1 Introduction to real-time systems

This is a course that will introduce various computer structures and real-time systems. The topics this course will look at are

1. describing real-time systems,
2. considering appropriate programming languages for real-time, embedded and operating systems,
3. looking at the organization of a computer,
4. describing static memory allocation,
5. describing dynamic memory allocation, specifically those appropriate for real-time systems,
6. explaining threads and tasks,
7. scheduling these tasks,
8. dealing with hardware interrupts,
9. synchronizing the execution of tasks,
10. generalizing synchronization to resource management,
11. avoiding deadlock,
12. facilitating inter-task communication,
13. creating systems that are fault tolerant,
14. describing operating systems,
15. simulating the execution of real-time systems,
16. verifying that correctness of systems,
17. dealing with file management,
18. efficient data management,
19. considering issues with virtual memory and caching,
20. digital signal processing,
21. an introduction to digital control theory,
22. security, and
23. looking at what is ahead.

We will begin with our introduction by

1. describing what a real-time system is,
2. looking at a case study of anti-lock braking systems,
3. describing the components of a real-time system, including the environment, hardware and software, and
4. reviewing a brief history of real-time systems.

We will begin by describing a real-time system.

1.1 What is a real-time system?

Most of the software you’ve used to date has been interactive: it responds to your commands. Interactive software is always subject to delays. Surely you have experienced that feeling of waiting over a second for a word processor to respond to you entering a single keystroke, or the mouse taking a split second longer to respond than would make it seamless. We will define such systems as follows:

Definition: General-purpose systems (hardware and software) are tangible and intangible components of computer systems where operations are not subject to performance constraints. There may be desirable response characteristics, but there are no hard deadlines and no detrimental consequences other than perhaps poor quality of service if the response times are unusually long.

In contrast with general-purpose systems, real-time systems are meant to monitor, interact with, control, or respond to the physical environment. The interface is through sensors, communications systems, actuators, and other input and
output devices. Under such circumstances, it is necessary to respond to incoming information in a timely manner. Delays may prove dangerous or even catastrophic. Consequently, we will define a real-time system as one where

1. the time at which a response is delivered is as important as the correctness of that response, and
2. the consequences of a late response are just as hazardous as the consequences of an incorrect response.

Those requirements that describe how the system should respond to a given set of inputs (both from sensors and messages received from communication systems) given the current state of the system and what the expected outputs (both signals to actuators and messages sent through communication systems) and changes of state of the system are described as functional requirements. Other requirements are collectively described as non-functional requirements, and these include requirements concerning safety, performance and security, as described in Table 1

<table>
<thead>
<tr>
<th>Non-functional requirement</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety</td>
<td>This deals with operational responses by the system that protect the system prevent the system from coming into harm.</td>
<td>It has been determined that an increase in engine temperature can be dealt with by reducing the throttle if the increase is detected within 5 s; consequently, if the temperature sensor is checked at least once every 2.5 s, even in the worst case, the temperature will not exceed a critical value for more than 5 s.</td>
</tr>
<tr>
<td>Performance</td>
<td>The deals with either timing of responses or throughput necessary to protect the system from harm or other non-desirable outcomes.</td>
<td>It may be required that the fire-suppression system must be activated within 10 ms of the detected light intensity of an optical beam dropping below 95 %, or it may be required that a drone must be able to accept and process ten inputs from various sensors per second including the processing of video frames.</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>The ability to protect the system from harm resulting from design faults.</td>
<td>A quadcopter drone that is able to continue flying even if one of its four engines fails would be more fault tolerant than one that fails as soon as one of the engine fails. A drone that immediately attempts to land safely in the event of an engine failure and communicate its location would be failsafe.</td>
</tr>
<tr>
<td>Robustness</td>
<td>The ability to protect the system from harm resulting from external interference and perturbations.</td>
<td>Any communication between drones or other tasks is subject to natural interference that may cause the received message to differ from the message that was originally sent. A robust system could detect and correct such introduced faults.</td>
</tr>
<tr>
<td>Scalability</td>
<td>The ability to perform reasonably in an environment with added load.</td>
<td>If suppose ten drones cooperated on a task and require 1 ms/s to communicate while performing the task. If all drones were required to communicate with all other drones, one hundred drones attempting a similar task would spend 10 ms/s communicating; meanwhile, if the drones were divided into ten groups of ten each with one drone designated as a leader, after which only the leaders communicate, communication may be reduced to as little as 2 ms/s.</td>
</tr>
<tr>
<td>Security</td>
<td>This describes the operation of the system to prevents the system from intentional harm, including harm that may cause the operation of the system to be inconsistent with the intentions of the user.</td>
<td>One hundred drones performing a search-and-identify mission of an escaped convict cannot be interfered with in such a manner as to allow the non-detection of the convict or an intentionally false identification of the location of the individual. Similarly, one hundred drones engaged in a performance at a public event cannot be redirected to cause harm to the audience.</td>
</tr>
</tbody>
</table>
Other non-functional requirements may include availability, configurability and regulatory compliance. Real-time systems are not meant to be fast, per se; instead, they should be just fast enough to ensure that all functional requirements and non-functional requirements including, but not limited to, performance requirements.

Some examples of real time systems include:

1. transportation: control systems for and traffic control of vehicles, ships, aircraft and spacecraft;
2. military: weapons system, tracking and communications;
3. industrial processes: control for production including energy, chemical and manufacturing using robotics;
4. medical: patient monitoring, defibrillation and radiation therapy;
5. telecommunications: telephone, radio, television, satellite, video telephony, digital cinema and computer networks;
6. household: monitoring and control of appliances; and
7. building management: security, heating, ventilation, air conditioning and lighting.

We will look at anti-lock braking systems as a case study of both hardware and software real-time systems. However, as time is a central component of any real-time system, we will quickly first define time and embedded systems.

1.1.1 What is time?
Time is a natural phenomenon where one “second” is

the duration of 9192631770 periods of the radiation corresponding to the transition between the two
hyperfine levels of the ground state of the caesium 133 atom at rest at a temperature of 0 K,

as defined by the Bureau international des poids et mesures. With the exception of the kilogram, all other units are
defined relative to the second. Atomic clocks are used to measure time, and coordinated universal time (UTC) is an
international standard for time. Your systems will, however, be using quartz clocks, where a quartz crystal is carved to
vibrate at $2^{15}$ Hz = 32768 Hz when an electric field is placed across it. A 5-bit digital counter will overflow once per
second as it counts the oscillations. With 86400 s/day, such clocks tend to drift less than 1 s/day and therefore different
systems will have different times even if they start synchronized (more expensive crystals will have less drift). In the
chapter on fault tolerance and robustness, we will look at techniques for synchronizing clocks between systems.

1.1.2 What are embedded systems?
Elicia White definition of an embedded system is

a computerized system that is purpose-built for its application.

The purpose-built includes both hardware and software components. Software for embedded systems is usually written
on general-purpose computers running integrated development environments (IDEs) using cross-compilers: compilers
that produce machine instructions for processors other than the processor running the IDE. An embedded system should
usually be considered an object within a larger system. The embedded system should have well defined functionality that
allows it to be replaced by another system that adheres to the same specification.

The challenges of writing applications for embedded systems include constraints such as

- cost,
- correctness (the system must be close to error free),
- main memory availability (random-access memory or RAM),
- code size restrictions (read-only memory (ROM) or flash memory),
- processor speed,
- power consumption, and
- available peripherals.
As you have seen in your study of algorithms and data structures, there is often a trade-off between speed and memory: for example, a doubly linked list requires $\Theta(n)$ more memory, but allows many $O(n)$ run-time operations in a singly linked list to now run in $\Theta(1)$ time. Similarly, trade-offs can be made between the above constraints. Other concerns with developing applications on embedded systems include

1. uncertainty as to whether issues are software or hardware,
2. the possibility of software errors causing damage to hardware, and
3. the systems tend to be remote; that is, access and maintenance (including upgrades) tend to be non-trivial issues.

None of these are concerns with software development for general-purpose processors.

### 1.2 Case study: anti-lock braking system

From physics, you may recall that static friction is stronger than dynamic friction. When trying to stop a vehicle in a very short distance or on a slippery surface, it is possible for the wheels to lock and stop rotating. When this happens, the vehicle begins to skid (dynamic friction) and loses traction. This also means the driver no longer has control over the vehicle—a dangerous situation. If the wheels do not lock up, the driver will not only have control while stopping, but the vehicle will also stop in a shorter distance.

A skilled driver can ascertain the maximum amount of brake force that can be safely applied without causing a skid. This technique is called threshold braking. It is a very difficult technique to learn and use, especially in a situation where emergency braking is needed.

Anti-lock braking systems (ABS) were first developed in the late 1920s for aircraft as a mechanism for preventing skidding during landings, as skidding will significantly reduce the lifetime of the tires and skidding in wet conditions can lead to dangerous situations. While threshold braking is possible in smaller systems such as automobiles, it is exceptionally difficult in aircraft. The entire ABS was hydraulic using a flywheel and valve that would under differential spin cause pressure to bleed from the brakes, allowing the wheels to unlock but continue to apply a braking force.

In 1958, an anti-lock brake was built for a motorcycle where it reduced stopping distances on slippery surfaces by as much as 30%. In the 1960s, such a system was built for automobiles. In both cases, the product never went into mass production.

Computerized anti-lock braking systems were introduced by Chrysler in 1971 and it was an option available for many luxury models for the next decade. It was first introduced as a standard feature in the 1985 Ford Scorpio, for which it was awarded the European Car of the Year Award in 1986.

In addition to speed sensors and hydraulic valves, modern ABS interfaces with a central electronic control unit (ECU). The ECU is an embedded system comprised of a number of computer modules that control various aspects of the car. The ECU today includes one or more microcontrollers, a clock, memory, both analog and digital inputs, and output drivers, while communication is usually through a CAN (controller area network) bus. ISO 26262 *Road vehicles—functional safety* is a standard that directs the development process of such modules.

Starting in late 2009, the National Highway Traffic Safety Administration (NHTSA) began receiving complaints concerning brake problems on the Toyota Prius that manifested itself as a short delay in regenerative braking when hitting a bump; consequently increasing the stopping distance. This was solved via a software update; however, it is not clear from the literature as to whether it was a hardware bug, or if the necessary correction could be done in software.

Note that for the microcontroller of ABS, faster is not better. A design that meets the required specified deadlines is all that is sufficient. Reliability is a much greater factor than performance. Once a design for a system such as ABS is developed, unlike desktop or mobile computer programs, there will be no need to revisit the design every year. In fact, the incentives point the other way: the system works and any change introduces the possibility of error.
1.3 Components of real-time systems

The defining characteristic of any real-time system are the timing requirements: not only must the system respond correctly to inputs, it must do so within a specified amount of time. Such requirements can generally be categorized as either

1. absolute requirements where the response must occur at defined deadlines, and
2. relative requirements where the response must occur within a specified period of time following an event.

The consequences of failing to satisfy deadlines allows one to describe real-time systems as

1. **hard real-time** where failure to meet a deadline results in a failure and any response—even if correct—following the deadline has no value,
2. **firm real-time** where failure to meet the occasional deadline will not result in a failure yet any response following a deadline has no value, but such a failure will result in a degradation of quality of service, and
3. **soft real-time** where the value of a response drops following the passing of a deadline, but the response is not wasted.

In the first two cases, if it can be determined *a priori* that the deadline will not be satisfied, it may be better to not even begin to calculate the response. More complex real-time systems will likely consist of subsystems from each of these three categories.

A real-time system is always interacting with the physical world, and a model of a real-time system, as described by Michal A. Jackson, includes the system itself, the environment and the interface. Connecting the system and the environment are input (e.g., sensors), output (e.g., actuators) and bi-directional flow of information (e.g., communication channels). These components invariably are physical in nature and thus, while providing information to the system, they are also part of the environment. This high-level approach is shown in Figure 1-1.

![Figure 1-1. A model of a real-time system.](image)

The system and interface will usually be comprised of both hardware and software; however, the last may be excluded in a purely mechanical or electrical system; however, this book will focus on those systems using a software-driven controller. Never-the-less, many of the lessons you take out of this book will have analogous applications in either pure mechanical or electro-mechanical systems. Reasons for using software to control real-time systems include:

1. the development costs are significantly lower (tools and developers are more readily available),
2. the software can be verified to be correct, and
3. maintenance can be easier as it may require only a software update.

The expense, however, is that the unit cost will be higher, as each unit will require a microcontroller and an appropriate power source. Despite this additional cost, approximately 99% of processors made today are for embedded systems, many of which are real-time systems. We will discuss these three aspects next.

### 1.3.1 The environment

The environment that the real-time system is in is beyond the control of the engineer and it must, therefore, be modelled. A real-time system can be tested in a simulated environment driven by the model and it can be validated to work under the most extreme circumstances presented by the model. If the model, however, is inaccurate, any subsequent system
may fail (as the real situation may be more demanding than the model suggested) or be excessively expensive (scenarios the system was set up to handle—costing developer time and possibly more expensive hardware—never occur). Modelling the environment is beyond the scope of this text.

1.3.2 Real-time hardware

The hardware of a software-driven real-time system first must be predictable. While this is likely obvious for any microprocessor, this also applies to sensors, actuators, other input and output devices, and communication systems.

Counter-intuitively, many of the advances in processor technology make it more difficult determine predictability: instruction pipelining, branch prediction, virtual memory and caching pose serious challenges for determining the timing behaviour of a system. These enhancements were designed to make the processor perform faster (under most circumstances), not more predictably. We will discuss some of these in a later topic.

The hardware must also be reliable and fault tolerant as well as controller driven; that is, it must be able to interact with the processor through a communication bus. Devices will require both polling and interrupt support. These concepts will also be discussed in Chapter 8 of this book.

Devices will be connected to the processor through one or more communication busses. Any shared bus will result in competitions for that resource that will degrade performance and make timing behaviour more difficult to ascertain. Furthermore, any interactions through a communications channel (wireless, Ethernet, etc.) also make for challenges in creating real-time systems (there are real-time protocols such as real-time transport protocol (RTP) as opposed to transmission control protocol (TCP), but these require additional support).

One observation is that there is no requirement for the hardware to be fast. It only needs to be fast enough as is necessary to control the expected environment in the desired manner. Consider, for example, the 8-bit Freescale RS08 microcontroller, which is a descendant of the Motorola 6800. It has only one data register: an 8-bit accumulator; it uses a 14-bit address register which allows for a maximum of $2^{14} = 16$ KiB of main memory, and the maximum processor speed is 20 MHz—200 times slower than modern general-purpose processors. The unit cost is on the order of 50 cents and less in bulk.

Hardware failures in real-time systems usually result in malfunctioning equipment, and the system may or may not be able to recover from such failures. An interesting example of a variation of a hardware failure from which a recovery was possible was in 2010, when Voyager 2, which was 13 light-hours away from Earth, experienced a communications failure. This was narrowed to a problem where “[a] value in a single memory location was changed from a 0 to a 1”. Fortunately, this could be solved with a reset of the memory; although it took over a day to determine that this solution was successful.

1.3.3 Real-time software

While there are issues that affect the predictability of hardware, the timing characteristics of hardware, never-the-less, tend to be easier to quantify. If the characteristics of a device are not adequate, it is possible to search other products. The jungle of possible software implementations of the same algorithms are, however, more varied. Therefore, the first two-thirds of this course will focus on real-time software systems: dealing with the challenges posed in devising algorithms that satisfy the timing constraints of real-time systems. A small real-time system may contain only one processor and a few hundred lines of code, while the projected estimates for the mid-1980s space station “Freedom” ran closer to 20 million lines of Ada.

---

1 Veronia McGregor of the Jet Propulsion Laboratory quoted in “NASA Finds Cause of Voyager 2 Glitch” , May 18, 2010 by Irene Klotz.
There are two configurations for real-time systems, programs where access to resources is

1. direct through machine instructions, and
2. indirect through an intermediate operating system that mediates such requests.

Whether or not there is an operating system mediating requests for resources, it is necessary to manage the resources available to programs. In this course, we will consider the management of such resources, including:

1. the processor,
2. main memory,
3. peripheral resources,
4. synchronization between tasks, and
5. file systems.

We will conclude the course by showing that the cumulative efforts we have made in managing these resources can be bundled into a single operating system kernel that executes in a protected environment which prevents executing programs from accidentally corrupting main memory or accessing other resources currently engaged in other tasks.

Numerous failures, apart from software errors (bugs), in real-time systems can be described as being the result of

1. race conditions,
2. unexpected environmental conditions, and
3. failures in the model.

The majority of this text will look at avoiding race conditions through synchronization and deadlock avoidance, but we will also look at software simulation and verification.

A race condition occurs when the response of the system (hardware or software) depends on the timing or sequencing of events or signals initiated by independent tasks, but where at least one of the responses is undesirable. These are non-deterministic bugs that are often difficult to find, as it may be very difficult to recreate the exact circumstances causing the failure; hence the alternate name, Heisenbug.

To give some examples of race conditions, suppose two individuals are driving their cars down a three-lane highway, one in the left lane and the other in the right, and each wishes to change into the middle lane. This is only a problem if both cars are in line with each other and both drivers want to make the lane change in the same five-second window. This is exasperated by factors such as lighting conditions, the alertness of the drivers, the presence of distractions, some drivers only checking the middle lane for traffic, some drivers checking first and then signalling, while others signalling first and then checking (ideally, you check, then signal and then check again), and yet others may not check, or not signal, or not do either.
Another example of a race condition is when you agree to meet someone at a building at a specific time, but when you get there, you realize that you could be meet at either the front or the back entrance. Staying at one entrance could see both of you waiting indefinitely long, but going from one entrance to the other may have both of you miss each other if you both within the same 20-second window decide to take two different paths between the two possible meeting points (after all, you could walk through the building, clockwise around the building or counter-clockwise around the building). This is less of an issue today, so long as everyone’s mobile phone is charged.

We will look at three examples of how race conditions:

1. killed patients in the Therac-25 killed,
2. almost ended the adventures of the Mars rover “Spirit” before the end of the first month, and
3. affect circuit and the benefits of circuit simplification.

We will start with Therac-25.

1.3.3.1 Therac-25

A race condition in the response of the Therac-25, a radiation therapy machine produced by Atomic Energy of Canada Limited (AECL), to operator instructions led to patients being given 100 times the expected radiation. This was the result of a race condition in which if the operator issued an instruction too soon after a previous instruction, the system was still responding to the first command and therefore ignored the second without any notification that it was doing so. Three patients died as a result.

1.3.3.2 The Mars rover “Spirit”

On January 4th, 2004, the Spirit rover set down on Mars to begin its 90-sol (Martian day or 1.027 Earth days) mission of exploring the planet surface. It would go on to communicate information back to the Earth for a total of 2210 sols, ending on March 22nd, 2010. However, a race condition due to a failure in modeling and an unexpected environmental condition may have catastrophically curtailed its mission to a mere 16 sols.

![Figure 1-3. The Martian rover “Spirit” (from NASA).](image)

The rover has a processor, 120 MiB of RAM and 256 MiB of flash memory, part of which contained files relevant to the operating system and 230 MiB of which are dedicated to a flash file system that stores data produced by the various instruments and cameras. The operating system is Vx-Works version 5.3.1 by Wind River Systems, a real-time OS that was compiled with flash file system extension. For the file system to work, however, critical information must be stored in appropriate data structures in main memory (this will be discussed in Chapter 17). Everything was fine, except for a sequence of unlikely events, which was not anticipated by the software designers.

After the rocket carrying Spirit launched on June 10th, 2003, it was determined that there were serious issues with the existing software. During the trip, new files were uploaded to the rocket carrying the rover and then installed on the rover itself. Everything seemed good to go. They even simulated Spirit in operation for 10 sols to ensure that this new installation would not cause any problems. However, the new installation added approximately a thousand extra files and directories compared to the original software.
On sol 15 (15 Martian days after landing), a utility was uploaded to Spirit to delete the obsolete files and directories, but only one of the two components was received; therefore, a second transmission was scheduled for sol 19. On sol 18, however, the rover’s scientific instruments and cameras were busy collecting data and creating data, and instructions were sent to add these new files into the flash file system. Only now, the flash memory system made a request for additional memory, but the old files and directories occupied the remaining memory, so the request for additional memory could not be fulfilled. The system did what it was designed to do if there was a failure: reset. This is more or less what most people do at home when their computer fails to respond, but in this case, the reset was automatic.

So the operating system reset, as directed. On start-up, it tries to mount the flash file system, this results in a memory request which is, again, denied. So the system resets again and again... This cycle of resets ended most communications with Earth and posed a serious problem for Spirit: it could not go to sleep at night, and therefore its system was overheating and the battery was running low. The operators on Earth even sent the command SHUTDOWN_DMT_TIL (shutdown, dammit, until—someone had a sense of humor) in hopes of putting Spirit to sleep, to no avail; unbeknownst to the operators, the reset sequence had priority, even over the shutdown command.

With no additional information, it was assumed that Spirit was in a reset cycle (there may have been other causes, for example, a solar event (solar flare or storm) had occurred just prior to Spirit’s silence, but a reset cycle was the only one that they allegedly could do anything about), and this would point to a problem in either the flash memory system, the EEPROM (Electrically Erasable Programmable Read-Only Memory), or a hardware failure. Fortunately, the software programmers included two features that allowed a recovery: a window of time was inserted between resets that allowed commands to be received, and it was possible to issue a command to boot without loading the flash file system. At this point, on sol 21, they were finally able to issue the command to give Spirit the sleep it required.

For the next two weeks, every Martian morning, a command was sent to wake up and reset without loading the flash file system. Utilities were uploaded to manipulate the flash memory directly without loading the file system. This caused some corruption, but some information was recovered, including a photograph of the Rock Abrasion Tool (RAT) (shown in Figure 1-4), and more importantly a log of every event leading up to, and including, the failed request for additional memory. Once the system was stable, an exception-handler utility was developed that would recover more gracefully from an allocation error than simply triggering a reset.

Incidentally, the Opportunity rover landed on Spirit’s sol 21—only hours after they were finally able to put Spirit to sleep. This summary is compiled from information appearing in Ron Wilson’s *The trouble with Rover is revealed* (http://www.eetimes.com/document.asp?doc_id=1148448) and Mark Adler’s blog entry and presentation *Spirit Sol 18 Anomaly* (http://hdl.handle.net/2014/40546).
1.3.3.3 Logic expression simplification

Another example of a race condition, but consider the circuit shown in Figure 1-5. From predicate logic, the result should always be equal to zero.

\[ A \land \overline{A} = 0 \]

Figure 1-5. A simple circuit with one input and output.

Unfortunately, with each circuit element, there is a slight delay as to how long it takes a change to propagate to the output. Consequently, the actual timing diagram of the voltages looks like what you see in Figure 1-6.

Thus, the output, rather than being a constant 0 V, it exhibits a spike (a window of short duration where the output is not zero). Any circuit, however, that expects a clean 0 V may react adversely to the spike if this is not accounted for. To minimize the number and impact of such transient intermediate states, Karnaugh maps are used to simplify Boolean expressions such as:

\[ A \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} \overline{D} \]

\[ = A \overline{B} + \overline{A} \overline{C} + A \overline{C} + \overline{B} \overline{D} \]

1.3.3.4 Summary of real-time software and race conditions

We’ve discussed some situations where the sequence in which events occur can result in problems. Such conditions are called race conditions. Later in the course, we will look at solutions to such problems, at least in software.

1.3.4 Summary of the components of real-time systems

Thus, a software-controlled real-time system will work in an environment of the physical world, interfaced through hardware and administered by software. This course will focus on the software component of real-time systems.
1.4 The history of real-time programming

Programming real-time systems arose in parallel with the construction of large commercial and government systems in the 1960s. In his 1965 text *Programming Real-time Computer Systems*, James Martin discusses issues such as dynamic scheduling, dynamic core allocation, allocation of priorities, multi-programming, interrupts, queues, overloads, multiprocessing, communication lines, random-access files, supervisory programs, communication with other computers, high reliability, duplexing and switchover, fall-back, programming test, problem of programmer coordination, design problems, and monitoring the programming progress. All of these issues remain associated with real-time programming today. At that time, the larger real-time systems included air defense, telephone switching, airline reservations and the space program and these often grew faster than programming paradigms could keep up. It was only in 1965 that Edsger Dijkstra proposed the concept of a semaphore, a variable used for controlling access to a shared resource (we will examine this in a later topic in great detail), to deal effectively with synchronization—synchronization and concurrency is not even a significant topic in Martin’s book.

With the introduction of semaphores (special flags) and other innovative ideas, issues such as mutual exclusion and serialization could now be dealt with in a manner that could be proved to be correct. One major step forward was with the United States government requirement of a language designed for real-time and embedded applications; the result was the programming language Ada. Furthermore, the greater availability and lower cost of processors made it desirable to shift control out of hardware and into software—not without failures—to reduce development costs. Finally, in the last two decades, real-time systems have moved into the realm of mass-produced consumer products and thereby providing significantly more investment in developing real-time systems in the commercial industries.

1.5 Topic summary

In this topic, we introduced real-time systems, we looked at a case study of the development of anti-lock braking systems, we described the relationship between the environment, hardware and software in a real-time system and looked at two situations where race conditions may lead to issues in real-time systems through race conditions.
Problem set

1.1 In one sentence, what differentiates a real-time computer system from a conventional computer system?

1.2 Recall that form your algorithms and data structures course that it is often possible to speed up an algorithm if you are willing to store more information. While this leads to often more complex functionality and increased development costs, such options are often taken in conventional computer systems. Why would you have to be more careful about such trade-offs when you are dealing with an embedded system?

1.3 There are two requirements for an anti-lock braking system (ABS):

1. the vehicle must slow down, and
2. the tires cannot skid.

Without specific numbers, what are some of the timing requirements for such a real-time system? Why does releasing the pressure on the brakes actually decrease the braking distance?

1.4 Suppose that the ABS component of a brake system fails, how should the system respond? Why?

1.5 Draw a block diagram of an ABS system.

1.6 Section 1.3.2 describes the characteristics of the Freescale RS08 microcontroller. It has only one data register—an 8-bit accumulator. All operations involve either modifying this register, writing to the register, or saving the value to a memory location. Any binary operation requires that one of the operands be located in main memory where it is fetched using direct or indirect addressing, possibly with an offset. Is this reasonable for a system where the majority of the operations involve calculating statistics based on input from a sensor, or would it be better to get a system that has two or more registers?