Chapter 1
Introduction

1.1 Embedded Systems: History and Overview

An *embedded system* can be broadly defined as a device that contains tightly coupled hardware and software components to perform a single function, forms part of a larger system, is not intended to be independently programmable by the user, and is expected to work with minimal or no human interaction. Two additional characteristics are very common in embedded systems: reactive operation and heavily constrained.

Most embedded system interact directly with processes or the environment, making decisions on the fly, based on their inputs. This makes necessary that the system must be reactive, responding in real-time to process inputs to ensure proper operation. Besides, these systems operate in constrained environments where memory, computing power, and power supply are limited. Moreover, production requirements, in most cases due to volume, place high cost constraints on designs.

This is a broad definition that highlights the large range of systems that fall into it. In the next sections we provide a historic perspective in the development of embedded systems to bring meaning to the definition above.

1.1.1 Early Forms of Embedded Systems

The concept of an embedded system is as old as the concept of a an electronic computer, and in a certain way, it can be said to precede the concept of a general purpose computer. If we look a the earliest forms of computing devices, they adhere better to the definition of an embedded system than to that of a general purpose computer. Take as an example early electronic computing devices such as the Colossus Mark I and II computers, partially seen in Fig. 1.1. These electro-mechanical behemoths, designed by the British to break encrypted teleprinter German messages during World War II, were in a certain way similar to what we define as an embedded system.

M. Jiménez et al., *Introduction to Embedded Systems*,
DOI: 10.1007/978-1-4614-3143-5_1,

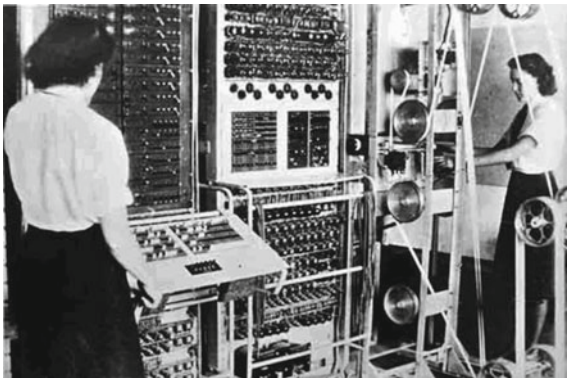

Fig. 1.1 Control panel and paper tape transport view of a Colossus Mark II computer (public image by the British Public Record Office, London)

Although not based on the concept of a stored-program computer, these machines were able to perform a single function, reprogrammability was very awkward, and once fed with the appropriate inputs, they required minimal human intervention to complete their job. Despite their conceptual similarity, these early marvels of computing can hardly be considered as integrative parts of larger system, being therefore a long shot to the forms known today as embedded systems.

One of the earliest electronic computing devices credited with the term "embedded system" and closer to our present conception of such was the Apollo Guidance Computer (AGC). Developed at the MIT Instrumentation Laboratory by a group of designers led by Charles Stark Draper in the early 1960s, the AGC was part of the guidance and navigation system used by NASA in the Apollo program for various spaceships. In its early days it was considered one of the riskiest items in the Apollo program due to the usage of the then newly developed monolithic integrated circuits.

The AGC incorporated a user interface module based on keys, lamps, and seven-segment numeric displays (see Fig. 1.2); a hardwired control unit based on 4,100 single three-input RTL NOR gates, 4 KB of magnetic core RAM, and 32 KB of core rope ROM. The unit CPU was run by a 2.048 MHz primary clock, had four 16-bit central registers and executed eleven instructions. It supported five vectored interrupt sources, including a 20-register timer-counter, a real-time clock module, and even allowed for a low-power standby mode that reduced in over 85 % the module's power consumption, while keeping alive all critical components.

The system software of the Apollo Guidance Computer was written in AGC assembly language and supported a non-preemptive real-time operating system that could simultaneously run up to eight prioritized jobs. The AGC was indeed an advanced system for its time. As we enter into the study of contemporary applications, we will find that most of these features are found in many of today's embedded systems.

Despite the AGC being developed in a low scale integration technology and assembled in wire-wrap and epoxy, it proved to be a very reliable and dependable design. However, it was an expensive, bulky system that remained used only for highly

Fig. 1.2 AGC user interface module (public photo EC96-43408-1 by NASA)

specialized applications. For this reason, among others, the flourishing of embedded systems in commercial applications had to wait until another remarkable event in electronics: the advent of the microprocessor.

1.1.2 Birth and Evolution of Modern Embedded Systems

The beginning of the decade of 1970 witnessed the development of the first microprocessor designs. By the end of 1971, almost simultaneously and independently, design teams working for Texas Instruments, Intel, and the US Navy had developed implementations of the first microprocessors.

Gary Boone from Texas Instruments was awarded in 1973 the patent of the *first single-chip microprocessor architecture* for its 1971 design of the TMS1000 (Fig. 1.3). This chip was a 4-bit CPU that incorporated in the same die 1K of ROM and 256 bits of RAM to offer a complete computer functionality in a single-chip, making it the first microcomputer-on-a-chip (a.k.a microcontroller). The TMS1000 was launched in September 1971 as a calculator chip with part number TMS1802NC.

The i4004, (Fig. 1.4) recognized as the *first commercial, stand-alone single chip microprocessor,* was launched by Intel in November 1971. The chip was developed by a design team led by Federico Faggin at Intel. This design was also a 4-bit CPU intended for use in electronic calculators. The 4004 was able to address 4K of memory, operating at a maximum clock frequency of 740 KHz. Integrating a minimum system around the 4004 required at least three additional chips: a 4001 ROM, a 4002 RAM, and a 4003 I/O interface.

The third pioneering microprocessor design of that age was a less known project for the US Navy named the Central Air Data Computer (CADC). This system implemented a chipset CPU for the F-14 Tomcat fighter named the MP944. The system

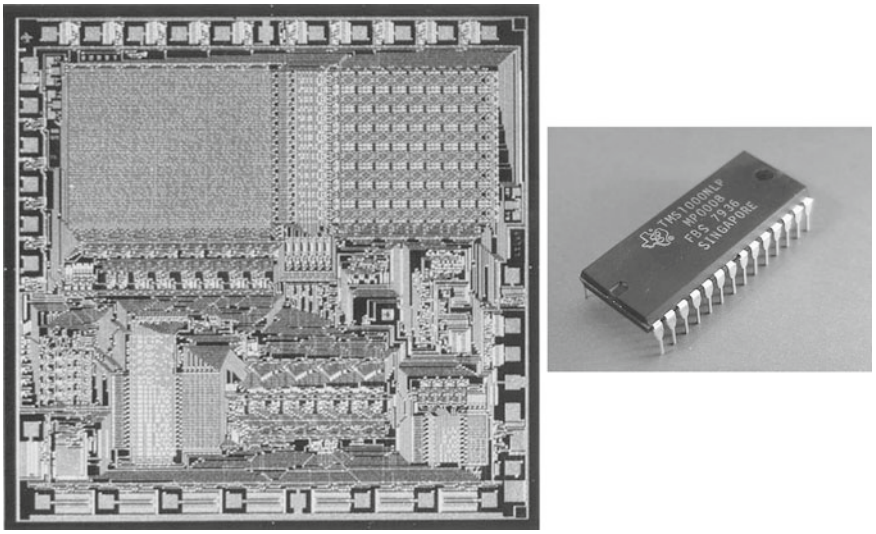

Fig. 1.3 Die microphotograph (*left*) packaged part for the TMS1000 (*Courtesy of Texas Instruments, Inc.*)

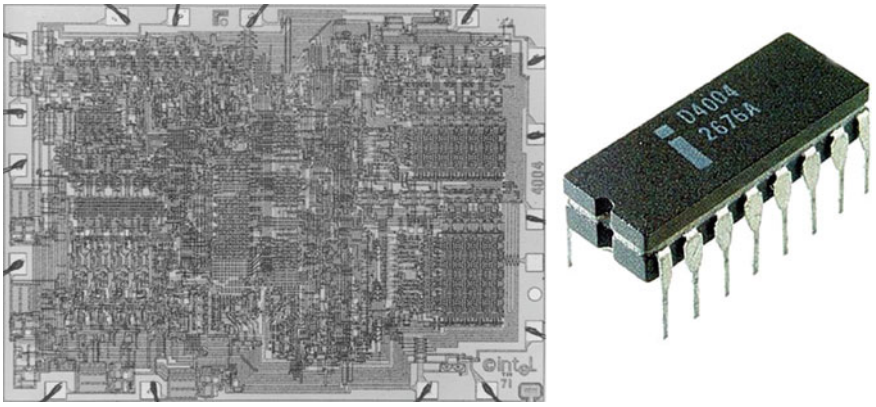

Fig. 1.4 Die microphotograph (*left*) and packaged part for the Intel 4004 (*Courtesy of Intel Corporation*)

supported 20-bit operands in a pipelined, parallel multiprocessor architecture designed around 28 chips. Due to the classified nature of this design, public disclosure of its existence was delayed until 1998, although the disclosed documentation indicates it was completed by 1970.

After these developments, it did not take long for designers to realize the potential of microprocessors and its advantages for implementing embedded applications. Microprocessor designs soon evolved from 4-bit to 8-bit CPUs. By the end of the 1970s, the design arena was dominated by 8-bit CPUs and the market for

microprocessors-based embedded applications had grown to hundreds of millions of dollars. The list of initial players grew to more than a dozen of chip manufacturers that, besides Texas Instruments and Intel, included Motorola, Zilog, Intersil, National Instruments, MOS Technology, and Signetics, to mention just a few of the most renowned. Remarkable parts include the Intel 8080 that eventually evolved into the famous 80 × 86/Pentium series, the Zilog Z-80, Motorola 6800 and MOS 6502. The evolution in CPU sizes continued through the 1980s and 1990s to 16-, 32-, and 64-bit designs, and now-a-days even some specialized CPUs crunching data at 128-bit widths. In terms of manufacturers and availability of processors, the list has grown to the point that it is possible to find over several dozens of different choices for processor sizes 32-bit and above, and hundreds of 16- and 8-bit processors. Examples of manufacturers available today include Texas Instruments, Intel, Microchip, Freescale (formerly Motorola), Zilog, Advanced Micro Devices, MIPS Technologies, ARM Limited, and the list goes on and on.

Despite this flourishing in CPU sizes and manufacturers, the applications for embedded systems have been dominated for decades by 8- and 16-bit microprocessors. Figure 1.5 shows a representative estimate of the global market for processors including microprocessors, microcontrollers, DSPs, and peripheral programmable in recent years. Global yearly sales approach $50 billion for a volume of shipped units of nearly 6 billion processor chips. From these, around three billion units are 8-bit processors. It is estimated that only around 2 % of all produced chips (mainly in the category of 32- and 64-bit CPUS) end-up as the CPUs of personal computers (PCs). The rest are used as embedded processors. In terms of sales volume, the story is different. Wide width CPUs are the most expensive computer chips, taking up nearly two thirds of the pie.

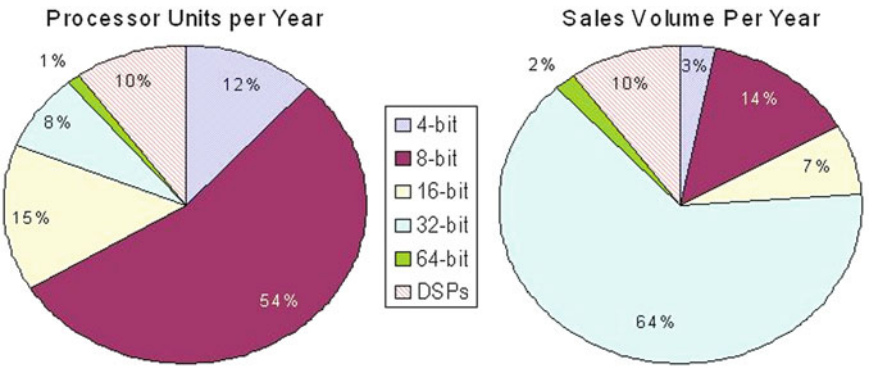

Fig. 1.5 Estimates of processor market distribution (*Source* Embedded systems design—www.embedded.com)

1.1.3 Contemporary Embedded Systems

Nowadays microprocessor applications have grown in complexity; requiring applications to be broken into several interacting embedded systems. To better illustrate the case, consider the application illustrated in Fig. 1.6, corresponding to a generic multimedia player. The system provides audio input/output capabilities, a digital camera, a video processing system, a hard-drive, a user interface (keys, a touch screen, and graphic display), power management and digital communication components. Each of these features are typically supported by individual embedded systems integrated in the application. Thus, the audio subsystem, the user interface, the storage system, the digital camera front-end, and the media processor and its peripherals are among the systems embedded in this application. Although each of these subsystems may have their own processors, programs, and peripherals, each one has a specific, unique function. None of them is user programmable, all of them are embedded within the application, and their operation require minimal or no human interaction.

The above illustrated the concept of an embedded system with a very specific application. Yet, such type of systems can be found in virtually every aspect of our daily lives: electronic toys; cellular phones; MP3 players, PDAs; digital cameras; household devices such as microwaves, dishwasher machines, TVs, and toasters; transportation vehicles such as cars, boats, trains, and airplanes; life support and medical systems such as pace makers, ventilators, and X-ray machines; safety-critical

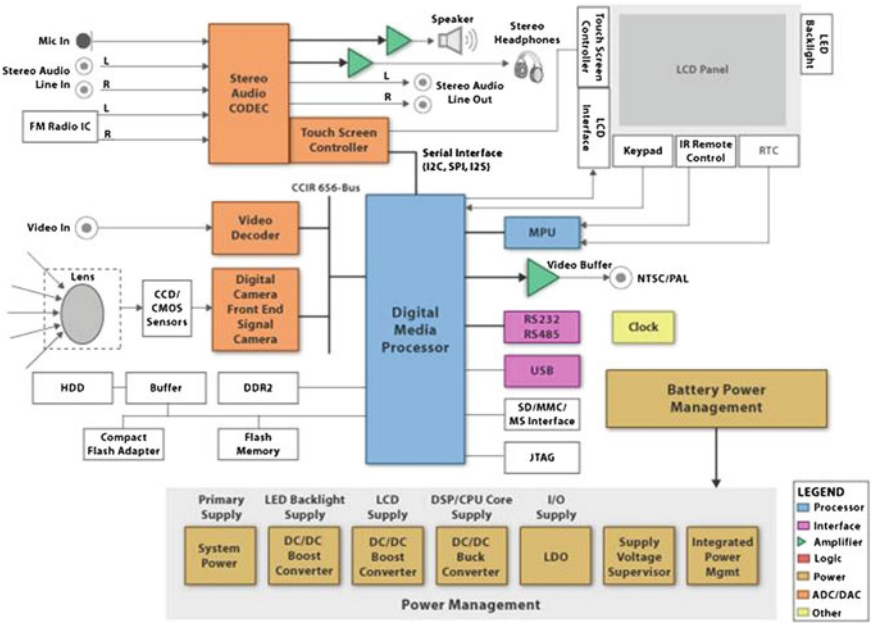

Fig. 1.6 Generic multi-function media player (*Courtesy of Texas Instruments, Inc.*)

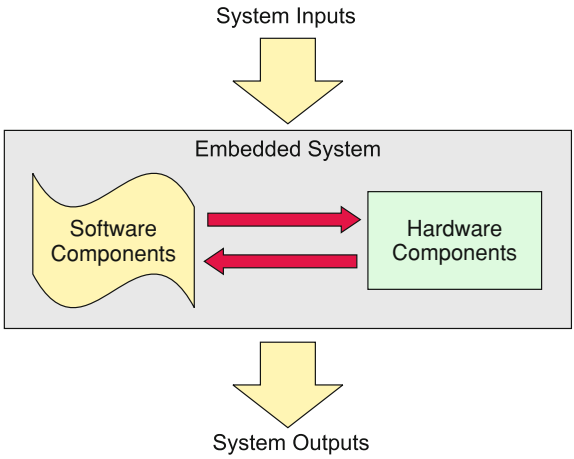

Fig. 1.7 General view of an embedded system

systems such as anti-lock brakes, airbag deployment systems, and electronic surveillance; and defense systems such as missile guidance computers, radars, and global positioning systems are only a few examples of the long list of applications that depend on embedded systems. Despite being omnipresent in virtually every aspect of our modern lives, embedded systems are ubiquitous devices almost invisible to the user, working in a pervasive way to make possible the "intelligent" operation of machines and appliances around us.

1.2 Structure of an Embedded System

Regardless of the function performed by an embedded system, the broadest view of its structure reveals two major, tightly coupled sets of components: a set of hardware components that include a central processing unit, typically in the form of a microcontroller; and a series of software programs, typically included as firmware,[1] that give functionality to the hardware. Figure 1.7 depicts this general view, denoting these two major components and their interrelation. Typical inputs in an embedded system are process variables and parameters that arrive via sensors and input/output (I/O) ports. The outputs are in the form of control actions on system actuators or processed information for users or other subsystems within the application. In some instances, the exchange of input-output information occurs with users via some sort of user interface that might include keys and buttons, sensors, light emitting diodes (LEDs), liquid crystal displays (LCDs), and other types of display devices, depending on the application.

[1] Firmware is a computer program typically stored in a non-volatile memory embedded in a hardware device. It is tightly coupled to the hardware where it resides and although it can be upgradeable in some applications, it is not intended to be changed by users.

Consider for example an embedded system application in the form of a microwave oven. The hardware elements of this application include the magnetron (microwave generator), the power electronics module that controls the magnetron power level, the motor spinning the plate, the keypad with numbers and meal settings, the system display showing timing and status information, some sort of buzzer for audible signals, and at the heart of the oven the embedded electronic controller that coordinates the whole system operation. System inputs include the meal selections, cooking time, and power levels from a human operator through a keypad; magnetron status information, meal temperature, and internal system status signals from several sensors and switches. Outputs take the form of remaining cooking time and oven status through a display, power intensity levels to the magnetron control system, commands to turn on and off the rotary plate, and signals to the buzzer to generate the different audio signals given by the oven.

The software is the most abstract part of the system and as essential as the hardware itself. It includes the programs that dictate the sequence in which the hardware components operate. When someone decides to prepare a pre-programmed meal in a microwave oven, software picks the keystrokes in the oven control panel, identifies the user selection, decides the power level and cooking time, initiates and terminates the microwave irradiation on the chamber, the plate rotation, and the audible signal letting the user know that the meal is ready. While the meal is cooking, software monitors the meal temperature and adjusts power and cooking time, while also verifying the correct operation of the internal oven components. In the case of detecting a system malfunction the program aborts the oven operation to prevent catastrophic consequences. Despite our choice of describing this example from a system-level perspective, the tight relation between application, hardware, and software becomes evident. In the sections below we take a closer view into the hardware and software components that integrate an embedded system.

1.2.1 Hardware Components

When viewed from a general perspective, the hardware components of an embedded system include all the electronics necessary for the system to perform the function it was designed for. Therefore, the specific structure of a particular system could substantially differ from another, based on the application itself. Despite these dissimilarities, three core hardware components are essential in an embedded system: The Central Processing Unit (CPU), the system memory, and a set of input-output ports. The CPU executes software instructions to process the system inputs and to make the decisions that guide the system operation. Memory stores programs and data necessary for system operation. Most systems differentiate between program and data memories. Program memory stores the software programs executed by the CPU. Data memory stores the data processed by the system. The I/O ports allows conveying signals between the CPU and the world external to it. Beyond this point,

a number of other supporting and I/O devices needed for system functionality might be present, depending on the application. These include:

- Communication ports for serial and/or parallel information exchanges with other devices or systems. USB ports, printer ports, wireless RF and infrared ports, are some representative examples of I/O communication devices.
- User interfaces to interact with humans. Keypads, switches, buzzers and audio, lights, numeric, alphanumeric, and graphic displays, are examples of I/O user interfaces.
- Sensors and electromechanical actuators to interact with the environment external to the system. Sensors provide inputs related to physical parameters such as temperature, pressure, displacement, acceleration, rotation, etc. Motor speed controllers, stepper motor controllers, relays, and power drivers are some examples of actuators to receive outputs from the system I/O ports. These are just a few of the many devices that allow interaction with processes and the environment.
- Data converters (Analog-to-digital (ADC) and/or Digital-to-Analog (DAC)) to allow interaction with analog sensors and actuators. When the signal coming out from a sensor interface is analog, an ADC converts it to the digital format understood by the CPU. Similarly, when the CPU needs to command an analog actuator, a DAC is required to change the signal format.
- Diagnostics and redundant components to verify and provide for robust, reliable system operation.
- System support components to provide essential services that allow the system to operate. Essential support devices include power supply and management components, and clock frequency generators. Other optional support components include timers, interrupt management logic, DMA controllers, etc.
- Other sub-systems to enable functionality, that might include Application Specific Integrated Circuits (ASIC), Field Programmable Gate Arrays (FPGA), and other dedicated units, according to the complexity of the application.

Figure 1.8 illustrates how these hardware components are integrated to provide the desired system functionality.

1.2.2 Software Components

The software components of an embedded system include all the programs necessary to give functionality to the system hardware. These programs, frequently referred to as the system *firmware*, are stored in some sort of non volatile memory. Firmware is not meant to be modifiable by users, although some systems could provide means of performing upgrades. System programs are organized around some form of operating system and application routines. The operating systems can be simple and informal in small applications, but as the application complexity grows, the operating system requires more structure and formality. In some of these cases, designs are developed

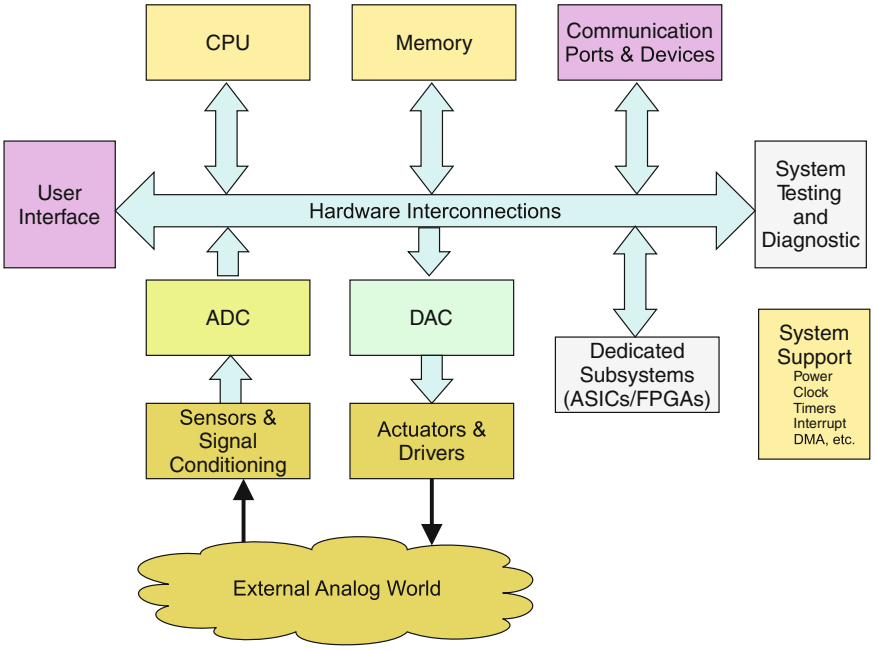

Fig. 1.8 Hardware elements in an embedded system

around *Real-Time Operating Systems* (RTOS). Figure 1.9 illustrates the structure on an embedded system software.

The major components identified in a system software include:

System Tasks. The application software in embedded systems is divided into a set of smaller programs called *Tasks*. Each task handles a distinct action in the system and requires the use of specific *System Resources*. Tasks submit service requests to the *kernel* in order to perform their designated actions. In our microwave oven example the system operation can be decomposed into a set of tasks that include reading the keypad to determine user selections, presenting information on the oven display, turning on the magnetron at a certain power level for a certain amount of time, just to mention a few. Service requests can be placed via registers or interrupts.

System Kernel. The software component that handles the system resources in an embedded application is called the *Kernel*. System resources are all those components needed to serve tasks. These include memory, I/O devices, the CPU itself, and other hardware components. The kernel receives service requests from tasks, and schedules them according to the priorities dictated by the *task manager*. When multiple tasks contend for a common resource, a portion of the kernel establishes the resource management policy of the system. It is not uncommon finding tasks that need to exchange information among them. The kernel provides a framework

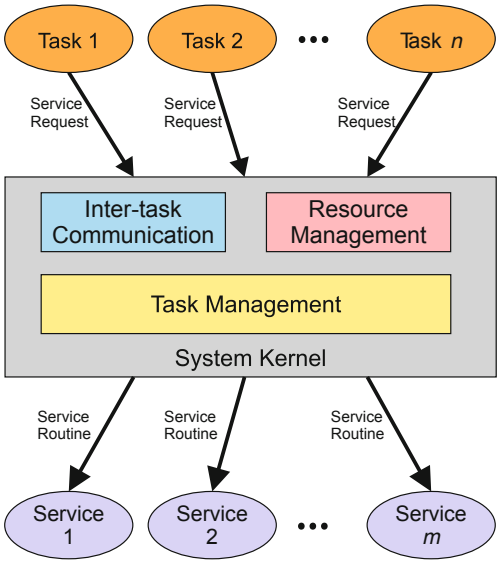

Fig. 1.9 Software structure in an embedded system

that enables a reliable inter-task communication to exchange information and to coordinate collaborative operation.

Services. Tasks are served through *Service Routines*. A service routine is a piece of code that gives functionality to a system resource. In some systems, they are referred to as device drivers. Services can be activated by polling or as interrupt service routines (ISR), depending on the system architecture.

1.3 Classification of Embedded Systems

The three pioneering microprocessor developments at the beginning of the 1970s, besides initiating the modern era of embedded systems, inadvertently created three defining categories that we can use to classify embedded systems in general: *Small, Distributed, and High-performance*. Figure 1.10 graphically illustrates the relationships among these classes.

Small Embedded Systems

Texas Instruments, with the TMS1000 created the microcontroller, which has become the cornerstone component of this type of embedded systems, which is by far, the most common type. This class is typically centered around a single microcontroller chip that commands the whole application. These systems are highly integrated,

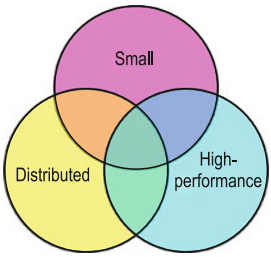

Fig. 1.10 Classification of embedded systems

adding only a few analog components, sensors, actuators, and user-interface, as needed. These systems operate with minimal or no maintenance, are very low cost, and produced in mass quantities. Software in these systems is typically single-tasked, and rarely requires an RTOS. Examples of these systems include tire pressure monitoring systems, microwave oven controllers, toaster controllers, and electronic toy controllers, to mention just a few.

Distributed Embedded Systems

The style created by Intel with the 4004 is representative of this type of embedded systems. Note that in this class we are not referring to what is traditionally known as a distributed computing system. Instead, we refer to the class of embedded systems where, due to the nature of the data managed by these systems and the operations to be performed on them, the CPU resides in a separate chip while the rest of components like memories, I/O, co-processors, and other special functions are spread across one or several chips in what is typically called the processor chipset. These are typically board-level systems where, although robustness is not a critical issue, maintenance and updates are required, and include some means of systems diagnosis. They typically manage multiple tasks, so the use of RTOS for system development is not uncommon. Production volume is relatively high, and costs are mainly driven by the expected level of performance. Applications might require high-performance operations. Applications like video processors, video game controllers, data loggers, and network processors are examples of this category.

High-Performance Embedded Systems

The case of the CADC represents the class of highly specialized embedded systems requiring fast computations, robustness, fault tolerance, and high maintainability. These systems usually require dedicated ASICS, are typically distributed, might include DSPs and FPGAs as part of the basic hardware. In many cases the complexity of their software makes mandatory the use of RTOS' to manage the multiplicity of tasks. They are produced in small quantities and their cost is very high. These are

the type of embedded systems used in military and aerospace applications, such as flight controllers, missile guidance systems, and space craft navigation systems.

As Fig. 1.10 illustrates, the categories in this classification are not mutually exclusive. Among them we can find "gray zones" where the characteristics of two or the three of them overlap and applications might become difficult to associate to a single specific class. However, if we look at the broad range of embedded applications, in most cases it becomes generally easy to identify the class to which a particular application belong.

1.4 The Life Cycle of Embedded Designs

Embedded systems have a finite lifespan throughout which they undergo different stages, going from system conception or birth to disposal and in many cases, rebirth. We call this the *Life Cycle of an Embedded System*. Five stages can be identified in the cycle: Conception or Birth, Design, Growth, Maturity, and Decline. Figure 1.11 illustrates the sequence of these stages, and the critical phases that compose each stage.

An embedded design is first conceived by the identification of a need to solve a particular problem, or the opportunity to apply embedded technology to an old problem. Many embedded systems are conceived by the need of changing the way old problems are solved or the need of providing cost-effective solutions to them. Other

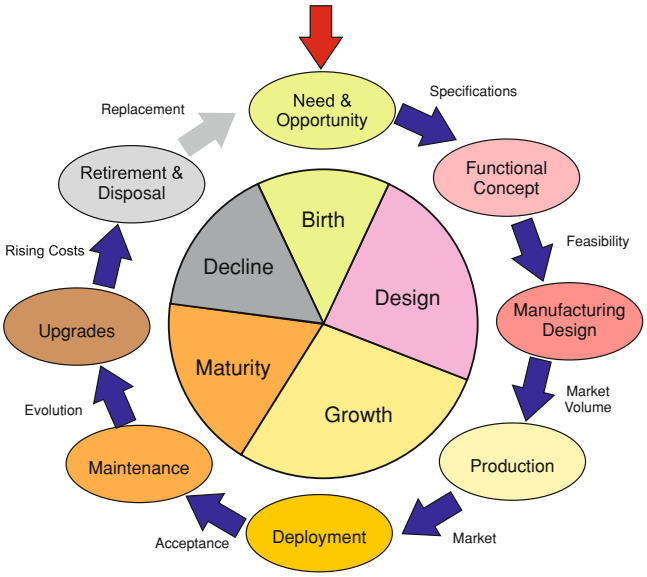

Fig. 1.11 Life cycle of an embedded system

systems are conceived with the problems themselves. Opportunity arises with the realization of efficient solutions enabled by the use of embedded systems. In any case, the application where the design needs to be embedded dictates the specifications that lead to the design stage.

In the design stage, the system goes first through a phase of functional design, where proof-of-concept prototypes are developed. This phase allows to tune the system functionality to fit the application for which it was designed. A feasibility analysis that considers variables such as product cost, market window, and component obsolescence determines whether or not a manufacturing and product design proceeds. This second design phase establishes the way a prototype becomes a product and the logistics for its manufacturing. Design is by far the most costly stage in the life cycle of an embedded system, since most of the non-recurrent engineering costs arise here. The expected market volume guides the entrance into the growth stage.

The growth stage initiates with the production of the system to supply an estimated market demand. Production is scheduled to deploy the system at the beginning of its market window. The deployment phase involves distribution, installation, and set-up of the system. The fitness of the system to its originating needs defines its acceptance, driving the embedded system into its mature stage.

In the maturity stage, the product is kept functional and up to date. This involves providing technical support, periodic maintenance, and servicing. As the application evolves, the system might require fitness to the changing application needs through software and/or hardware upgrades. The maturity stage is expected to allow running the product with minimal costs. As the system begins to age, maintenance costs begin to increase, signaling its transition into the decline stage.

In the decline stage, system components become obsolete and difficult to obtain, driving the cost of maintenance, service, and upgrades to levels that approach or exceed those of replacing the whole system at once. This is the point where the system needs to be retired from service and disposed. Product designs need to foresee this stage to devise ways in which recycling and reuse might become feasible, reducing the costs and impact of its disposal. In many cases, the system replacement creates new needs and opportunities to be satisfied by a new design, re-initiating the life cycle of a new system.

1.5 Design Constraints

A vast majority of embedded systems applications end up in the heart of mass produced electronic applications. Home appliances such as microwave ovens, toys, and dishwasher machines, automobile systems such as anti-lock brakes and airbag deployment mechanisms, and personal devices such as cellular phones and media players are only a few representative examples. These are systems with a high cost sensitivity to the resources included in a design due to the high volumes in which they are produced. Moreover, designs need to be completed, manufactured and launched in time to hit a market window to maximize product revenues. These constraints

shape the design of embedded applications from beginning to end in their life cycle. Therefore, the list of constraints faced by designers at the moment of conceiving an embedded solution to a problem come from the different perspectives. The most salient constraints in the list include:

Functionality: The system ability to perform the function it was designed for.
Cost: The amount of resources needed to conceive, design, produce, maintain, and discard an embedded system.
Performance: The system ability to perform its function on time.
Size: Physical space taken by a system solution.
Power and Energy: Energy required by a system to perform its function.
Time to Market: The time it takes from system conception to deployment.
Maintainability: System ability to be kept functional for the longest of its mature life.

Aside from functionality, the relevance of the rest of the constraints in the list changes from one system to another. In many cases, multiple constraints must be satisfied, even when these can be conflicting, leading to design tradeoffs. Below we provide a more detailed insight into these constraints.

1.5.1 Functionality

Every embedded system design is expected to have a functionality that solves the problem it was designed for. More than a constraint, this is a design requirement. Although this might seem a trivial requirement, it is not to be taken lightly. Functional verification is a very difficult problem that has been estimated to consume up to 70 % of the system development time and for which a general solution has not been found yet. The task of verifying the functionality of the hardware and software components of an embedded system falls in the category of NP-hard problems. Different methods can be applied towards this end, but none of them is guaranteed to provide an optimal solution. In most cases, the combination of multiple approaches is necessary to minimize the occurrence of unexpected system behavior or system bugs. The most commonly used methods include the following:

- Behavioral Simulation: The system is described by a behavioral model and suitable tools are used to simulate the model. This approach is more suited for embedded systems developed around IP cores where a behavioral description of the processor is available, along with that of the rest of the system, allowing verification to be carried early in the design process.
- Logic- and circuit-level Simulation: These methods apply to dedicated hardware components of the system. When custom hardware forms part of an embedded application, their functionality must be verified before integration with the rest of the components. Analog and digital simulation tools become useful for these chores.

- Processor Simulation: This method uses programs written to simulate the functionality of software developed for a target processor or microcontroller on a personal computer or workstation. Simulators become useful for verifying the software functionality off the target hardware. The functionality that can be verified through this method is limited to some aspects of software performance, but they provide an inexpensive way to debug program operation. Software simulators simulate hardware signals via I/O registers and flags. In general, they allow to examine and change the contents of registers and memory locations; and to trace instructions, introduce breakpoints, and proceed by single steps through the program being debugged.
- JTAG Debuggers: JTAG is an acronym for Joint Test Action Group, the name given to the Standard Test Access Port and Boundary Scan Architecture for test access ports in digital circuits. Many present development tools provide ways of debugging the processor functionality directly on the target system through a JTAG port. This is achieved through the use of internal debug resources embedded in the processor core. The IEEE-1149.1 standard for JTAG is one of the most commonly used ways of accessing embedded debugging capabilities in many of today's microcontrollers. JTAG debuggers allow to perform the same actions listed for simulators, but directly on the embedded system through the actual processor.
- In-circuit Emulation (ICE): Hardware Emulation is the process of imitating the behavior of a hardware component with another one for debugging purposes. An ICE is a form of hardware emulation that replaces the processor in a target system with a test pod connected to a piece of hardware which emulates the target processor. This box connects to a personal computer or workstation that gives the designer a window of view into the embedded application being debugged. ICEs are one of the oldest, most powerful, and most expensive solutions to debug and verify the functionality of embedded systems. They provide the designer with the ability of tracing, single stepping, setting breakpoints, and manipulating I/O data, memory, and register contents when debugging the application. They also give the designer control and visibility into the processor buses, the ability to perform memory overlay, and to perform bus-triggered debugging, where breakpoints are activated by the contents of specific hardware signals in the system. These are features not available through other hardware verification methods.
- Other Verification Tools: Other tools used for embedded system verification include Background Mode Emulators (BMEs) and ROM Monitors. A BME is similar to a JTAG debugger in the sense that it relies on a dedicated chip port and on-chip debugging resources to exert control over the CPU. However BMEs are not as standardized as JTAG ports. BMEs change in format and capabilities from one chip to another. ROM monitors are another type of debuggers that utilize special code within the application to provide status information of the CPU via a serial port. This last type relies on processor capabilities to introduce hardware breakpoints.

1.5.2 Cost

Developing an embedded system from scratch is a costly process. In a broad sense, the the total cost C_T of producing a certain volume V of units an embedded system can be obtained as,

$$C_T = NRE + (RP * V), \tag{1.1}$$

where *NRE* represent the Non-recurring Engineering (NRE) costs for that product, (*RP*) represents the Variable or Recurrent Production costs , and the production volume V is the number of marketed units.

The NRE costs include the investment made to complete all the design aspects of an embedded solution. This component, independent of the production volume, results from the addition of the costs of all engineering aspects of the system development including research, planning, software/hardware design and integration (prototyping), verification, manufacturing planning, production design, and product documentation.

The traditional design process of most embedded system solutions is developed around *commercial, off-the-shelf* (COTS) parts. Although other methodologies exist, standard COTS methods are the default choice for embedded system designers having commercial components as the hardware resources to assemble an embedded solution. These methods allow for minimizing the hardware development costs through tight constraints on physical parts components, usually at the expense of larger software design and system verification cycles. Figure 1.12 shows a representative diagram of a typical COTS-based design process for an embedded solution. The diagram details the steps in the system conception, design, and prototyping stages. It also includes the estimated time duration of the different stages in the process. It can be observed that although hardware and software design could be carried in parallel, their functional verification can only happen on a functional prototype. The combination of these steps can take up to 50 % of the product development time, with up to 75 % of the NRE costs.

Recurrent costs include all the expenses involved in producing the units of a given production volume. Recurrent costs include the cost of parts, boards, enclosures and packaging, testing, and in some cases even the maintenance, upgrade, and system disposal costs. Recurrent costs are affected by the production volume as indicated in Eq. 1.1, and are significantly lower than NRE costs. Representative figures for these terms on a given embedded product can be in the order of millions of dollars for the NRE costs while the RP cost could be only a few dollars or even cents. As an example consider the development of the first iPod, by Apple and sub-contractor PortalPlayer. Coming up with the functional product idea, defining its specifications, the product architecture; designing the hardware for each supported feature, the software for the desired functionality, debugging the system, planning its packaging, form factor, manufacturing plan, production scheme, and customer support took a team of nearly 300 engineers working for about one year. A conservative estimate puts NRE costs anywhere between five and and ten million dollars. The recurrent

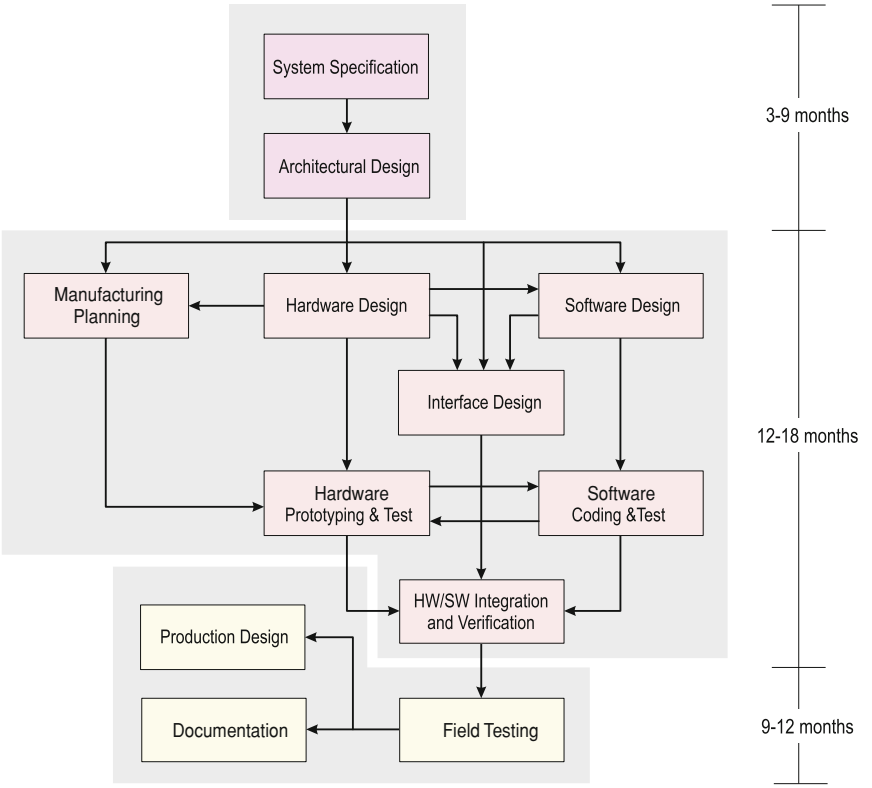

Fig. 1.12 Embedded systems design flow model based on COTS parts [8]

costs for a first generation player can be estimated around $120, considering that its architecture was based on two ARM7 MCUs, one audio codec, a 5 GB hard drive, lithium batteries, and miscellaneous knobs, buttons, display, and enclosure. Apple reported selling 125,000 iPods in the first quarter 2002, so let's assume their first batch of units consisted of exactly that number of units—that is a least a lower bound. With these numbers, and NRE set at $10 M, we can make a rough estimate of a total cost of at least $25 M for producing that first batch of units. Note that this is a simplistic estimate since it does not include the cost of items such as production testing, marketing costs, distribution, etc.

So, if it is so expensive to produce that kind of embedded system, how come they can be sold for as cheap as $300? The explanation for that is their production in large quantities. A common measure of the cost of an embedded system is the per-unit cost, U_C, obtained by dividing the total cost by the volume. Applying this formula we can estimate the cost of producing each unit in a batch of a given volume V, for a break-even. The selling price is then set considering the expected company revenues, among other factors.

$$U_C = \frac{C_T}{V} = \frac{NRE}{V} + RP \tag{1.2}$$

Equation 1.2 reveals that NRE costs are diluted by the production volume, making the unit cost of electronic systems affordable for the average individual. This formula also justifies the approach refining a design to minimize the cost of hardware, as is done with COTS parts, since the extra NRE investment is brought down when the production is large enough. Care must be exercised when deciding the level of optimization introduced into the design, since reduced production volumes could make the product cost prohibitively high. For our iPod example, the estimated unit cost would drop to about $200. As the product matures, bringing new versions into the market becomes much cheaper since IP can be reused bringing down the NRE costs. This explains why the next generation of a gadget, even when bringing new and more powerful features can be launched to the market at the same or even lower price than the previous generation.

1.5.3 Performance

Performance in embedded systems usually refers to the system ability to *perform* its function on time. Therefore, a measure of the number of operations per unit time will somehow always be involved.

Sometimes, performance is associated with issues such as power consumption, memory usage, and even cost; however, in our discussion we will restrict it to the system ability to perform tasks on time. Now, this does not make performance measurement an easy task. A number of factors need to be considered to measure or to establish a performance constraint for a system.

Given our definition, one might naively think that the higher the system clock frequency, the better the performance that can be obtained. Although clock frequency does have an impact on the system performance, it could end up being a deceiving metric if we do not have a more general picture of the system. Performance in embedded systems depends on many factors, some due to hardware components, some other due to software components. Hardware related factors include:

- Clock Frequency, the speed at which the master system clock ticks to advance time in the system. Given two identical systems, executing the exact same software, changing the clock frequency will have a direct impact on the system speed to finish a given task. However, decisions in most cases are not as simple. Changing for example from processor A to processor B we might experience that the same instructions take more clock cycles in B than in A, and therefore processor A could have a higher performance than B, even when B might be running at a higher clock frequency.

- Processor Architecture has a deep impact in the system performance. The use of multiple execution units, pipelining, caches, style (CISC[2] versus RISC[3]) buses, among other architectural differences affect the time the system takes to complete a task.
- Component Delay, i.e. the time it takes a component to react at its output to an input stimulus, will affect system performance. In the operation of synchronous bus systems, the maximum attainable speed will be limited by that of the slowest component. Semi-synchronous or asynchronous bus systems would slowdown only when accessing slow devices. A typical example of component delay affecting system performance is the memory access time. The time taken by a memory device to respond to a read or write operation will determine how fast the processor can exchange data with it, therefore determining the memory throughput and ultimately the system performance.
- Handshaking Protocols, i.e. the way transactions begin and end with peripheral devices and how frequently they need to occur will take their toll in the time the system takes to complete a task. Consider for example a device for which each word transfer requires submitting a request, waiting for a request acknowledgment, making the transfer and then waiting again for a transfer acknowledgment signal to ensure that no errors occurred in the transfer would function significantly slower than another where a session could be established, transferring blocks of words per handshaking transaction.
- Low-power Modes in hardware components also affect performance. When a component is sent to a low-power or sleep mode, either by gating its clock, its power lines, or some other mechanism, waking the device up consumes clock cycles, ultimately affecting its response time. In the case of devices able to operate at different voltage levels to save power, the lower the voltage the longer the response time, therefore reducing the system performance.

One consideration that needs to be addressed when dealing with hardware factors that affect performance is that high speed is expensive. The higher the speed of the hardware, the higher the system cost and in many cases the higher also its power consumption. Therefore the designer must exercise caution when choosing the hardware components of a system to satisfy specific performance requirements in not overspending the design. Once our embedded design reaches a level of performance that satisfies the expected responsiveness of its functionality, a higher performance becomes waste. Consider for example an embedded system design to give functionality to keyboard. The system must be fast enough to not miss any keystroke by the typist. A fast typist could produce perhaps 120 words per minute. With an average of eight characters per word, each new event in the keyboard would happen every 200 ms. We do not need the fastest embedded processor in the market to fulfill the performance requirements of this application. A different story would result if our embedded design were to interface for example a high-speed internet channel to a video

[2] Complex Instruction Set Computer.

[3] Reduced Instruction Set Computer.

processing unit. With data streaming at 100 Mbps or faster, the application requirements would demand a fast and probably expensive, power hungry, hardware design.

Software factors also affect system performance. Unlike hardware, we always want to make the software faster because that will make the system more efficient not only in terms of performance, but also in terms of power consumption and hardware requirements, therefore helping to reduce recurrent costs. Software factors affecting system performance include:

- Algorithm Complexity. This is perhaps the single most relevant software factor impacting system performance. Although algorithm complexity traditionally refers to the way the number of steps (or resources) needed to complete a task scales as the algorithm input grows, it also provides a measure of how many steps it would take a program to complete a task. Each additional step translates into additional cycles of computation that affect not only time performance, but also power consumption and CPU loading. Therefore the lower the complexity, the faster, in the long run, is the system expected performance. Some particular situations may arise for small input. For these cases it might occur that a higher complexity algorithm completes faster than one with lower complexity. This implies that a documented decision needs to consider aspects such as typical input size, memory usage, and constant factors among others.
- Task Scheduling determines the order in which tasks are assigned service priorities in a multitasking environment. There exist different scheduling algorithms that can be used when a system resource needs to be shared among multiple tasks. The rule of thumb is that one resource can be used by only one task at a time. Therefore, the way in which services are scheduled will affect the time it takes for services to be provided thereby affecting the system performance. Although in a single-task embedded system this consideration becomes trivial, multitasking systems are far common and meeting performance requirements makes it necessary to give careful consideration to the way application tasks will be scheduled and their priorities.
- Intertask Communication deals with the mechanisms used to exchange information and share system resources between tasks. The design of this mechanism defines the amount of information that needs to be passed and therefore the time overhead needed for the communication to take place. Therefore, the selected mechanisms will impact the system performance.
- Level of Parallelism refers to the usage given by the software to system resources that accept simultaneous usage. For example, an embedded controller containing both a conventional ALU and a hardware multiplier can have the capacity of performing simultaneous arithmetic operations in both units. A dual-core system can process information simultaneously in both cores. It is up to the software how the level of parallelism that can be archived in the system is to be exploited to accelerate the completion of tasks, increasing the system performance.

In summary, meeting performance constraints in an embedded system requires managing multiple factors both in hardware and software. The system designer needs to plan ahead during the architectural design the best way in which software and

hardware elements will be combined in the system to allow for an efficient satisfaction of the specific performance requirements demanded by an application.

1.5.4 Power and Energy

Power in embedded systems has become a critical constraint, not only in portable, battery operated systems, but for every system design. The average power dissipation of an embedded design defines the rate at which the system consumes energy. In battery powered applications, this determines how long it takes to deplete the capacity of its batteries. But aside from battery life, power affects many other issues in embedded systems design. Some of the most representative issues include:

- System Reliability: In digital electronic systems, which operate at a relatively fixed voltage level, power translates into current intensities circulating through the different system traces and components. This current circulation induces heat dissipation, physical stress, and noise in the operation of system elements, all associated to system failures. Heat dissipation is a major concern in system reliability due to the vulnerability of semiconductors to temperature changes. At high temperatures, device characteristics deviate from their nominal design values leading to device malfunction and ultimately system failure. The physical stress induced by large current densities on traces, contacts, and vias of integrated circuits and printed circuit boards is the primary cause of electromigration (EM). EM describes the transport of mass in metal under the stress of high current densities, causing voids and hillocks, a leading source of open and short circuits in electronic systems. Noise is generated by the pulsating nature of currents in inductive and capacitive components of digital electronic systems. As the current intensities increase with the power levels, so does the noise energy irradiated by reactive components, reaching levels that could exceed the system noise margins, leading to system malfunction.
- Cooling Requirements: A large power consumption leads to large levels of heat dissipation. Heat needs to be removed from the embedded electronics to avoid the failures induced by high temperature operation. This creates the necessity of including mechanisms to cool down the hardware. Increasing power levels rapidly scales forced air ventilation in static heatsinks to active heat removal mechanisms that include fans, liquid pumps, electronic and conventional refrigeration devices, and even laser cooling techniques. The necessity of including such components in embedded systems increases the system recurrent costs, and imposes physical requirements to the mechanical design in the form of increased size and weight, and shape limitations.
- Power Supply Design: All embedded systems need a source of energy. Portable systems rely on batteries. Stationary systems need power supplies. The larger the power dissipation of a system, the larger the size of its batteries or power supply,

and the lesser the chances of using energy scavenging techniques or alternate sources of energy. This also impacts system costs and mechanical product design.

- System Size, Weight, and Form: The mechanical design characteristics are affected not only by the requirements of power supply and cooling mechanisms, but also by the minimum admissible separation between components to limit heat density and reduce the risk of heat induced failures.
- Environmental Impact: The number of embedded systems around us continues to grow at accelerated rates. Estimates by the Consumer Electronics Association (CEA) show that nowadays an average individual uses about sixty embedded systems per day. Twenty years ago this number was less than ten. This growing trend is expected to continue for decades to come as embedded systems become more pervasive in contemporary lifestyles. Although this trend is fueled by the improvements they bring to our quality life, it is also placing a real burden on the planet energy resources and leaving a mark in the environment. A study of energy consumption by consumer electronics in the US commissioned by the CEA found that in 2006 11 % of the energy produced in the US, this is 147 TWh, was destined to household consumer electronics. Figure 1.13 shows the consumption distribution of the types the CEA included in the study. It can be observed that excluding personal computers, 65 % of the energy was consumed by devices we have catalogued as embedded system applications. It deserves to mention that the numbers in this study do not include digital TV sets, the electronics in major appliances such as dishwashers, refrigerators, ovens, etc.; nor the consumption by other segments of the population such as corporations, schools, universities, etc. Despite these exclusions, the numbers portrait the environmental impact of embedded systems and highlight the importance of tighter constraints in their power consumption.

Meeting the power constraints of an embedded application can be approached from multiple fronts:

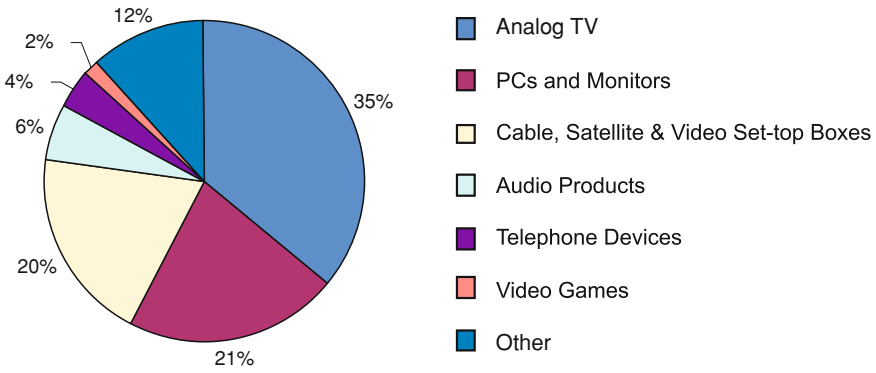

Fig. 1.13 Distribution of U.S. residential consumer electronics (CE) energy consumption in 2006. The total CE energy consumed was 147 TWh (*Source* Consumer Electronics Association)

- Hardware Design: Utilizing processors and hardware components specifically designed to minimize power consumption. Low-voltage, low-power peripherals, processors with power efficient standby and sleep modes, and low-power memories are some of the most salient types of components available for reducing the power consumption by the system hardware.
- Software Design: The way in which software is designed has a significant impact on the system power consumption. A given software function can be programmed in different ways. If each program were profiled for the energy it consumes during execution, each implementation would have its own energy profile, leading to the notion that software can be tailored to reduce the energy consumed by an application. This concept can be applied during programming if the programmer is conscious of the energy level associated to a programming alternative or exploited through the use of power optimizing compilers. The bottom line consideration here is that every cycle executed by the processor consumes energy in the system, therefore, the least number of execution cycles by the processor, the lower the energy consumed by the system.
- Power Management Techniques: Power management combines software strategies for power reduction with the hardware capabilities of the system to reduce the amount of energy needed to complete a task. This is perhaps the most effective way of developing power efficient applications.

1.5.5 Time-to-Market

Time to market (TTM) is the time it takes from the conception of an embedded system until it is launched into the market. TTM becomes a critical constraint for systems having narrow market windows. The market window W is defined as the maximum sales life of a product divided by two. Thus, the totals sales life of a product would be $2W$. The market window for embedded systems, as for any commercial product, is finite with profits usually following a Gaussian distribution with mean around W. Figure 1.14 shows a typical market window curve, illustrating an on-time product entry for a well managed time-to-market.

The beginning of the sales life imposes a strong constraint on the time-to-market since the total revenues, obtained as the area under the curve, are maximized when an on-time market entry is achieved. A delayed product deployment causes large losses of revenues. Figure 1.15 illustrates this situation with a simplified revenue model using linear market rise and fall behaviors.

When a product enters the market with delay D, its peak revenue R'_{max} is scaled by a factor $(1 - D/W)$. This model presumes that the delayed peak revenue occurs at the same point in time as the on time peak, and a maximum delay not exceeding W. The loss of revenue L can be calculated as the difference between the on-time total revenues and the delayed total revenues, obtained as indicated by Eq. 1.3.

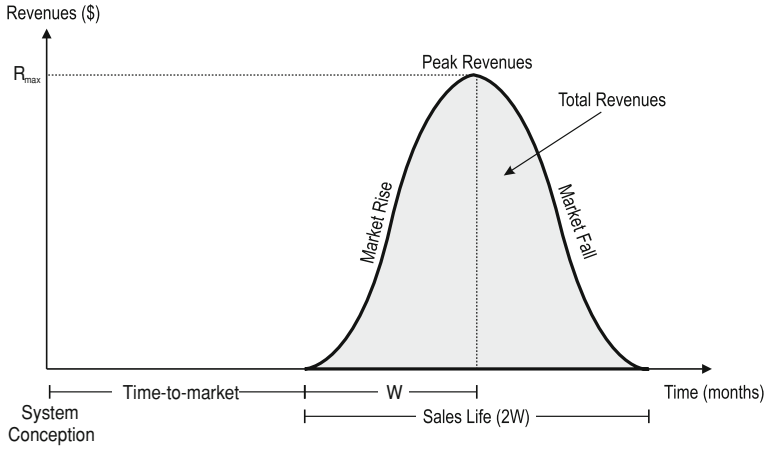

Fig. 1.14 Typical revenue-time curve for embedded products, denoting the time-to-market and market window

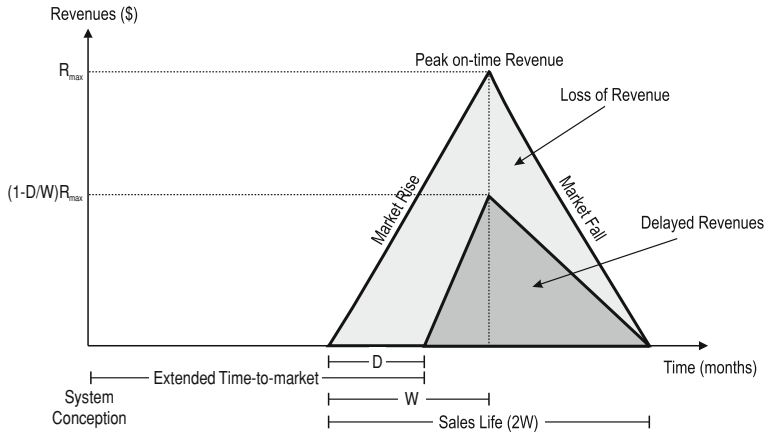

Fig. 1.15 Linear revenue model with a delayed system deployment

$$L = \frac{R_{max} D}{2W}(3W - D) \tag{1.3}$$

It is common to express losses as a percentage of the maximum on-time revenues. This can be obtained by dividing Eq. 1.3 by $R_{max} W$, resulting the expression in Eq. 1.4.

$$L\,\% = \frac{D(3W - D)}{2W^2} * 100\,\% \tag{1.4}$$

To numerically illustrate the impact of a violation to the time-to-market constraint, consider a product with a sales life of 24 months. A TTM extended by 10 weeks

($D = 2.5$) would cause a total revenue loss of 29 %. A 4-month delay would reduce revenues by nearly 50 %.

1.5.6 Reliability and Maintainability

Maintainability in embedded systems can be defined as a property that allows the system to be acted upon, to guarantee a reliable operation throughout the end of its useful life. This property can be regarded as a design constraint because, for maintainability to be enabled, it has to be planned from the system conception itself.

The maintainability constraint can have different levels of relevance depending on the type of embedded system being considered. Small, inexpensive, short lived systems, such as cell phones and small home appliances are usually replaced before they need any maintenance. In these type of applications, a reliable operation is essentially dependent on the level of verification done on them prior to their deployment. Large, expensive, complex embedded systems, such as those used in airplanes or large medical equipment, are safety critical and expected to remain in operation for decades. For this class of systems, a reliable operation is dependent on the system ability to be maintained.

From a broad perspective, maintainability needs to provide the system with means for executing four basic types of actions:

- *Corrective Actions*: that allow to fix faults discovered in the system.
- *Adaptive Actions*: which enable the introduction of functional modifications that keep the system operating in a changing environment.
- *Perfective Actions*: to allow for adding enhancements to the system functionally attending to new regulations or requirements.
- *Preventive Actions*: that anticipate and prevent conditions that could compromise the system functionality.

These actions need to be supported throughout the hardware and/or software components of the embedded system, but their implementation needs to deal with constraints inherent to each of these domains (Fig. 1.16).

Issues in Hardware Maintainability

Hardware designers face several limiting factors when it comes to integrating maintainability in their systems. These include:

- The cost overhead in the system NRE related to the design and validation of maintenance and testing structures.
- The impact on the time-to-market caused by additional development time required for the inclusion of maintainability features. This is also a factor in the software design.

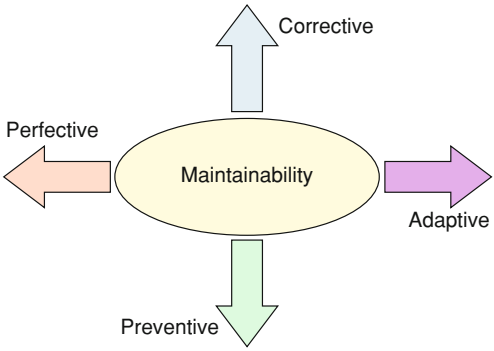

Fig. 1.16 The four actions supporting system maintainability

- The increase in recurrent costs caused by the additional hardware components needed in the system to support testing and maintenance.
- Other factors affecting hardware maintainability in deployed systems include problems of component obsolescence, accessibility to embedded hardware components, and availability of appropriate documentation to guide changes and repairs.

Issues in Software Maintainability

Supporting software maintainability in embedded systems represents an even greater challenge than hardware because of the way embedded programs are developed and deployed in such systems. In the application design flow, changes occurring either on the hardware or software components during their concurrent design, sharing software modules among product variants, the low-level requirements and tight interaction with the hardware, and the hardware constraint themselves are some of the characteristics that make embedded software maintainability particularly difficult. Results from representative case studies that include consumer electronics, communication equipment, aerospace applications, and industrial machinery have indicated that there are 10 major issues that need to be addressed in embedded software development to increase its maintainability.

1. Unstable requirements: Requirements change during the software development phase, which usually are not effectively communicated to all partners, creating problems in the hardware software integration, and therefore making maintenance more complex.
2. Technology changes: New technologies require new development tools which not necessarily preserve compatibility with early software versions, making difficult maintenance activities.
3. Location of errors: A common scenario in software maintenance is that the person assigned to fix a problem is not the one who reported the problem and also

different from the one who wrote the software, making the process of locating the error very time consuming. Moreover, the uncertainty of whether the error is caused by a software or a hardware malfunction adds to the difficulty.

4. Impact of Change: When a problem is identified and the remedial change is decided, identifying the ripple effect of those changes, such that it does not introduce additional problems becomes a challenge.
5. Need for trained personnel: The tight interaction between hardware and software components in embedded applications makes the code understanding and familiarization a time consuming activity. This calls for specialized training in an activity that results less glamorous and attractive to a new professional than designing and developing new code.
6. Inadequate documentation: All too common, the software development phase does not produce adequate, up-to-date documentation. Maintenance changes are also poorly documented, worsening the scenario.
7. Development versus application environment: Embedded software is typically developed on an environment other than that of the target application. A problem frequently caused by this situation is that real-time aspects of the system functionality might not be foreseen by the development environment, leading to hard to find maintenance problems.
8. Hardware constraints: Due to the direct impact of hardware in the system recurrent costs, stringent constraints are placed on hardware components. This requires software written in a constrained hardware infrastructure that leaves little room for supporting maintenance and updates.
9. Testing process: Due to the cost of testing and verification, it is common that only the basic functionality is verified, without a systematic method to ensure a complete case software test. This, albeit of the intrinsic complexity of the software verification process itself, accumulates to problems to appear later when the system is deployed.
10. Other problems: Many other issues can be enumerated that exacerbate the tasks of maintenance. These include software decay, complexity in upgrading processes, physical accessibility, and integration issues, among others.

1.6 Summary

This chapter presented a broad view of the field of embedded systems design. The discussion on the history and overview helped to identify the differences between traditional computing devices and embedded systems, guiding the reader through the evolution of embedded controllers from their early beginning to the current technologies and trends.

Early in the discussion, the tight interaction between embedded hardware and software and software was established, providing an insight into their broad structure and relationship. A broad classification of embedded systems was then introduced to identify typical application scopes for embedded systems.

The last part of the chapter was devoted to the discussion of typical design constraints faced by embedded system designers. In particular, issues related to system functionality, cost, performance, energy, time-to-market, and maintainability were addressed in attempt to create awareness in the mind of the designer of the implications carried by the design decisions, and the considerations to be made in the design planning phase of each application.

1.7 Problems

1.1 What is an Embedded System? What makes it different from a general purpose computer?
1.2 List five attributes of embedded systems.
1.3 What was the first electronic computing device designed with the modern structure of and embedded system?
1.4 What were the three pioneering designs that defined first microprocessors?
1.5 Which design can be credited as the first microcontroller design? Why is this design considered a milestone in embedded systems design?
1.6 What proportion of all designed microprocessors end up used in personal computers? From the rest, which CPU size has dominance in terms of units per year? Which generates the largest sale volume in dollars?
1.7 Write a list of at least ten tasks the *you* perform regularly that are enabled by embedded systems.
1.8 What are the fundamental components of an embedded system? Briefly describe how each is structured.
1.9 Think about the functionality of one simple embedded system enabled device you regularly interact with (for example, a video game console, cellular phone, digital camera, or similar) and try to come up with the following system-level items:

a. Functional hardware components needed.
b. Software tasks performed by the device.
c. Identifiable services that could be requested by each task and the hardware resource it uses.

1.10 List three embedded system applications for each of the classes identified in the discussion in Sect. 1.3. Provide at least one example of an application falling between two classes, for each possible combination of two classes. Can you identify one example with characteristics of the three classes? Briefly explain why you consider each example in the class where you assigned them.
1.11 Select one specific embedded system application you are familiar with and enumerate at least three important issues arising in each of its five life cycle stages. Rank them in order of relevance.
1.12 Perform an internet search to identify at least three commercially available verification tools for the Texas Instruments MSP430 family of microcontrollers.

Categorize them by the verification method they use. Comment on the cost-effectiveness of each.

1.13 What is the difference between non-recurrent (*NRE*) costs and recurrent production costs (*RP*). Use the diagram in Fig. 1.11 and identify which stages generate *NRE* costs and which stages generate recurrent *RP* costs.

1.14 A voice compression system for an audio application is being designed as an embedded system application. Three methods of implementation are under consideration: Under method A, the design would use a Digital Signal Processor (DSP); in method B, the design would be completed with an embedded microcontroller; while in method C, the design will be completed with an Application Specific Integrated Circuit (ASIC) where all functions will reside in hardware. Alternative A has NRE_a of \$250,000 and a recurrent cost per unit RP_a of \$75. Alternative B has $NRE_b = \$150{,}000$ and $RP_b = \$130$; while the ASIC solution has $NRE_c = \$2{,}000{,}000$ and $RP_c = \$20$.

a. For each of the three alternatives, plot the per-unit cost break-even price of the system production volume V.
b. Determine the production volume V where the ASIC solution becomes the most cost effective solution.
c. If the market for this product had an expected volume sales of 1,500 units, which implementation alternative would you recommend? Justify you recommendation.
d. In part c, what should the market price of the system be if the company expects a profit margin of 25 %? How much would the expected revenues be if all marketed units were sold?

1.15 Explain the difference between clock frequency and system performance in an embedded design. Depict a scenario where increasing the clock speed of an application would not result in an increase in performance. Explain why.

1.16 Enumerate three approaches each at the hardware and software level design that could impact the amount of energy consumed by an embedded application. Briefly explain how each impacts the energy consumption.

1.17 Consider the voice compression system of problem 1.14. For a market window of two years, and expected profit margin of 27 % in the marketing of 50,000 units implemented with alternative A, determine the following:

a. Determine the expected company revenues in dollars.
b. What would the percent loss of revenue be if the time-to-market were extended, causing a deployment delay of 6 months? How much is the loss in dollars?
c. Plot the cumulative loss of revenues of this product for delays in its time-to-market in a per-week basis.

1.18 Identify three issues each at the software and hardware design levels to consider when planning the maintainability of an embedded system application. Explain briefly how to integrate them in the design flow of the system.