderthal `printf( )` statement in an attempt to isolate the bug will completely change the execution timing (my aged PC/200 takes nearly 1 ms to complete a call to printf). Confusion often arises when dealing with prototype hardware. Errors can be blamed on the software, when in fact the problem is due to the new electronics. Such uncertainty makes debugging more difficult and challenging. Extra equipment may need to be acquired, by purchase, hire or loan, to generate complex test signals, and capture the results using sophisticated logic analysers, In Circuit Emulators (ICE) or digital storage oscilloscopes. Initially, a very useful trick is to insert a couple of output instructions within your code, which will emit a short indicator pulse from a spare output port. This can be picked up by the oscilloscope and viewed. It is an enormous advantage to be able to see the relative timings of ongoing processing activity, set against traces obtained from external events. When dealing with systems which are processing fast streams of data interleaved with intermittent, much slower events, capturing the correct sequences for analysis can be tricky. In this situation, you may be able to get your software to trigger the viewing trace, and so synchronize the oscilloscope display to the events under investigation.



Oscilloscopes can display timing information from software, too

1.13 Access to hardware

Because real-time computer systems are often working in tight integration with the surrounding equipment, they need to have efficient access to hardware. This means that the normal hardware/software separation, imposed by an operating system for security purposes, has to be broached. The application software must be able to directly read input ports and write to output ports. With Unix and Windows, these operations are forbidden to all but supervisor-level code. To run all the application tasks with supervisor permission would incur unnecessary risk, so special device driver routines are needed to provide the I/O facilities that real-time programs require. Operating systems can get in the way.



Direct access to hardware

1.14 Machine I/O

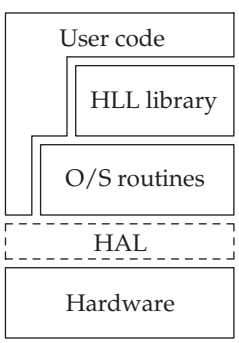
All machine instruction sets must include some mechanism allowing the programmer to transfer data into and out of the computer. To that end, Intel provides its CPUs with special `IN` and `OUT` instructions which operate solely on ports located within a designated I/O address space. In a more unified, von Neumann approach Motorola chose to avoid separate I/O instructions and address spaces, and so enabled programmers to use the normal group of data transfer instructions with I/O ports.

This is possible because all the ports are located within memory address space, alongside the RAM or ROM chips. From the CPU's perspective, ports, ROM and RAM can look much the same for access purposes. Only when data caching facilities are included does this homogeneity break down.

- Dedicated and periodic **polling**
- **Interrupt** driven
- Direct Memory Access (**DMA**)

Different I/O techniques

From the software point of view there are three principal techniques used to initiate and control the transfer of data through a computer I/O port. Direct Memory Access (DMA) is distinct in that it depends substantially on autonomous hardware which is required to generate the bus cycle control sequences in order to carry out data transfers independently of the main CPU. We will discuss each I/O method in greater detail later in this chapter. All require software driver routines to work closely with associated hardware units. These routines are normally part of the operating system and not infrequently written in assembly language. In the PC marketplace, extension card suppliers provide such driver routines on CD or floppy disk, along with the hardware, so that they may be installed by the user. It is also increasingly



Software access to hardware

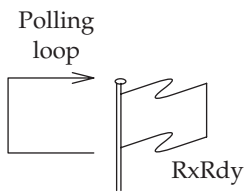
common to have access to driver routine libraries via the Internet. Following the pioneering example of Unix, modern operating systems are written as far as possible in HLL, probably C. In this way, porting the operating system to a new processor is faster and more reliable, once a good C compiler has been obtained. Windows NT has defined a specific hardware interface layer of software, HAL, which acts as a virtual machine layer to aid porting to new processors. The traditional view of software is a hierarchy of intercommunicating layers as presented above. Each layer has a specific data processing role and exchanges messages with adjoining layers.

HAL hides much of the specific hardware differences between Pentium, ALPHA and MIPS processors, from the main part of the operating system code, making it easier to port and maintain the system code. Although Windows 98 allows direct access to the I/O hardware, with Unix and Windows NT/XP it is strictly denied for security reasons. Such a limitation does not concern most application programmers who only ever access I/O facilities by calling library procedures provided with the HLL compiler, such as `getc( )` and `putc( )`. These library procedures may then call underlying operating system functions to gain access to the actual hardware.

The introduction of a 'virtual machine' software layer has also been used in the development of a version of Linux, RTAI, for real-time applications. We will discuss this more in Chapter 19.

1.15 Programmed I/O

The fundamental method of reading data from an input port involves the simple execution of either a MOVE or IN instruction, depending on whether the port is memory mapped or I/O mapped. An example of input by programmed polling from an I/O mapped port is presented in C and Pentium assembler code below. This would only work on a system running DOS or Windows 98 because Linux expressly denies direct access to hardware in this fashion for security reasons. Access to all port addresses is limited to processes running with root permissions, so if you have the supervisor password, and are prepared to risk a complete system rebuild should you inadvertently blunder into an unexpected port, you are free to try your hand! The Linux 'suid'



Spin polling

permissions, explained in Chapter 10, offer a middle path through the security quagmire. Operating system code handles all the I/O operations, so all the assembler-level IN and OUT instructions are hidden inside device driver routines. The receive ready flag (RxRdy) in the status register (STATUS) is repeatedly checked in a tight polling loop until it has been set to 1 by the port hardware, indicating that a new item of data has arrived in the data receive register (RxData). The loop then drops through and the newly arrived data byte is read from the data receive register. In this example, it is then checked for zero because this would indicate the end of the current data transfer. If it is non-zero, the item is saved into the data array using a pointer, and the loop continues.

```
do  {
    while (!(INP(STATUS) & RXRDY)) { };   /* wait for data */
}   while (*pch++ = INP(RXDATA));          /* check data for a NULL */
```

The ASM equivalent of the above code uses the Pentium `IN` input instruction and might look something like this. Again, the status port register is checked before reading the data port itself.

```
        MOV EDI,PCH             ;init pointer to start of data buffer
TLOOP:  IN AL,STATUS            ;read status port
        AND AL,RXRDY            ;test device status bit
        JZ TLOOP                ;blocking: if no data go round again

DATAIN: IN AL,RXDATA            ;data from Rx port & clear RXRDY flag
        OR AL,AL                ;test for EOS marker
        JZ COMPLETE             ;jmp out if finished
        MOV [EDI],AL            ;save character in data buffer
        INC EDI                 ;bump buffer pointer to next location
        JMP TLOOP               ;back for more input
COMPLETE: ....                  ;character string input complete
```

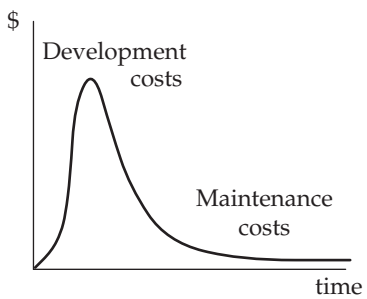
Example input polling loop in C and ASM code

It is also very important to understand that I/O port hardware detects the action of data being read from the data register, RxData, and clears down the RxRdy flag. This prepares the hardware for the arrival of the next item of data. The complementary version which *outputs* data is nearly identical, except the TxData flag in the status register is polled until it changes to 1, indicating that the data transmit register is empty and available. The next data item is then moved from memory into TxData, the data transmit register. At this point the polling loop starts all over again.

1.16 Hardware/software cost tradeoff

To an increasing extent, product functionality has been invested in the embedded software rather than special purpose hardware. It was immediately

appreciated, with the introduction of microprocessors in the 1970s, that the cost of duplicating and distributing software was trivial compared to manufacturing and transporting hardware units. Although this may still be true, it is apparent that hardware *production* costs are falling, and software *development* costs dramatically increasing. In addition, the lifetime maintenance cost of software has often been neglected because it was not really understood how software could deteriorate over time in a similar way to corroding metal parts. The need to fund continual software maintenance can in part be attributed not to an ageing process within the system, but rather to an evolving environment which no longer fits the software. Maybe this is paralleled in the theatre, where Shakespeare is continually reinterpreted, generation after generation, seeking to match the evolving expectation of audiences. Since 1606, the accumulated maintenance cost of *King Lear* has certainly far outstripped the original commissioning fee. In fact, software suppliers may still not fully understand the problems associated with the management and maintenance of their products; hardware revisions remain more visible and controllable. But perhaps the most problematic issue for all software products is the ease with which changes can be made, and the future need for documentation forgotten.



Software lifetime costs

1.17 Hard, soft and firm

Often the distinction is drawn between 'hard' and 'soft' real-time systems. Hard systems impose tight limits on response times, so that a delayed result is a wrong result. The examples of a jet fuel controller and a camera shutter unit illustrate the need to get a correct value computed and available at the right time. Soft real-time systems need only meet a time-average performance target. As long as most of the results are available before the deadline, the system will run successfully, with acceptably recognizable output. Audio and video transmission and processing equipment are examples of real-time systems which must achieve an average throughput performance. A single lost speech sample or image frame can normally be covered up by repeating the

previous item. Only when responses are delayed repeatedly will a seriously unacceptable error occur. The category of 'firm' is also being mooted as a crossover between the other two, because real-world systems do not always fall into either category for response deadlines.

A somewhat clearer distinction is visible between 'large' and 'small' real-time systems development. Design techniques, management methods, implementation languages and many other critical aspects are dealt with differently by groups operating at the two extremes of this application spectrum. Typical projects on the small side would be coffee or ticket vending machines, entertainment equipment, or protocol converter units. Large systems could be production plant monitoring equipment, air traffic control and telecommunication networks. Real-time systems, large and small, are becoming a routine part of our everyday life.

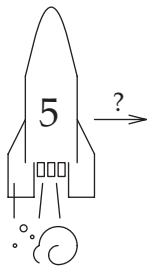
1.18 Software Quality Assurance (SQA)

The production and maintenance of high quality software has been the special concern of software engineers since the 1970s, when the term 'Software Engineering' was first coined in an attempt to express the frustration of programmers with the repeated failures of large software projects. By studying the separate activities involved in designing and realizing programs, it was hoped to improve the industry's performance. The complete lifecycle of a software product spans several distinct but overlapping phases which can, to some extent, be discussed in isolation. The software engineering approach to real-time systems emphasizes the importance of the early requirements acquisition phase and later product testing activity. As software continues to grow in size and sophistication, the need to coordinate large teams of analysts and programmers, all working on the same project, becomes more problematic. Some parallels can be drawn with traditional engineering methods, and benefits can be derived from their long experience, but this can also be misleading. The techniques which have evolved to successfully support large civil engineering projects or automobile production plants may not necessarily be appropriate for computer programmers. Remember that bridges still collapse and cars fail due to design faults, so the admiration and emulation should be cautious. Undoubtedly the complexity of software will increase still further and automated methods will have to be developed to assist the development process. In particular, real-time systems have suffered from some disasterously public failures, such as the loss of the Ariane 5 rocket during its initial

'Hardware degrades despite maintenance, software degrades because of it.'

A depressing aphorism

launch and the recall of engine management units for bug fixes, which have contributed to a general scepticism about all computer-based projects.



Costly software failures

1.19 Experience and history

Unfortunately, in computing, the lessons learned during earlier eras are often overlooked. Pioneering mainframe programmers despised the small DEC PDP-8 minicomputers when they first arrived, and the Intel 8080 microprocessors were initially ignored by everyone except hobby-mag readers and hardware engineers. In my own department, an experienced CS colleague expressed the now ludicrous view that he could see no reason to include details of the recently introduced Z80 microprocessor and CP/M operating system into university curricula. Each generation seems determined to recapitulate earlier discoveries and waste vast effort in the process. With the introduction of the PC, Microsoft and IBM spurned many well-designed, field proven operating systems in favour of DOS. This now seems an incredible leap backwards in developmental terms.

When re-reading the RTL/2 reference book written by John Barnes in 1976, I am struck by the freshness of its focus, the continuing relevance of the ideas and the apparent lack of progress achieved in dealing with the same set of software problems during the intervening three decades. The perceived need to adopt the latest jargon and intellectual style seems to have created a fashion-conscious industry which refuses to sift out and carry forward the best ideas.

Part of the problem could be that the modern computer science textbook rarely contains much technical information about past achievements in hardware and software. If there is a history section, it occurs along with the introduction, and concentrates on industry 'heroes' and archive photographs of shiny sales-room cabinets. Comments on their tiny 16 Kbit core memories do not draw out our admiration for the efficiency of the code, but rather laughter at the ludicrous idea of programs running in such confined space. Indeed, the subtle ideas and algorithms contained within them are not often

discussed or explained. History is bunk, but if we ignore it, we are condemned to repeat its mistakes and continually suffer the same frustrations.

1.20 Futures?

For real-time developers, a very relevant revolution, which may parallel that triggered by the arrival of 8 bit microprocessors, could be in progress at this very moment with the introduction of large Field Programmable Gate Arrays (FPGAs). These are configured for a particular application by writing a specification program in a language such as VHDL or Verilog. With the size and gate density achievable at present, it is possible to install several fast RISC processors on the same FPGA, and still leave space for peripheral devices. So the opportunity for 'roll your own' microcontrollers is available now, with the possibility of powerful bespoke clustering not far off. Such a development is not so revolutionary, but if the expressive potential of VHDL is pushed a bit further, it may be capable of capturing the complete application, with all its algorithms, in digital hardware without recourse to processors and software. The advantage of parallel, synchronous circuits implementing all the functionality is yet to be thoroughly investigated. Such an approach draws back together the divergent skills and traditions developed by software and hardware engineers. Those involved in real-time systems design and implementation should keep their eyes open for evolving developments from this direction.

1.21 Chapter summary

This chapter introduces the key issues which make the development of real-time software more challenging than desktop, or traditional DP applications. A set of characteristics is offered which can be used to identify those applications which may require special real-time expertise. But a clear distinction is not really valid because most modern programs have some measure of real-time features. The key significance of designing systems to handle many discrete, concurrent activities has been emphasized because of the extra complexity that this introduces. The sequencing of code at run-time in response to changing environmental circumstances is possibly the principal defining characteristic. Handling I/O activity with unusual devices can be a particular problem for real-time programmers which demands extra hardware knowledge. Hard real-time systems need to meet strict response deadlines, while soft real-time systems only have to achieve a satisfactory average performance. It is now recognized that large real-time systems require special expertise, tools and techniques for their successful development. The current revolution in the field of embedded systems centres on the application of FPGA chips as a replacement for programmable microcontrollers.

Considerations of timing must be appreciated by the system designer and programmer.

1 ms, a millisecond, one thousandth of a second	10^{-3}
1 μs, a microsecond, one millionth of a second	10^{-6}
1 ns, a nanosecond, one thousandth of a millionth of a second	10^{-9}
1 ps, a picosecond, one millionth of a millionth of a second	10^{-12}
1 fs, a femtosecond, one thousandth of a millionth of a millionth of a second	10^{-15}

1 year	32 nHz	year number rollover
6 months	64 nHz	GMT<->BST changeover
8 hr	30 μHz	AGA coal stove cycle time
10 s	0.1 Hz	photocopier page printing
1 s	1 Hz	time-of-day rate
300 ms	3 Hz	typing speed
300 ms		human reaction time
150 ms	7 Hz	mechanical switch bounce time
15 ms	70 Hz	motor car engine speed
	260 Hz	middle C
	440 Hz	concert pitch A
1 ms	1 kHz	serial line data rate
125 μs	8 kHz	digitized speech, telephone quality
64 μs	15.6 kHz	TV line rate
50 μs		Mc68000 interrupt latency
0.5 μs	2 MHz	Mc68000 instruction rate
0.074 μs	13.5 MHz	video data rate
0.050 μs		semiconductor RAM access time
0.01 μs	100 MHz	Ethernet data rate
10 ns	100 MHz	memory cycle, PC motherboard
2.5 ns	400 MHz	logic gate delay
555 ps	1.8 GHz	cellular telephone transmissions
500 ps	2 GHz	single instruction issue, Pentium IV
0.3 ps	3 THz	infrared radiation
16 fs	600 THz	visible light

$2^{10} \approx 10^3$	1000, known as 1 kilo
$2^{20} \approx 10^6$	1000_000, known as 1 mega
$2^{30} \approx 10^9$	1000_000_000, known as 1 giga
$2^{40} \approx 10^{12}$	1000_000_000_000, known as 1 tera
$2^{50} \approx 10^{15}$	1000_000_000_000_000, known as 1 peta
$2^{60} \approx 10^{18}$	1000_000_000_000_000_000, known as 1 exa

Timing parameters, from slow to fast

1.22 Problems and issues for discussion

1. What should be the intellectual basis of computer science, the CPU fetch–execute cycle or the use of abstract languages to specify functionality?

   Will the use of VHDL or Verilog to configure large FPGA chips become as significant for programmers as the traditional HLLs: C/C++ and Java?
2. With the increase in CPU speeds from 20 to 2000 MHz in 20 years (1980 to 2000), have many of the original reasons for using complex multi-tasking software been rendered irrelevant by enhanced hardware performance?
3. What aspects of code sequencing can be set at compile time, and what aspects still have to be determined at run-time? (This concerns the 'granularity' of concurrency.)
4. If every process had its own private CPU, what facilities, currently offered by operating systems, would no longer be required?
5. Look up the circumstances of the Ariane 5 launch catastrophe (4/6/96), and see whether too little or too much software engineering was principally to blame. Would the rocket have crashed if the programming had been carried out in C rather than Ada, or if the 'trusted and proven' Ariane 4 software had not been reused?
6. Compare the technical specifications for several microprocessors:

	Clock speed	*MIPS*	*Max memory*	*MMU*	*External interrupts*	*H/W timer*	*FPU*
PIC 12C508							
Intel 8051							
Motorola MCF5282							
ARM-7							
Intel Pentium-4							
Itanium 2							

1.23 Suggestions for reading

Allworth, S. & Zobel, R. (1987). *Introduction to Real-time Software Design.* Macmillan.

Barnes, J. (1976). *RTL/2, Design and Philosophy.* Hayden & Sons.

Bruyninckx, H. (2002). Real-time and Embedded Guide. From: herman.bruyninckx@mech.kuleuven.ac.be

Burns, A. & Welling, A. (2001). *Real-time Systems and Programming Languages.* Addison Wesley.

Cooling, J. E. (2003). *Software Engineering, Real-time Systems.* Addison Wesley.

Gomaa, H. (1993). *Software Design Methods for Concurrent and Real-time Systems.* Addison Wesley.

Lawrence, P. & Mauch, K. (1985). *Real-time Microcomputer Systems Design: An Introduction.* McGraw Hill.

Shaw, A. C. (2001). *Real-time Systems and Software.* Wiley.

Simon, D. (1999). *An Embedded Software Primer.* Addison Wesley.