Chapter 1 (AK) - Real Time Basics

1.1. Introduction

From the time when in 1946 the first practical computer was unveiled, we have discovered many uses of this ingenious programmable electronic device. In addition to the original task of number crunching, computers have been used in a variety of control applications replacing conventional analog devices and adding flexibility and functionality to the conventional hard-wired digital systems. Entering the second part of the computer century, we observe an incredible and still raising impact of the computers on our society. Computers have become a significant part of our lives. Software intensive systems are all around us in any imaginable endeavor of human life. They control planes, trains, and automobiles. They support, in a transparent way, our daily routines when making a cup of coffee or making a phone call. They are a part of our buying gas, air tickets, or groceries. We do not pay too much attention to those facts since the users do not perceive the majority of the “computing intensive systems” as computers. The computer hardware and software is intrinsically embedded within a system. The computing component of the system is thus dedicated to support a particular system function. Also, the system needs to be reactive, responding to the external events, stimuli, and conditions in a time-critical fashion. As the system controls the environment, the safety, reliability and fault tolerance may be also an issue. All these represent what we define under a broad umbrella of real-time systems.

In the earlier part of the computer era the major component of the computer system development was hardware. Rapid progress of digital technology and the development of interfacing standards resulted in modularization of computing hardware and availability of off the shelf components. Such components can be interfaced together and integrated into a larger multifunctional system. This caused a shift of the effort level. The software contribution to any complex project became a major part of the total project cost and effort.

Despite significant progress on the software side, we still have not reached the expected level of reusability and software integration. Historically, the hardware developers, mostly electronic engineers, created the software on a “system by system” basis. The major characteristic of real-time systems has been always the intimate bond between hardware and software. A right mix of skills when designing and implementing real-time systems is essential. Graduates of the conventional computer science programs, focusing on programming, data structures and algorithms, need considerable on the job training to be efficient real time developers for interface-intensive systems. On the other hand electrical and computer engineering program graduates face challenges when their system became more complex requiring more sophisticated algorithms and software system construction (as opposed to “writing a program”).

An explosion of job opportunities in various aspects of computing field closely matches this incredible impact of computer on every facet of modern life. We do have hardware engineers designing, manufacturing, and integrating digital systems. We have information technology and networking specialists, operating system developers, application
programmers, and many other shades of software engineers. For the embedded, dedicated, and reactive systems it is critical to have the developers understand both hardware, and software.

There is an established body of knowledge that every software engineer must possess. Problem solving and analytical skills, proven by the completion of math and science prerequisites, are the base on which can be build knowledge of engineering discipline and process, discrete mathematics, programming, data structures, digital hardware concepts, operating systems, software development methodologies and notations. Additional topics as networking, databases, artificial intelligence, simulation, and real-time complete the minimal subset of educational requirements for a software engineer.

The specifics of real-time systems require that the software developers would have clear understanding of reactive and time-critical nature of the application. Also, for such dedicated and embedded systems, it is important to understand that the application program does not operate in a vacuum but has to interface closely with the hardware via run-time operating system kernel and various device drivers. It is also important to remember the issues of timing and exception handling. The incoming events are detected by computer hardware as interrupts, requiring the operating system kernel and application program to respond and handle them. As in real world the events originated from various sources may arrive “at the same time”, there is a need of handling concurrency of the application. This latter issue related to various synchronization and communication problems. Except some simplistic and trivial systems, most of the real time applications exhibit certain level of concurrency. This in turn requires an appropriate analysis of timing in terms of meeting the application deadlines. The real-time systems development methodology must include such issues as representation of system behavior in time and concurrency.

All these elements constitute material to be studied as an independent subject. The material requires some initial exposure to computer science/engineering basics, which may be equivalent to junior undergraduate level. In particular, the following pre-requisites would facilitate greatly comprehension of the material:

- A working knowledge of computer programming (C and/or Ada).
- Familiarity with basic computer hardware operations and operating system concepts.
- A rudimentary exposure to the software life-cycle and development process.

1.2. Background

In the early days of computing the prevalent applications were in the area of data processing. A batch of programs, often prepared in a form of a punched card deck, was delivered to the computer operator who fed it into a dedicated punch card reader. The computer read the program(s) into external magnetic tape memory. Then the computer operating system - a dedicated set of programs managing the computer operation - read one program at a time from the magnetic tape to the core memory. Subsequently, the programs were executed and the results spooled to a printer. While the results were printed the processor was idle. Possibility of placing more than one program in memory for better utilization of processor started what was defined as multiprogramming (or multitasking). A progress in the hardware design paved the way for using multiple processors and thus to the multiprocessing concept. However, the operator terminal used to be the only means of interactive communication with the computer. As underpaid computer operators wanted to have more time to read their comics strips, appropriate programs were created to control these operations. All external devices required specialized programs for control their behavior including start, stop, read, write, data access, etc. These specialized programs designed to control and manage the executing application programs and helping communication between the operator, external devices, and computer constituted the base for creation of modern computer operating systems.

Very soon it has been noticed that any external device can be effectively connected to computer thus allowing for creation of interactive programs geared toward controlling various physical devices in industrial and military applications. The progress of electronics produced a variety of interfacing devices to support data entry and displaying of results. As computers were used to control the environment with specified timing constraints, the software had to keep up with the “outside world” timing. In most of such systems the computing device was not used in its own right, but rather was embedded inside some specific functional system. The software was not modifiable and designed to work with and support user without his/her specific knowledge of computer programming and operation. Real-time embedded interactive software was born.
The following issues were critical for real-time software development to emerge as an independent area in computing disciplines:

- replacement of analog processing section of control systems by their digital equivalent,
- progress and miniaturization of computing hardware (minicomputers, microprocessors, micro-controllers),
- application of computers in military and aerospace domain,
- research in operating systems, scheduling, concurrency, exception handling, reliability, and safety,
- proliferation of computers in all areas of science, technology, and everyday life.

The advances of computer technology, with ever increasing computing power and flexibility, started the trend of computing (or software) intensive systems used in many domains. Examples of modern applications requiring software with what we call real-time characteristics include:

- process control (chemical industry, food processing)
- robotics (manufacturing, automated control)
- avionics (flight management, GPS - Global Positioning System)
- aerospace (jet engine control, fly-by-wire)
- military (weapon management, encryption, C^3 - Command/Control/Communication)
- data collection (acquisition, monitoring, signal processing)
- communication (fax machines, digital phones)
- appliances (microwave, dishwasher, thermostats)
- automotive (engine/cruise control, anti-lock brakes)
- medical (pacemaker, radiology, diagnostic)
- computer peripherals (printers, terminals, modems)

The computing support for a typical industrial application is shown in Fig. 1.1. The controller boards in an industrial enclosure interface with the plant through analog-digital (A/D) and digital-analog (D/A) converters. A separate workstation allows the operator to control and monitor the plant. It can also serve as a development environment for the software running on the controller boards.
Progress in technology and changes in the way we live makes the mankind more and more dependent on computers. Modern computer applications are controlling nuclear, military, and aerospace equipment as well as supporting many everyday devices. The typical characteristics of such devices are that:

- they have predictable and guaranteed timing behavior thus failing if the timing constraints are not met (real-time)
- they interact with and control the environment (safety critical)
- they operate continuously as a part of a larger system (embedded)

Before embedded software is designed we must consider its place and role in the entire embedded system. The developer has to understand the implication of the system environment, expected performance, and the required interfaces. In the category of environmental factors we have physical, electrical, and operational conditions contributing to the developer selection of hardware to match the system characteristics such like humidity, temperature, vibration, weight, size, power supply, electrical interference, life-span and required maintainability. In the performance category the questions are focusing on response time, throughput, reliability, predictability, and safety. For interactive systems controlling environment the issue of handling consequences of system faults and failures, and the related issues of safety, are of paramount importance. We need to stress that the fault is here defined as a defect within the system, while the failure is a situation when the system fails to perform its required function.

For the interfacing we need to consider the type of signals, their range, frequency, communication protocols, connector standards. The actual devices to be used include various sensors, switches, actuators, motors, display panels, converters, communication links, converters, controllers, etc. In this category the developer must also consider possible exception handling, memory and interrupt interfaces, and system malfunctions. All these consideration must find its way into the software requirements and eventually into real-time software implementation.

### 1.3. Nature of Real Time System

Any entity with a set of inputs and outputs (and the internal finite number of states) meeting specified behavioral requirements is defined as a system. Systems range from very simple controller measuring temperature and switching the heating on/off to national air traffic control with hundreds of terminals, radar stations, and thousands of airborne avionics devices. Of course, the complexity of the system often prevents us to get a full understanding of its operation. In such case we use the concept of decomposition to identify subsystems. Further decomposition leads to a hierarchy of subsystems, with the subsystems on the lower level of significantly lower complexity.

A system has its environment defined at the system boundary identifying everything "outside" the system scope. The environmental happenings are to be detected by the system and the system must respond to them accordingly. As we speak about the software systems, i.e. computer programs, any such occurrence must eventually cause an event forcing the program to branch. Synchronous events occur at predictable time in flow of control. In most cases they are internal to the system, often initiated within the system software. Asynchronous events occur at unpredictable points in the flow of control and are usually triggered externally.

Any real time system deals with several issues that need to be considered. Most systems require manipulation with real numbers as the data represent various analog entities. Of course, the system reliability and safety are of primary importance. It is evident that the real-time system shall not hang nor crash. Implementation of concurrency is one of the critical issues. Real time facilities such like clocks and timers are supporting timeliness of application. Interrupt controllers and various non-standard input/output devices are often encountered in real time systems to support interactive nature of real-time systems. And even though the speed of execution is not a distinctive feature of real-time system, the predictability and related efficiency of computation is.

In real time system the inputs from the outside world may arrive at an unpredictable time points. An enemy radar lock, a chemical component reaching certain level, transferring to the cruising speed, or overheating of engine may happen asynchronously. If such condition occurs, the program must be able to interrupt its current activity and then execute some pre-defined code to respond to the input. The accepted data can, in turn, activate some other programs that were waiting for that input. The computer is able to resume its previous activity after appropriate serving the asynchronous event. All this should be done in a timely fashion.
As the system reacts to the environmental events and interacting with external entities it is natural to have software constructed as a set of multiple tasks responsible for various events, entities, and functions. Both external and internal events require timely response.

There are two ways a task can be activated. One situation requires task to execute in a specific time interval. For example, the temperature reading, and an appropriate action related to the reading, must be done every five seconds. A task activated in specified time intervals, driven by some timing mechanism, is a periodic task. Another situation is when the task is activated by some external or internal event. Such activation is sporadic, driven by events. We see that tasks may be activated periodically or sporadically. Completion of each task before the deadline is another important feature of real-time system. For periodic task it is evident that all processing must be completed before the next activation, most often equivalent to the time period. For sporadic task it is required that the task shall be processed as soon as possible. In such situation the deadline is arbitrary depending on the task level of importance and urgency. We may have two possible situations: some events require minimal and bounded response time while for other events important is that in average the response be kept in a reasonable range.

A real time program is usually organized as a set of separate tasks executed either sequentially or concurrently. The term used here is multitasking. We may have relatively high degree of interdependence between tasks, which require appropriate synchronization and communication. Synchronization is a mechanism allowing the tasks to reach defined processing point at the same time. It is accomplished in such way that one task reaching such point waits for another task notification. If the data are to be exchanged, the process is called communication. The tasks need to provide timely response to events.

The system may use more than one processor (CPU) connected via bus. The tasks are distributed across the processors. How the tasks are distributed depends obviously on the hardware architecture, the operating system, and the way the software developer designed the application task distribution. This situation is defined as multiprocessing. Another version of the same concept is when the application is divided on different pieces with each running on different processor and the results are aggregated for the final solution. Such systems are referred as parallel systems. Solution of partial differential equations, complex computations, weather predictions, and finite element methods are the prime examples of applications appropriate for parallel systems.

---

**Figure 1.2**
The timing relationship is a matter of task scheduling. A popular categorization distinguishes between hard and soft real-time applications:

- **Hard real-time**: specify precisely the time at which each task must finish, estimate execution time needed, no missed deadline can be tolerated, system fails when the deadline is missed.
- **Soft real-time**: specify only the level of urgency of the task; occasionally missed deadline may be tolerated, the system performance may degrade when the deadline is missed and we use the “late” results.

Fig. 1.2. shows the distinction between the categories. It is based on the utility of timely response. For a **hard real-time** system, response produced after the deadline can be harmful and thus it is presented with a “negative” utility value. For a **soft real-time**, late response is of lower utility, but still can be accepted. Examples of hard real-time are various process control, military, aerospace, and medical applications. Soft real-time occurs in consumer electronics, data acquisition, interactive data processing, etc. Some authors define intermediate **firm real-time** application, which does specify hard deadlines, but occasionally a missed deadline can be tolerated. The system discards the “late” results but does not fail.

### 1.4. Determinism and Schedulability

The critical issue in real-time systems is **determinism**. We must be certain that for a given mode of operation and set of inputs, the system responds in a precisely defined and repeatable way. We define a **deterministic system** as a system in which for each state and set of inputs a unique set of outputs can be determined. Very often, the issue is not to know what is the value of the output but whether or not the time of the generating of the response is bounded. When the response time is known, or bounded, for each set of outputs we call the system **temporally deterministic**. Related with this is concept of **predictability**. Predictability is the property of meeting the temporal determinism criteria. Real-time systems must be predictable.

The computer is nothing else but a **state machine**. Any dynamic system may be described as a sequence of states (or **modes of operation**). The system state changes in response to an input. A typical characteristic of systems reacting to and interfaced with the outside world is that they need to respond to asynchronous **events** (often defined as an **interrupt** or **signal**). As a side-effect of the state change, an output may be produced. Such output often is characterized as the system **action**. Any change of state requires time. We may thus define a real-time system as a computer system that must satisfy explicit **response time constraints**. It can be said that the system correctness depends not only on the correctness of produced output data but also on the time at which the data are produced. The system fails if the **timing constraints** are not met. Guarantee of timing behavior makes the system **predictable**. Another characteristic of a real-time system is that it often exhibits high degree of **utilization** while satisfying the timing constraints. It is obvious that a system that is used for only minuscule fraction of the time, waiting idle for most of the time, is not that challenging. Such notion is of particular weight for soft real-time systems focusing on average response and throughput. It needs to be noted that the predictability and meeting timing constraints are main features of hard real-time systems.

**Response time** is the time interval between the presentation of inputs (stimuli) and the appearance of associated outputs (response). Considering the scenario above, the response time is the time between the detection of the interrupt generated by external happening (temperature exceeded the limit) and completion of a routine generating an output responding to this external happening (switching off the heater). A part of this time constitutes **interrupt latency** i.e. the time required to recognize and start responding to an interrupt.

**Deadline** may be defined as a time instant before which certain event must occur. In real-time software, we often have a programming thread responding to external stimuli. The response must be completed before the next occurrence of the stimuli. The example of such situation is when the program performs certain computations activated by a clock every 20 milliseconds. If the computation exceeds 20 milliseconds, it is obvious that the next activation will be delayed and the deadline will be missed. If the software is composed of several tasks, all of them must meet their respective deadlines. A property of a set of tasks ensuring that all tasks shall meet their respective deadlines is called **schedulability**.

The computer often is in an idle state waiting for an event to activate a routine responding to this event. The ratio of the time spent on routines responding to events and executing "useful" actions to the total time is a measure of the
system utilization. Often this term is called time-loading. Time loading in the range of 80-90% is a rule of thumb measure of properly designed and utilized system (still with a potential extension capability). Systems of lower time loading are underutilized and probably could be designed cheaper. System of very high utilization may be overloaded in some extreme situations.

1.5. Development Environments

The real-time programs range from couple of hundreds of lines of code for a simple micro-controller application, to millions of lines in complex air traffic control programs on a distributed network of computers. At heart of any computer system is a microprocessor - a highly sophisticated general purpose integrated circuit combining functionality of arithmetic/logic unit with data control.

The microprocessor, a central element of modern computing devices, responds to the external events triggered by change of voltage on one or many of its dedicated inputs. Such response is processor architecture specific and most modern processors are going through the following interrupt service sequence. The processor, after detecting the interrupt, completes the current instruction, places the current processor state and the return address of executing thread on stack, and starts executing the Interrupts Service Routine (ISR) associated with this specific interrupt. Only after the ISR is completed the "interrupted" thread resumes execution from the next instruction. All this happens entirely in the hardware context and it is below the level of the operating system and scheduling. The interrupt can be generated externally or internally. The operating system layer isolates the application from hardware responding to interrupt by generating a specific signal that may initiate an event.

There are various hardware platform combinations. In the automotive industry we have examples of using simple micro-controllers communicating with each other on a special bus (CAN). They control operation of various components of an automotive system, collect the data, and monitor the performance of the system. The functions of simple controllers can be aggregated in one more powerful processor installed on a single board with appropriate memory and peripheral devices. When the system is more complex it may require dedicated operations like data collection, digital signal processing, or graphic display. In such case we may use a system with multiple boards in a standard enclosure connected with a standard bus. Most of the avionics and military systems exhibit such architecture. Then there are systems where a conventional microcomputer, connected to the environmental sensors and actuators, is used for dedicated applications. Point of sales terminals and various industrial control stations are examples of such solution. On a factory floor, where the information and control is distributed across a wide area, the architectural solution is to use a distributed network of such workstations. For complex computations a single computer is not enough. A network of supercomputers performs computations in a coordinated fashion in selected military and aerospace applications.

We may thus identify the a variety of architectural hardware platforms for a real time implementations:

- a simple micro-controller – a simple integrated device with dedicated application
- a single controller board – a simple system with dedicated application
- a complex controller boards on a bus – a complex multi-function system
- a dedicated workstation – a simple to complex system with operator interaction
- a distributed workstations – a complex multi-user system
- a network of interconnected supercomputers – a high performance system for special applications

The first three categories of platforms are stand-alone systems, which are operated by users without in-depth knowledge of software development. The system does not have appropriate user interface nor development environment to create or update their software. Such systems are treated as targets requiring a separate host environment to develop the code and download its executable to the target for execution. Such host-target development is typical for most real-time applications. The host environment may be workstations of completely different configurations and using different processor architecture than the target computers.

The workstation real-time software is typically developed on the same system where the resulting programs are running. The workstation is powerful enough to house the entire development environment with CASE tools, compilers, debuggers, loaders, etc. The verified and validated programs are loaded on the target workstation for

DRAFT – Chapter 1
Kornecki/Zalewski
operational use. This type of development is called native, as the architectural platforms of both host and target are usually the same.

1.6. Programming Languages

Selection of an application programming language is always an emotional issue. In a world of real-time embedded applications the specific languages established their entrenched positions. We have a number of dated systems using assembly languages occasionally combined with FORTRAN. Significant number of military applications has been implemented in Ada. This approach gaining support in late 80’s and early 90’ seem to be losing its momentum due to the proliferation of C/C++ based systems. The emerging consumer electronics applications and proliferation of Internet savvy devices makes Java yet another strong player on the implementation language field.

A consideration of whether or not a specific programming language is appropriate for real-time software implementation include the following characteristics:

- safety – whether or not the language construct are safe enough to limit potential implementation errors
- efficiency – whether or not the language produces the executable code of a small footprint of and fast execution time
- readability – whether or not the language syntax in terms of control structures and operators is clear enough to make the application easy to understand and maintain
- flexibility – whether or not the language contains, and allows for creation, all the required operators, data structures, and constructs to support real-time system concepts dealing with concurrency, timing, interfacing, synchronization
- portability – whether or not the wide variety of compilers, debuggers and other tools exists that the application can be easily ported to another hardware platform
- simplicity – whether or not the language is simple enough to allow for easy development thus supporting the development cost reduction

It is beyond any discussion that Ada, being specifically designed for real-time, is best suited for this purpose. It is widely known that C/C++ in the full version is only a marginal choice for such applications. It is also known that while use of assembly is often the only applicable option, it may be a source of serious problems. The truth is, however, that the real-time systems have been and will be implemented using various programming languages and platforms. The assembly and C/C++, not mentioning FORTRAN and Pascal, have a significant ”share of the market”. As an example, the languages used to develop software supporting modern military aircraft avionics have an equal share of about 30% of total code - written in Ada, C/C++ and Jovial/assembly (with the remaining 10% spread among 30+ other languages). The selection of language is often the matter of past project history, available infrastructure, proficiency of the available developers, company policy, and various non-functional system requirements. The critical issue for the developer is to understand the real time concepts and identify the potential difficulties in order not to fell into trap of faulty implementation violating the system requirements. The selection of ”right” language can better support real time development but does not replace good software engineering and knowledge of real time concepts.

Rather than discussing a wide variety of languages, we present here a simplified categorization focused on the way the language supports the target system. The first category is assembly language. It is platform specific and requires development on the low level with detailed knowledge of hardware. The major platforms for the assembly development are processors of Motorola and Intel. In military applications the DOD1750 processor has been in wide use. To the second category of languages we include all conventional high-level sequential languages requiring interpreters or compilers to translate them into platform specific machine code. They rely on the operating system support (and often assembly code inserts) to accomplish the required concurrency, multitasking and input/output operations. The examples of such include BASIC, Pascal, FORTRAN, Jovial, Coral66, RTL/2, C, or C++. The third category includes concurrent high-level languages supporting the concurrency on the language level. The examples of such are Ada, Java, Concurrent C, Modula2, Mesa, CHILL, Euclid or PEARL.

We should not underestimate the importance of defining and following reasonable coding standards. The appropriate naming conventions, limitation of module length, definition of all variables and clear interfaces between the modules, proper comments and version control are some of the characteristics of a successful software project.
Another important consideration is the operating system standard. A widely accepted industry standard has been introduced to support the portability of applications. POSIX stands for Portable Operating System Interface (with an X added to make a cool sound). It provides source-code portability for applications running on different platforms. POSIX was created in response to the government needs (document FIPS-151) by IEEE working groups and accepted as a standard by ANSI and ISO. The size of the standard and the number of its versions does create considerable confusion. It started with POSIX.1 in 1990, which specified basic operating system calls. The new version, POSIX.4 (or 1003.1b), adds real-time (shared memory, message queues, priority scheduling, synchronized I/O). The standard address issues of: concurrency, timing, deadlines, predictability, dynamic resource allocation, synchronization, portability, performance, system evaluation, etc.

1.7. Summary

We have discussed a variety of issues related to what is understood under term “real time software”. From the historical perspective there has been always need of monitoring and controlling the environment. The control systems since the times of Watt’s regulator have been doing this before computers entered the scene. The flexibility of computers, however, allows us to use them in nearly every facet of the technology and everyday life. More and more of the systems became computing-intensive, with the software reacting to external events in a timely fashion. It is essential for the software to be developed in such way that the resulting system is reliable, predictable and safe. For all practical purpose, most of the modern software intensive systems may be characterized as a real time system. This is why the presented concepts are so essential to be mastered by software developers.