1

The Art of Embedded Computers

The rapid evolution of embedded computers, along with the abundant educational possibilities they offer, has attracted the interest of instructors at all levels of education. This chapter recommends five distinct categories of embedded computers, in terms of the tasks linked either to *computer science (CS)* or *electronic engineering (EE)* discipline. An overview of this interdisciplinary technology between the two disciplines aims at helping instructors clarify the possibilities and limitations, as well as learning difficulties of each category. Then the chapter provides readers with a unique perspective on the educational aspects of microcomputer programming and application development. The analysis applies to the *technological pedagogical content knowledge* (*TPACK*) model and attempts to clarify why the programming language should be considered as the technology integration toward helping students create their knowledge about the subject matter. It also justifies why the employed technology may arrange the tutoring more appropriate for CS or EE students.

Subsequent to this analysis, the author explores the additional endeavor required to understand the capabilities of microcomputer technology and addresses the coined *micro-computational thinking* (*μCT*) term to describe the thought processes involved in the solutions, carried out by a microcomputer-based system. The author does not intend to coin a new term entirely differentiated from the existing *computational thinking* (*CT*) concept, but rather to reveal the thought processes related to the application development with embedded computers.

To understand the today's impact of microcontroller technology on the *maker* industry, the chapter also follows the advancement of microcontroller programming and application development and identifies the *long-cycle and short-cycle* development era.

Overview of Embedded Computers and Their Interdisciplinarity

Computer programming teaching and learning has been thoroughly researched throughout the years and at all levels of education [1–3]. Lately, instructors experience the widespread dissemination of embedded computer systems and the challenge to enhance students' perspective in the field of embedded computer programming and application

Microcontroller Prototypes with Arduino and a 3D Printer: Learn, Program, Manufacture, First Edition.
Dimosthenis E. Bolanakis.

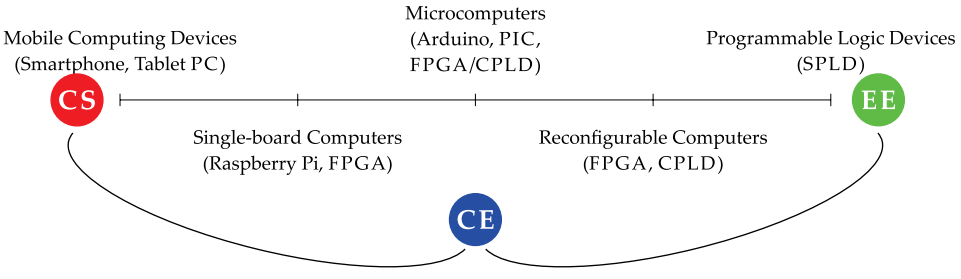

Figure 1.1 Interdisciplinarity of embedded computers.

development [4–6]. The questions raised are "What is the difference between the programming of a regular computer and an embedded computer system? What are the challenges for an educator who wishes to get involved with the second approach?"

It would be wise to start with a definition of the term *embedded computer*. Yet, it is sometimes essential to compromise with an informal definition of a complex and multifaceted theme that cannot be straightforwardly expressed within a single phrase. In consideration of the reader who is introduced to a field of study that does not necessarily fall within his/her area of expertise, the term is primarily addressed as follows. *Embedded computers* encompass any electronic device (contained in a hardware system) that can be programmed (with some kind of code) to carry out some computing. By definition, the process of developing programming code for an electronic device draws one's attention to an interdisciplinary task between the disciplines of CS and EE.

Figure 1.1 distinguishes five categories of embedded computers, in terms of the tasks that are more closely linked with the discipline of either CS or EE. To understand the position that each type of embedded computer holds on the proposed scheme, it is important to make reference to the basic features of conventional computer programming (which is originally rooted to CS).

Computer vs. Embedded Computer Programming and Application Development

The application software that is designed by a computer scientist or engineer to run on a personal computer interacts with the computer hardware through a system software known as the *operating system* (*OS*). The programming language used by the developer who builds custom-designed software incorporates utilities that are considered part of the OS. This set is referred to as the *application programming interface* (*API*) and encrypts the underlying hardware operations from the developer. For instance, the API in C programming language is declared in the header files, such as the "stdio.h," which embeds input and output functions. Hence, the developer learns how to exploit functions in order to *input/output* (*IO*) data to/from the computer system. Starting with the design of the simplest application, students are introduced to functions and syntax rules toward inputting/outputting data from/to the outside world. To do so, the students make use of two regular IO units of the computer, that is, the (input) keyboard and (output) monitor.

According to the aforementioned information, a computer program does not only direct the computer hardware toward computing tasks, it is also in charge of handling some IO units. Despite the standard keyboard and monitor units, the developer may perform more advanced IO operations through today's dominant computer interfaces, such as *universal serial bus* (*USB*), Ethernet, and so forth. In computer programming, advanced IO operations demand merely the calling of (perhaps) more complicated ready-to-use functions. The absence of these libraries would demand tremendous endeavor by the software developer just for the control of such IO units. One would have to go through several low-level tasks in order to access the computer hardware.

According to their complexity, an embedded system may or may not use an OS. If it does, then the emphasis is placed on software design tasks and the application development constitutes (more likely) a distinctive procedure for the computer scientist. Otherwise, the application development necessitates serious involvement with the hardware, and therefore it can be considered a more familiar territory for an electronic engineer. Due to the requisite accessibility at the machine level, the firmware development process has a need for special treatment of the incorporated programming language (compared to the software development for a personal computer, even if we use the exact same programming language). Moreover, an embedded computer application regularly incorporates nonstandard IO units that render the learning process of a novice designer more difficult.

All these aspects of complexity and interdisciplinarity of embedded computers are a case study of an interdisciplinary curriculum, which integrates partial training from CS and EE. This discipline is known as *computer engineering* (*CE*) and bears the major responsibility on the embedded computer programming and application development. However, due to the rapid evolution of this technology along with the abundant educational possibilities they offer, embedded computers often transcend the boundaries of CS, EE, and CE classes and migrate to several diverse disciplines [7, 8]. Some researchers have attempted to introduce embedded computers in K12 education, as well [9, 10]. In the following, the author proposes and overviews five distinct categories of embedded computers in order to help instructors clarify the possibilities and limitations and the learning difficulties of each category.

Group 1: Programmable Logic Devices

Adjacent to the discipline of EE discipline we could place the *programmable logic devices (PLDs)*. While this technology is nowadays considered obsolete, it is useful to be further explored in order to identify the (embedded computer) practices related to the discipline of electronics.

Twenty years ago, the development of an embedded computer system was not trivial. The process was regularly launched by the design of a *printed-circuit board* (*PCB*) for holding a set of carefully selected and interconnected components, outlining the functionality of the overall electronic system. If the system carried out some kind of digital computing, the PCB designer could select a PLD to build a reconfigurable digital circuit. For instance, the incorporation of a single PLD chip of some combinational logic that was determined by the designer, worked as a replacement for a few different logic gates (i.e. one chip replacing a few electronic chips). This option prevented the designer from wasting additional PCB

resources, and subsequently extra working time and effort, as the PCB design alone constitutes a particularly time-consuming task.

The programming languages of such devices are called *hardware description languages (HDLs)* as they are used for describing the behavior of an electronic (hardware) circuit. Figure 1.1 uses the term *simple programmable logic devices (SPLDs)* for this technology, after the naming decided by Atmel Corporation (products that nowadays belong to Microchip Inc.). SPLDs are of electrically erasable flash memory technology, while a popular HDL that is used for their configuration is known as CUPL.

Group 2: Reconfigurable Computers

Near to the EE discipline, but with a greater distance from the allied technology of SPLDs, are two comparable technologies known as *complex programmable logic devices (CPLDs)* and *field-programmable gate arrays (FPGAs)*. To avoid too many technical details, we consider CPLDs as an advancement of the SPLDs, with gates, flip-flops, and fast IO pins, where the designer can upload more complicated digital circuits. FPGAs, on the other hand, offer even more IO pins as well as resources for the configuration of particularly sophisticated and high-speed designs. Therefore, FPGAs constitute an ideal solution for testing complicated circuits before proceeding to the production of an error-free *application-specific integrated circuit* (*ASIC*). Because of the possibility of implementing high-speed and complex computations along with the fast IO response, FPGAs has played an important role in fast data acquisition systems, like, for example, the ones that are used in the high energy physics experiments at CERN [11].

The dominant manufacturers of FPGAs and CPLDs are Altera and Xilinx, while the most widely used HDLs for their configuration are Verilog and VHDL. In contrast to the conventional programming method for computers (that is, the sequential programming where statements are executed in their order of occurrence in the code), HDLs are addressed for expressing concurrency. Therefore, the designer must have (during the programming process) a clear sense of the digital logic circuits that describe the structure and behavior of the potential electronic system (to be uploaded into the FPGA/CPLD). This is the reason why these two technologies are placed nearby the EE discipline, in Figure 1.1. We may also acknowledge the sequential programming in FPGAs and CPLDs and in that particular case, FPGAs and CPLDs can be shifted nearer in the direction of CS discipline.

The term *reconfigurable computer* is sometimes addressed to illustrate FPGA technology. Because CPLDs have been advanced to the point that can be exploited by other means, too (rather than just implementing the behavior of a digital circuit), we will use this term to describe both FPGAs and CPLDs (in terms of their position in Figure 1.1). All these issues will be discussed next in the chapter.

Group 3: Microcomputers

The term *microcomputer* has been entirely changed from its original meaning. To thoroughly illustrate this concept we need to identify the fundamental parts that compose a common computer system. These are: (i) the processor (which is responsible for the execution of machine instructions generated by a computer software), (ii) the memory (i.e. program and data memory, where the former holds the machine instructions and the latter, the data that may occur during the code execution), and (iii) the IO devices (that is, the

means by which the IO units are connected to the computer so as to insert/pull-out data into/from the application system).

Based on the aforementioned definition, *microcomputer* term is regularly referred to as a complete computer embedded to a single chip. A popular microcomputer in our age is the microcontroller. The *microcontroller* (*μC*) is a chip device that incorporates program and data memory as well as IO devices that end up to the device's pins. Therefore, to build a microcomputer-based application the designer needs to connect some IO units to the microcontroller's pins (e.g. switches, keypads, sensors, liquid-crystal displays) and develop the code that inputs/outputs data to/from the microcontroller, as well as perform some computing. Today, the *do it yourself* (*DIY*) and *maker* cultures have established a wave of ready-to-use board systems for microcontrollers, which, along with the availability of free and shareable libraries over the internet, they have rendered feasible the rapid development of microcontroller-based applications. The designer may purchase a microcontroller-based "motherboard" as well as a separate (and compatible) daughterboard that employs, for instance, a Bluetooth module. Attaching the boards together and then utilizing a shareable library, the designer can straightforwardly implement an application that exchanges data between the microcontroller and a mobile phone.

The programming of a μC has dramatically changed the last decade. In the previous era, the μC was programmed with an assembly-level approach. This option required a deep understanding of the microcontroller's internal structure and a fluent handling of the assembly programming language. This programming method, together with the need for designing and developing a PCB from scratch, would have placed the microcontroller technology closer to the EE discipline. However, the passage to today's era where a higher level of programming is being dominated, and the PCB design is being eliminated (because of the available DIY board systems for microcontrollers), the μC technology could be positioned at the middle of CS and EE disciplines (Figure 1.1).

The most popular and widely used method to program a microcontroller device, nowadays, is the embedded C programming (which is based on the well-known C programming language plus a set of extensions). Because of the absence of an OS, the firmware development for a microcontroller device entails several register access operations in order to perform particular IO tasks with the outside world. This is perhaps the most important differentiation from the regular C programming method of a personal computer. Other extensions of familiar programming languages, such as the Pascal and Python, have been addressed for microcontroller programming, while the DIY culture of our age has engaged hobbyists' interests in the use of Arduino programming. The Arduino programming approach is based on ready-to-use libraries and modules, where the programming process is focused merely on the top-level code. However, the development of a microcontroller-based system cannot be defined by fixed hardware and, hence, learning microcomputer programming and application development entails involvement, to some extent, with some hardware resources [12].

Microcontrollers share this interdisciplinary (and equally distributed in both disciplines) position of Figure 1.1 with the CPLD and FPGA technologies (of the previous group of embedded computers). This is because CPLDs and FPGAs offer the possibility of building a microcontroller core within the reconfigurable computer, such as the PicoBlaze 8-bit microcontroller for the devices of the manufacturer Xilinx. This possibility relieves the

demanding process of expressing concurrency of a digital computing system (as the incorporation of a sequential logic is, in many cases, required).

Group 4: Single-Board Computers

Next and near the CS discipline we may identify the single-board computers. Based on the aforementioned analysis on the computer's fundamental parts, as well as the title in this category, we refer to ready-to-use boards that incorporate a processor (or microprocessor), memory, and IO devices. This group of embedded computers is considered to run an OS and, hence, the programming approach looks much like the process followed for a regular computer.

A popular type of embedded computers in this category is the Raspberry Pi. This particular single-board computer incorporates a slot that accepts an SD card for holding the OS, while the designer may connect a monitor and a keyboard to the available HDMI and USB IO connectors, and program the Raspberry Pi as a common computer. The code development can be performed with a regular programming language, such as Python and C. Moreover, the board employs a connector with *general-purpose input/output* (*GPIO*) pins, which can be exploited as per need (and similarly to the pins of a microcontroller device). This additional feature of Raspberry Pi (and of other comparable boards), along with the fact that runs an OS, defines the position they hold on the diagram of Figure 1.1 (influenced more by CS and less by EE discipline).

We observe that single-board computers share this position in the diagram of Figure 1.1 with the FPGA technology (also present in the previous two groups of embedded computers). This sharing arises from the fact that the contemporary technology of FPGAs can be either delivered with a hardwired processor core, or being configured with a software-defined processor (such the Xilinx MicroBlaze 32-bit architecture), and thereafter to be used in association with an OS. It is worth noting that FPGAs constitute a particularly complicated technology, but also flexible in consideration of the way that is being utilized, as revealed by three positions they hold in the diagram of Figure 1.1.

Group 5: Mobile Computing Devices

The final group of embedded computers (specified as mobile computing devices) refers to all portable computers that are delivered with a mobile OS (such as the Android and iOS). Using this technology, the designer has an opportunity to develop software components (also known as *apps*) and exploit the device's IO units in a custom-designed application. Modern handheld and portable computers (such as the smartphones and tablets) embed several sophisticated components; that is, the touch screen, sensors (accelerometer, gyroscopes, barometers, etc.), communication interface modules (e.g. Bluetooth and WiFi), and so forth.

The programming and application development for this kind of embedded computers is limited to the utilization of a standard set of hardware devices, where low-level practices are encrypted by the OS. This is the reason why mobile computing devices are located this close to the CS discipline. However, the cutting-edge technology of this category provides to the educator several experimentation possibilities. For instance, an application development that utilizes the barometer sensor can direct the students toward an inquiry-based learning approach in barometric altimetry [13].

The code development method for a mobile computing device depends on the employed OS. For instance, the software development for an Android OS is usually performed in Java using the Android *software development kit* (*SDK*); the SDK refers to a set of software tools for developing apps for particular software and hardware platforms. There are also other tools for *apps* development, appropriate for novice programmers. A popular one is the *App Inventor* for Android, which is supported by the *Massachusetts Institute of Technology* (*MIT*) and provides a drag-and-drop method of programming.

TPACK Analysis Toward Teaching and Learning Microcomputers

Microcontrollers have become very popular lately. Because of the widespread dissemination of DIY culture in microcontroller technology, it has nowadays been developed an entire industry that is meant for novice designers. This technological evolution has also spread like a virus to the *maker* culture of our age, as the innovation with microcontroller-based electronics can be addressed for several interdisciplinary practices (e.g. innovative experiments in science, wearable electronics with art, and much more). Nowadays, microcontrollers constitute a low-cost and easily accessible technology that can be incorporated by every single *makerspace* and *Fab Lab* [14, 15]. Because of the *maker* movement in education (and today's rapid evolution of embedded computers), it would be wise to consider a possible increase to the educational research efforts related to microcomputer programming, at expense of or complementary to the conventional computer programming learning. Following an educational research on microcomputer practices appropriate for undergraduate students within computing curricula[16–19], the author provides a personal perspective on the teaching and learning aspects of microcomputer programming and application development. A *TPACK* analysis of the interdisciplinary technology of microcontrollers is performed hereafter.

TPACK Analysis of the Interdisciplinary Microcontroller Technology

TPACK constitutes a framework that enhances the initial perception on *pedagogical content knowledge* (*PCK*) [20, 21]; i.e. the intersection between the *content knowledge* (*CK*) and *pedagogical knowledge* (*PK*). The former refers to the instructor's knowledge domain about the subject matter to be learned or taught, and the latter to the knowledge domain of teaching and learning methods.

Building on this concept, TPACK incorporates another component to describe the technology integration in the framework, as defined by Pierson in 2001 [22]. Initially referred to as TPCK, today's structure of the model has been illustrated by Mishra and Koehler, in 2006, as a Venn diagram [23]. The latest component in the model, that is, the *technology knowledge* (*TK*), determines the role of technology in improving education. TK reveals three more intersections: (i) the *technological pedagogical knowledge* (*TPK*) found in between TK and PK, (ii) the *technological content knowledge* (*TCK*) found in between TK and CK, and (iii) the TPACK, which is the intersection of the three components (CK, PK, TK).

The analysis on TCK and TPK interceptions [23, 24] mentions (among others) that teachers should be aware of how the technology application affects the subject matter (i.e. TCK intersection) as well as teaching (i.e. TPK intersection). Based on the aforementioned suggestions, this subsection addresses the TPACK model to share a personal perspective on the three components CK, PK, TK in (i.e. *what* we know, *how* we teach, and *why* the selected technology has a serious effect on) microcontroller-based education. Then, it justifies why the employed technology may cause a shift in microcomputer education between disciplines of EE and CS. When necessary, the parallelization between computer and microcomputer programming and application development is addressed so as to illustrate these concepts (in terms of the sequential method of computer programming, which is the standard programming method for microcontrollers).

Content Knowledge (The What)

In their survey of teaching introductory programming, Pears et al. [2] identify that the traditional views of learning programming (including the associated textbooks) are orientated toward the syntax and structures of the employed programming language. Yet, they also identify the strong movement in CS education and their staunch supporters who consider programming as the application of skills toward solving a problem.

Nowadays, there are several general-purpose programming languages that can be used to direct computers (and microcomputers) toward solving a problem. Some of them support a kind of higher-level approach of programming (such as the Python, where programs are developed in fewer code lines to support code's readability). Others provide a kind of lower-level programming approach (such as the C language, the constructs of which map more effectively with instructions at the machine level). While the approach of programming can be varied from the viewpoint of the code development process (when different languages are addressed), the result in the way generalized mechanics are implemented at the machine level relies on the same concepts.

I argue, too, that learning of computer and microcomputer programming should be considered as the development of skills toward solving a problem. Therefore, the programming language should not be considered as being incorporated in the CK component. Rather, it should be considered as the technology integration toward helping students create their knowledge about the subject matter. The latter should more appropriately be defined as the concepts and theories of the sequential method of computer programming toward solving a problem. Some of today's most popular programming languages may be obsolete in a few years. Yet, the computer programming topics, such as the methods of alternating the normal execution of the code (aka *control flow*), the arrangement and accessing of data in memory, etc., will also be present in tomorrow's popular programming languages.

The aforementioned concepts and theories of CK component, in regard to a potential introductory programming course, are related merely to *software practices* (as hardware practices are encrypted by the computer system). A corresponding introductory course on microcomputer programming would require some hardware practices, as well. The difficulty in arranging a technology-related course, textbook, etc., is in agreement with the fact

that the employed technology will most possibly be obsolete in a few years. The question raised is "How could we arrange CK of a technological subject matter without too much focus on the specified technology?"

Technology Knowledge (The Why)

When it comes to microcontroller education, the procedure needed to defocus from a particular technology is not trivial. To build some experiments you need to arrange the examples around a particular microcontroller device. And there is no compliance, even with devices of the same manufacturer. Chip vendors supply different models with more or fewer differentiations in the architecture in order to cover a large number of buyers (according to the needs specified by a custom-designed application). Such differentiations are the memory locations of the internal registers, the available IO subsystems embedded in the microcontroller (such as timers and analog-to-digital converters), and so forth. This is why typical textbooks in microcontrollers regularly incorporate a description of the device's architecture [25–27].

Nevertheless, there are strategies that can be addressed in order to defocus from a particular technology in microcontroller education. One of them could be the incorporation of the external subsystem, instead of the complex internal subsystems available in the employed microcontroller device. In this way, the information related to the architecture of a particular device is limited to the knowledge of issues, such as how to use the microcontroller pins as regular inputs/outputs. Of course, the incorporation of external subsystem increases the PCB requirements; however, the DIY culture of our age facilitates this option.

Another strategy could be the arrangement of experiments around long-lasting technologies. The regularly limited pin resources in a microcontroller device create the need for serial communication interfaces in typical microcomputer applications. Particular examples of long-lasting types of serial protocols are the *universal asynchronous receiver/transmitter* (*UART*), *serial peripheral interface* (*SPI*), and *inter-integrated circuit* (*I2C*), which have been for many years used in microcontroller applications, (and there more to come as there is a tremendous development of chips based on these three protocols).

These serial communication protocols are included as standard IO subsystems available in several microcontroller devices. However, the configuration and control of such subsystems are performed through the accessing of three sorts of registers; that is the *control*, *status*, and *data* registers. The contents and memory addresses of these registers, as well as the pins location of the specified subsystems, differ from device to device. Hence, the learner has no choice but study the architecture of the employed microcontroller device in solving the problem. A possible strategy toward implementing an architecture-independent serial interface is to manually implement the process through the regular IO parallel interface of the microcontroller pins [28, 29].

Figure 1.2 recommends a revision of Kordaki's [30] problem-solving model for beginners learning programming (using C programming language). The revised scheme illustrates how the hardware practices in microcomputer programming are differentiated from the software practices of a regular computer (when addressing a high-level programming approach). The dashed-line arrows depict at which point (of the problem-solving process)

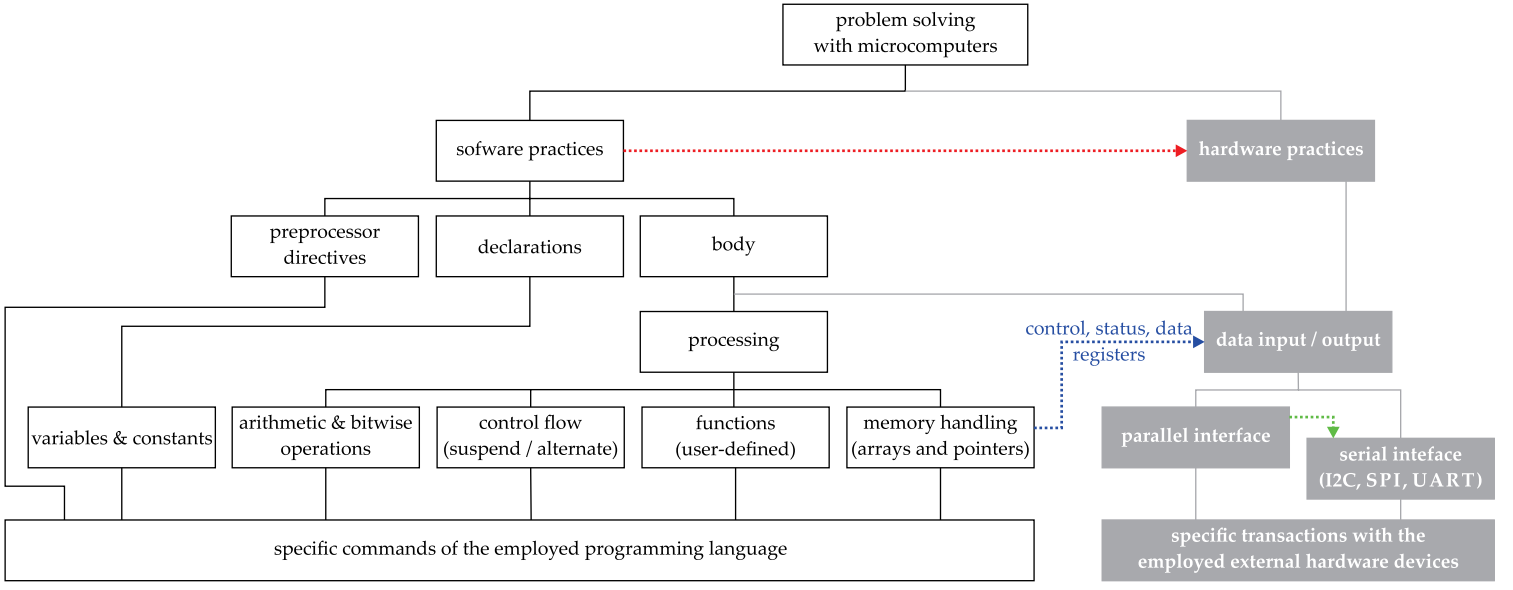

Figure 1.2 Problem-solving with microcomputers (a revision of Kordaki's model).

the differentiation is achieved. That is, the *data* IO processes to/from the microcontroller device, which are handled by the corresponding memory locations.[1] The dashed-line arrow pointing from the *parallel interface* to the *serial interface* block illustrates the alternative choice of implementing architecture-independent serial interfaces in order to defocus from a particular technology [28].

To reply to the question *"why the selected technology has a serious effect on microcontroller education?"* we need to consider what happens if a low-level programming language is addressed for the tutoring system. This choice would place the *memory handling* block immediately after the *processing* block (Figure 1.2). What this means is that all software practices would be related to the microcomputer architecture. For example, to implement an iterative loop the designer should: (i) utilize a register from data memory as counter; (ii) decrease the content of the counter at each repetition of the loop as well as evaluate its content, and (iii) explore the content of the processor's register bits in order to decide the alternation of the program flow. Such processes shift away from the software practices familiar to CS students and are directed toward the hardware practices appropriate for EE.

The assembly language learning requires a particular endeavor, whereas the time for practice within an introductory class is already limited. Subsequently, the possibility of the learner to develop skills in solving problems would be considerably decreased [31]. This would affect the subject matter of CK in the TPACK model; i.e. because of the influence generated by the TCK intersection. In that case, the tutoring would most probably be focused on learning the syntax and structures of the assembly programming language, rather than on problem-solving practices. In addition, the process of defocusing from a particular technology (i.e. from a particular microcontroller device and its identical instruction set architecture) would almost be impossible [16].

Pedagogical Knowledge (The How)

Figure 1.3a presents the generalized scheme of the TPACK model, while Figure 1.3b presents a personal perspective of the model being applied to the recent trends in microcontroller education. Arranging a course (or textbook) around a regular (high-level) programming approach, the subject matter of the CK component can be equally shared in software and hardware practices. This allows addressing efficient pedagogies, such as the *project-based learning* (*PBL*) approach which constitutes one of the most popular research methods in microcontroller education for many years now [32, 33]. This student-centered pedagogy approach cannot be easily addressed with a low-level programming approach. As mentioned earlier, the difficulties in learning (as well as developing code in) assembly language absorb considerable time resources from the students' practice. The process of facilitating the programming language learning by the incorporation of a high-level approach allows us to address such pedagogies (i.e. TPK intersection), which consequently support the arrangement of the subject matter (i.e. PCK intersection) toward engaging students in investigation tasks, collaborative development of the project, etc.

---

1 No process is depicted subsequent to the corresponding blocks of the original hierarchical network (referred to as *data entry* and *output* blocks in Kordaki's scheme); a choice that clearly portrays the encrypted IO operations in learning computer programming. Please refer to Kordaki's original scheme [30] for this particular information.

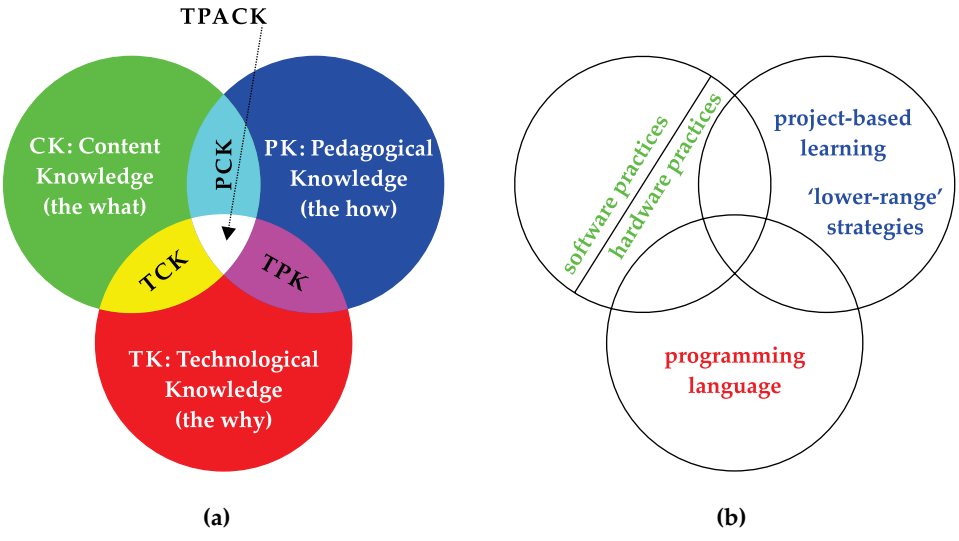

Figure 1.3 (a) TPACK model and (b) its application in microcontroller education.

Other "lower-range," though of particular importance, strategies are also affected by the incorporated technology (i.e. TPK intersection). In detail, the selection of low-level programming approach may create the need for teaching strategies that facilitate the assembly language learning [18]. On the other hand, the arrangement of tutoring toward a higher-level programming approach could rely merely on the utilization of more transparent teaching strategies, such as *flowcharts* and *pseudo-codes* (that also aim at facilitating the language learning). It is worth noting that *flowcharts* are in agreement with Levin's suggestions [34] on the transformational picture function, appropriate for difficult-to-remember information. On the other hand, the *pseudo-code* works as an intermediate step between the programming steps depicted by a flowchart, and the strict syntax required by the development of the final code. While the utilization of *flowcharts* and *pseudo-codes* are taken for granted in computer programming learning, they are of particular importance since the learning endeavor of a programming language, especially for a beginner, is not negligible.

From Computational Thinking (CT) to Micro-CT (μCT)

This subchapter explores the additional endeavor required to understand the capabilities of microcomputer technology and addresses the coined μCT term to describe the thought processes involved in the solutions carried out by a microcomputer-based system. Embedded computers have become an integral part of everyday life as they can be found in almost every single electronic device. If the most effective approach for developing CT is learning computer science, it would be wise to consider today's widespread dissemination of embedded computes and promote μCT concept within the computing curricula. The purpose of this effort is not to coin a new term entirely differentiated from the existing

CT concept, but rather to reveal the thought processes related to the application development with embedded computers.

In addition to the justification and detailed description of the μCT term, the information presented hereafter introduces the terms (i) *pseudo-architecture* and (ii) *pseudo-timing diagram*, and (iii) *pseudo-hardware*, complementary to the well-known and "transparent" teaching method of *pseudo-code* (used to support students' learning processes). Like the *pseudo-code* is addressed to describe the operating principles of a computer code (which is independent of language-specific rules and does not rely on to a standard format), the *pseudo-architecture*, *pseudo-timing diagram*, and *pseudo-hardware* are addressed for illustrating the hardware operating principles of microcomputers, while also making a clear link between the firmware execution and the hardware response.

CT Requirement and Embedded Computers

In 2006, Jeannette M. Wing introduced the term CT to share a vision that everyone can benefit from thinking like a computer scientist [35]. The term has received considerable attention since then, while also raising an ongoing discussion about the definition and essence of CT. Wing revisited the CT term and described it (in 2011 [36]) as "the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent." While such a high-level definition kept the debate warm, some educational researchers found the term acceptable and focused on how to promote CT in education [37, 38]. In 2016, the *K–12 Computer Science Framework Steering Committee* [39] recognized CT as the heart of the CS practices. In detail, seven core practices represent what computationally literate students should do to fully engage with the CS core concepts of (i.e. what they should know), where four of the overall seven practices are aspects of CT (that is, *Recognizing and Defining Computational Problems*, *Developing and Using Abstractions*, *Creating Computational Artifacts*, *Testing and Refining Computational Artifacts*). It would be wise to stand aside from this debate on CT definition and focus the reader's attention on what CT requires [39], as defined by the *K–12 Computer Science Framework* (p. 68):

"Computational thinking requires understanding the capabilities of computers, formulating problems to be addressed by a computer, and designing algorithms that a computer can execute. The most effective context and approach for developing computational thinking is learning computer science; they are intrinsically connected. Computational thinking is essentially a problem-solving process that involves designing solutions that capitalize on the power of computers; this process begins before a single line of code is written. Computers provide benefits in terms of memory, speed, and accuracy of execution. Computers also require people to express their thinking in a formal structure, such as a programming language."

Given the aforementioned description and taking into account the widespread dissemination of embedded computers in everyday life, the question raised is: What critical thought processes are involved in the solutions carried out by embedded computers? To answer this question, we first need to explore the different types of embedded computer systems as well as the possibility of being exploited by a broad range of users. As Aho remarks [40], "the

term computation means different things to different people depending on the kinds of computational systems they are studying and the kinds of problems they are investigating." According to the proposed categorization of embedded computers presented earlier in this chapter, we may identify two major categories in accordance to the applied method of programming. Accordingly, we may address an HDL (e.g. VHDL) to express concurrency of a digital computing system, or an imperative style of programming that applies to a common programming language (C, Python, etc.). Leaving aside the former type of embedded computers (such as the reconfigurable FGPA technology) as they require advanced skills not suitable for inexperienced learners, the second group (more appropriate for novice designers) can be further separated into two subcategories. Accordingly, in the electronics industry we may identify embedded computers that run an OS (such board systems, like the popular Raspberry Pi, regularly employ microprocessor technology), as well as embedded computers that do not use an OS (such board systems, like the familiar Arduino, incorporate microcontroller technology).

Microcomputers and Abstraction Process

Microprocessor (*μP*) and *μC* devices are regularly referred to as microcomputers, even though it is more precise to associate the second technology with the generalized term of microcomputers. This is because μCs embed the processor unit, the (program and data) memory as well as the IO devices (that interface with the outside world) within the same *integrated circuit* (*IC*) and, hence, they compose single-chip complete computers. The information given hereafter applies to both systems and therefore, μCT term may refer to either a μP-based or μC-based system. In regard to μP technology which is commonly found in boards running an OS, the information applies to the nonstandard hardware interfacing practices. Such practices are related to the endeavor required to plug hardware devices to the GPIO port pins of a board (e.g. a Raspberry Pi single-board computer) and to develop a custom-designed application. This kind of practices is associated with the thought processes the coined μCT term is meant to highlight. To make μCT term even clearer, particular attention is paid to the OS-less μC-based system. Microcontrollers (i.e. a technology that was primarily meant for electrical/electronic engineers) have become very popular the last decade and have been gradually shifting in the direction of CS discipline. The answer key for this reallocation is hidden behind the most common used word in the definition of CT (as depicted by the familiar word-cloud graph of reference [41]); that is, the abstraction.

In [42], Wing clearly explains the significance of abstraction in computing and the importance in deciding what details to highlight or ignore. When working with regular computers the abstraction is generally a clearer process. However, when working with μCs the thought processes become more complicated and the decision on the abstraction is quite critical. The cause is mainly due to the fact that:

a) μC-based applications are arranged around a set of nonstandards IO units (LEDs, switches, keypads, sensors, actuators, etc.) and interfaces (USB, Ethernet, wireless, and so forth);
b) The OS-less system of a μC requires specific configuration, and this configuration might differ a lot from device to device.

The *DIY* culture in μC technology has accomplished a major achievement in the delivery of abundant low-cost stackable hardware platforms, which render the implementation custom-designed systems as easy as one-two-three. In regard to the software domain, the DIY culture is orientated toward the development of drivers of the employed external subsystems that abstract low-level tasks, such as, the complex configuration of an Ethernet module. This is a desirable process of abstraction, which is also in agreement with the directions given in P4.2 description of *K–12 Computer Science Framework* (p. 78) [39]. That is, the students should be able to use well-defined abstractions that hide complexity, and understand that they do not need to know the underlying implementation details of the abstractions they use. Similarly, the well-defined interfaces between the layers of abstractions in a common computer enable developers to interact with components, without needing to know all the details of the component's implementation [42].

The aforementioned description illustrates the contribution of DIY culture in shifting μC technology closer to CS discipline, by placing the emphasis on the abstraction process of the incorporated peripheral devices. However, the developer of a μC-based application works with two distinct elements; that is, (i) the μC device and (ii) the peripheral devices (such as, sensors, interface modules, etc.). *The most critical process in the programming and application development with microcomputers is the signal transaction between the processor and a peripheral device*. Because of the absence of an OS, the modern development environments for writing μC code encompass built-in features and libraries that tend to encrypt such transactions. However, those low-level processes should not be delivered totally encrypted to the learners, but rather the students should be able to develop their own abstractions, as defined by *P4.3 guidelines* of *K–12 Computer Science Framework* (p. 79) [39]. *The most common way to identify and fix errors in a custom-designed system is to explore the signal transaction between the μC and a peripheral device*. For instance, the option a designer regularly holds in order debug and restore the communication between two wirelessly interfaced μC-based setups, is to evaluate the exchanged information between the μC and the incorporated module, which establishes the wireless interface. This might seem like a "scary" process for a novice designer, however, there are only a few dominant hardware interfaces used in microcomputers. This is the critical segment in programming and application development with μCs, as it is in line with what students should be able to compare in results and intended outcomes (P6.1), and subsequently examine and correct their thinking (P6.2) [39]. As Brennan and Resnick remark [43], "Things rarely (if ever) work just as imagined; it is critical for designers to develop strategies for dealing with – and anticipating – problems." The next section aims at demystifying the fundamental and essentially limited resources that are needed to exploit microcomputer technology within computer science education at an introductory level.

The μCT Concept: An Onion Learning Framework

Relative to the program code developed for a personal computer, the fundamental difference in developing microcomputer code consists in the IO interfacing with the outside world. While students within an introductory programming course make use of two standard IO units (that is, the keyboard and monitor) to interface with the outside environment, the students of a microcomputer-based programming class have an abundance of

nonstandard units to choose from. Thereby, the introductory examples are commonly arranged around IO interfacing practices. Moreover, while the software code is arranged around ready-to-use functions that control the computer units (e.g. the monitor), the corresponding code for a microcomputer controls the application-specific units through the device's IO pins. The latter are directed by read/write operations of specific memory locations, which is another feature encrypted in the conventional programming.

Simple IO operations shape the microcomputer's parallel interface to the outside world. However, a μC-based application usually incorporates several peripheral modules, a requirement that has a negative consequence to the (limited) pin resources of the μC chip. For that reason, the serial interfaces are of vital importance for the implementation of a microcomputer-based system. Due to this need, typical μC devices embed subsystems that support the implementation of a serial interface (commonly I2C, SPI, and UART) with the outside world. It is here noted that other popular subsystems can also be found inside of a μC chip, such as *analog-to-digital converter* (*ADC*), *pulse width modulation* (*PWM*), USB controllers, and so forth. However, the possibilities and limitations, as well as their configuration and control of such subsystems, differ from device to device.

While the μC programming and application development entails the knowledge of the device's architecture, the μCT should look beyond specific architectures. Rather, the learner should have a clear sense of the generalized architectural framework and operating principles of μC, in the direction of solving a problem with microcomputers. The future designer needs to be able to cross boundaries across diverse chip devices and architectures. Fortunately, the DIY culture (applied to μC technology) sustains this educational possibility with the abundant daughterboard systems, available in today's market. Figure 1.4 recommends on an onion framework toward learning μC programming and application development. If the language practices, common to μC and personal computer programming (e.g. control-flow statements, arrays, subscripts), are placed within the innermost layer then the outer layers are consecutively formed as follows:

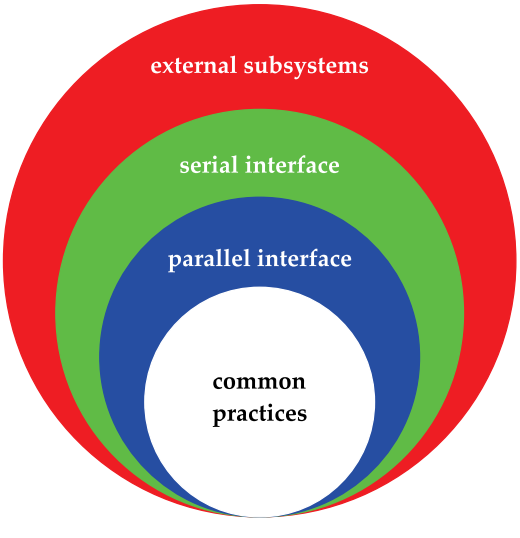

Figure 1.4 Onion learning framework for μC programming and application development.

i) ***Parallel interface:*** refers to the microcontroller's IO pin transactions as directed by the read/write operations of specific memory locations. This is foremost and perhaps the most important difference between programming a μC and a conventional personal computer, as it determines the hardware response according to the code execution. This category can enclose more advanced issues, such that the interrupt mechanisms;

ii) ***Serial interface:*** refers to the hardware interface between the microcontroller and the external peripheral devices. There are three different methods of implementing a serial interface. The most popular is the employment of a μC device with an embedded serial interface module (yet, this is an architecture-specific method). The second is the utilization of ready-to-use functions available within the employed compiler, which direct the regular μC pins to a serial transaction. The final (which is my personal choice because it supports low-level practices [28]) is in agreement with the building of custom-designed functions for this purpose. This alternative is strongly associated with another important concept in μC programming and application development; that is, the timing concept.

iii) ***External subsystems:*** refers to the selection of the appropriate peripheral system as well as the necessary transactions that will contribute to the building of a custom application. The external subsystems could be as simple as an ADC that is used to capture an analog signal and pass this information to the μC device, or a complex interface, such as the Ethernet which meant for outside-the-box communications [44]. It should be noted that the time constraints needed to learn the functional properties of a complex module, can considerably overload the amount of information one introductory class can cover [45].

"Transparent" Teaching Methods

The term transparency has been used to characterize particular pedagogical technologies in consideration of the possibility of being directly related to their function (e.g. a pencil) [46], which is opposed to protean feature of digital technologies (e.g. a computer) in regard to their usability in many different ways [47]. This term is hereafter borrowed to describe the well-known *pseudo-code* teaching method, addressed to illustrate the operating principles of a computer code. There is no standard format for the generation of a *pseudo-code* and no particular dependency on rules defined by a programming language. Yet, it is a dominant teaching approach for computer and microcomputer programming, though of low-range (compared to other approaches in μC education of extended-range such as, the familiar PBL approach [32]). To expand further the utilization of transparent teaching methods (of low-range) as they apply to the aforementioned analysis of μCT concept, we herein introduce the terms *pseudo-architecture*, *pseudo-timing diagram*, and *pseudo-hardware*.

The term *pseudo-architecture* is addressed to illustrate a simplified (and generalized) scheme of the inner workings of a μC. As in the *pseudo-code* approach, the representation depends on strategies determined by the instructor. My personal strategies (as they apply to 8-bit μCs [28]) are in agreement with a limited representation of the basic registers of the *central processing unit* (*CPU*), as well as a part of data memory (i.e. IO and regular registers)

and a part of program memory (depicting vectors, the location where the application code is stored, etc.) in a Von Neumann architectural representation for simplifying conceptualization. The *pseudo-architecture* is addressed to illustrate how the internal registers of the µC response to the execution of a *pseudo-code*. A *pseudo-timing* diagram is additionally addressed to initiate the timing concept, which is particularly important in µC programming and application development. It does not refer to the actual clocked processes that take place during the code execution, but rather to the fundamental processes decided by the instructor. A *pseudo-hardware* (referring to a simplified structure of the employed µC and possibly some peripheral units) can be addressed to provide a clear link between the firmware execution and the hardware response.

Figure 1.5 presents a *pseudo-code* example that blinks a *light-emitter diode* (*LED*), which is uploaded to the *pseudo-architecture* of a *pseudo-hardware* device. The *pseudo-timing diagram* in the upper left area of the figure represents the first six successive clock cycles that take place in the execution. The arrow depicts the current execution, which forces the LED to turn ON. This is a result of writing a logical one to the bit 5 of data (IO) register traced in *addr1* memory location. The *program counter* (*PC*) inside of the CPU points to the memory location of the subsequent pseudo-instruction to be executed (i.e. PC = 0x00A). The *accumulator* (*ACC*) register is addressed to illustrate the difference in the length of the

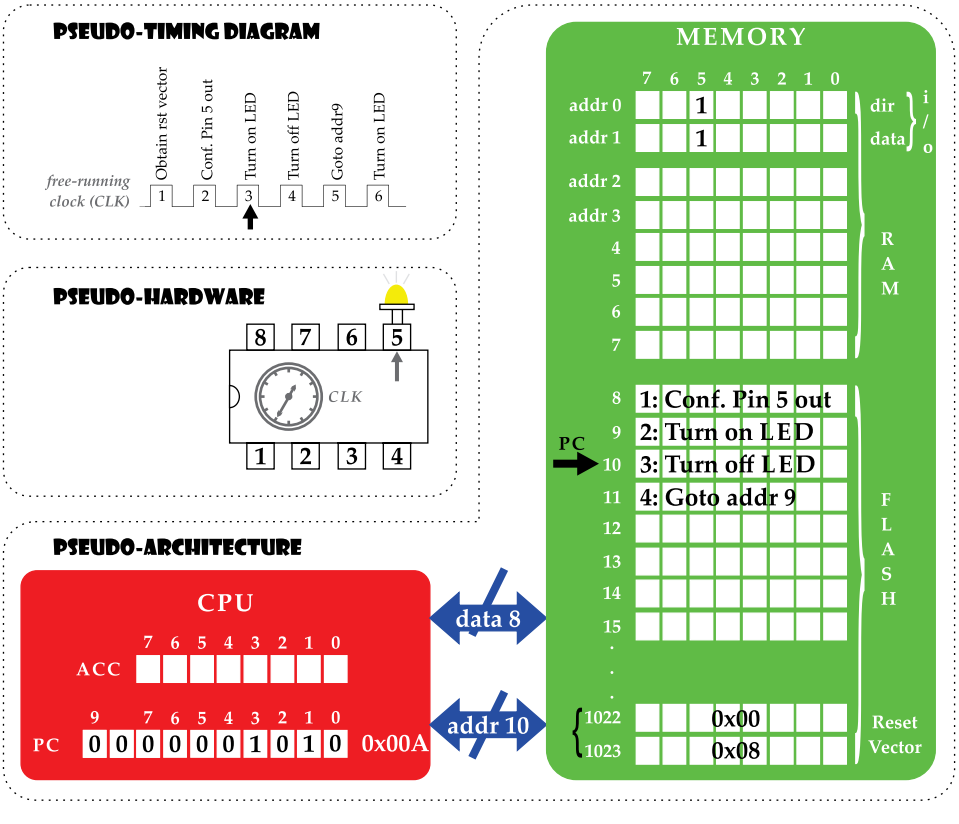

Figure 1.5 An example of pseudo-architecture, pseudo-timing diagram, and pseudo-hardware.

two CPU registers as the memory organization in the current μC is of 8-bit, while the available registers in overall memory cover a range greater than $2^8 = 255$. In detail, the available registers in overall memory are extended to $2^{10} = 1024$ memory locations; a value that is in agreement with the current length of the program counter.

It is worth noting that the underlying hardware mechanisms (as the ones depicted by Figure 1.5) are encrypted during an introductory computer programming course. However, those mechanisms are of particular importance for a course applying to μC programming as they settle the conditions for more advanced topics (e.g. interrupt mechanisms, bootloader in microcontroller's memory, a device reset caused by stack overflow, and much more). Many of these aspects arise from the fact that a microcomputer is of limited abilities, compared to a regular computer system. For instance, the developer will not deal with a situation that the generated code may not fit within the computer memory (which is not an impossible fact for a μC code). Complementary to the transparent (and low-range) *pseudo-code* teaching approach, the *pseudo-architecture* along with the *pseudo-teaching diagram,* and *pseudo-hardware* representations, could be the transparent teaching approaches that sustain the μCT in an introductory microcomputer programming and application development course.

The Impact of Microcontroller Technology on the Maker Industry

From the technological point of view, the μC device constitutes a single-chip computer, incorporating the following parts:

i) ***Central processor unit (CPU):*** that is, the device's processor core, which is responsible for executing the application code (where the *instruction execution* is performed in *sequential order*);
ii) ***Program memory:*** refers mainly to the type of memory that is used to hold the application code (aka *firmware*); that is, a *read-only memory* (*ROM*) the content of which can be updated by the user (i.e. the code developer);
iii) ***Data memory:*** refers to the *random-access memory* (*RAM*) which is used to hold the temporary data generated during the code execution, as well as to the IO RAM registers that interface with the outside world. The microcontroller's IO registers can be used to control the device's IO port pins. The latter can act as simple digital IO pins (for inputting/outputting a digital signal from/to the outside world), or as special function pins that are directed to an embedded subsystem inside the microcontroller device (e.g. an ADC, PWM); It is worth noting that a μC device may incorporate some or more subsystems, which are commonly found in embedded applications.

According to the aforementioned information, the *microcontroller* constitutes a *microcomputer,* which is optimized for control applications; as defined by the identical term of this technology. Microcontrollers constitute a popular type of *embedded computers*, nowadays, while this technology has been experiencing the widespread dissemination of *DIY* culture, the last decade. Today's wave of the *ready-to-use* and *stackable* boards, and the *shareable* – over the internet – libraries, renders feasible the rapid development of

microcontroller-based applications. Furthermore, the modern μC programming methods have managed to abstract the low-level tasks with the underlying hardware and, hence, have rendered this technology sensible for the inexperienced *makers*. Nowadays, microcontrollers constitute a low-cost and easily accessible technology that can be straightforwardly incorporated within any *makerspace*.

To understand the impact of microcontroller technology on the maker industry, it would be wise to follow the advancement of microcontroller programming and application development during the years of maturation. The advancement of this technology can be separated into two major eras, in consideration of the requisite time needed to learn, program, and develop a microcontroller-based application [48]. That is, the (i) *long-cycle* development era and (ii) *short-cycle* development era.

Hardware Advancement in μC Technology

A significant reduction of the requisite time needed to program and develop a microcontroller-based application can be reasonably supposed to have arisen from the advancement of the *nonvolatile* memory technology. The *nonvolatile* (contrary to the *volatile* type of) memory is the one that is used to hold the application code, as it retains the stored information when the power supply is removed. The previous generation of μCs where either of an early type of *electrically erasable programmable read-only memory* (*EEPROM*), or of *ultraviolet programmable read-only memory* (*UVPROM*).

Both types of memories required a separate board (regularly referred to as *programming board*) in order to upload new code in the μC. This demand arose from the fact that it was required a different circuit for reading and executing code from μC's memory, and a different circuit for writing new code to the μC's memory. The interface for uploading the updated code to the μC device was regularly based on the *recommended standard 232* (*RS-232*), while the programming board required an additional power supply unit to power the internal components of the *programming board* (Figure 1.6a). In addition, the UVPROM type of memory required an extra UVPROM eraser, where the quartz window on the top of the device allowed the μC's memory to be exposed to ultraviolet light in order to erase previous data before uploading the new code (Figure 1.6b). The common exposure time was approximately 20 minutes and, hence, the code developer could assess the functionality of an error-free code two to three times per hour. In most cases, the user should develop a custom-designed PCB that would hold the application circuit (like the example presented in Figure 1.7).

The aforementioned information reveals the considerable hardware involvement, time, and cost of the requisite occupation just to upload a firmware to the μC's memory (in the *long-cycle* development era). When the later-type of EEPROM memory made its appearance in μC market, it allowed the data to be erased and rewritten within the application circuit. This fact rendered unnecessary the extra *programming board* that was used for the firmware updates and promoted the, so-called, *in-system programming* (*ISP*) capability. In addition, the more recent *Flash* type of memory does not require the μC to be completely erased before rewritten (i.e. it can be read/written in blocks) and, hence, it promotes faster firmware updates that accelerate the debugging process.

The critical time during the μC programming and application development process reduced even more with the replacement of the RS-232 interface. The today's USB industry

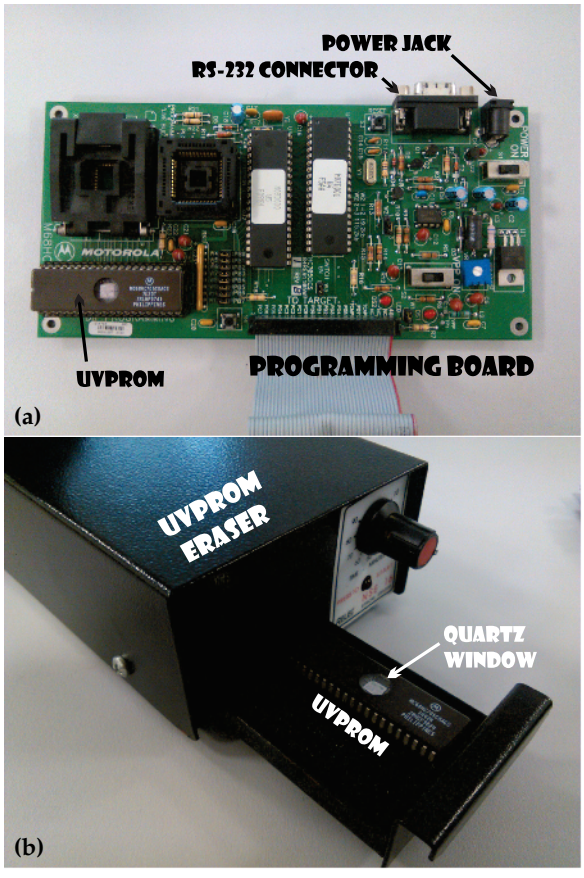

Figure 1.6 (a) µC programming board and (b) UVPROM eraser (from the *long-cycle* era).

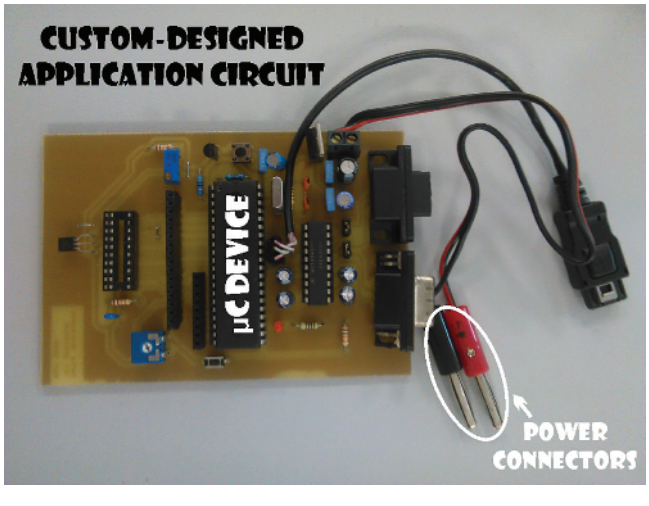

Figure 1.7 Custom-designed PCB of the µC's application circuit.

standard not only supports a high-speed communication between the μC and a personal computer, but also provides the ability to supply power to an external device. Hence, the modern microcontroller boards are commonly delivered with a USB connector, which is used to supply electric power, program the microcontroller, as well as to provide a communication interface of the μC board with the personal computer. All the above hardware-related improvements shape the *short-cycle* development era of the modern microcontroller-based applications.

Figure 1.8a depicts a typical microcontroller board of our age; that is, the classic Arduino Uno. Like similar boards on today's market, the Arduino Uno (on the left of Figure 1.8a) is preprogrammed with a *bootloader* code. The bootloader is referred to a small portion of code in μC's memory which provides the ability of updating the user-defined firmware code. With a single press of a button, in the corresponding software running on the user's personal computer, the bootloader receives the firmware information externally from the

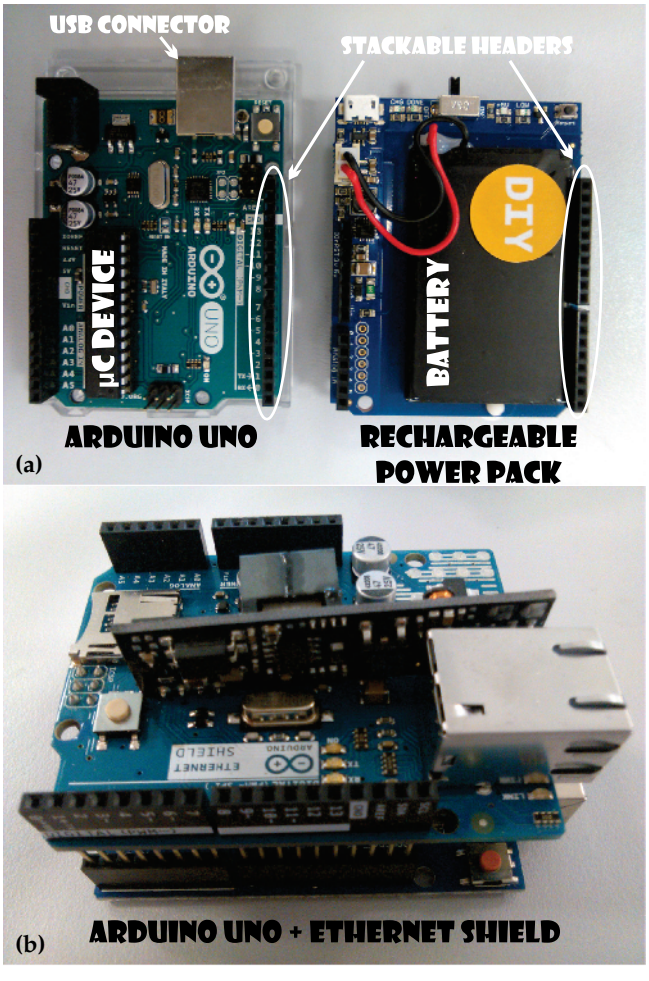

Figure 1.8 (a) Arduino uno board and powerpack shield; (b) Arduino Uno and Ethernet shield.

USB board, and uploads the new application code to the μC's memory. The stackable headers employed in such board systems, allow us to attach one or more daughterboards (aka *shields*) to the main motherboard and arrange the hardware part of a sophisticated microcontroller-based system (more or less) within a minute or so (i.e. in many cases there is no need for a custom-designed PCB in order to implement a μC-based application). For instance, on the right of Figure 1.8a a rechargeable powerpack can be attached to the Arduino Uno board and make available an autonomous μC-based system. Another example is depicted in Figure 1.8b where an Ethernet shield is connected on the top of the Arduino Uno motherboard, through which the μC can connect to the internet.

Software Advancement in μC Technology

The requisite time needed to develop the source code for a microcontroller device considerably reduced with the replacement of the *low level*, with a *higher level* of programming. The dominant μC programming method in the *long-cycle* development era was performed in assembly language. The assembly language constituted a suitable programming method at that time, as the code developer would normally spend considerable effort in simulations before uploading the updated firmware to the device (in order to evaluate its functionality). As mentioned earlier, the successive firmware updates were possible, merely two to three times per hour. Hence, the source code debugging process through simulations, was an unavoidable necessity, at that time.

The replacement of the assembly level with a higher level of programming, along with the hardware advancement related to the successive firmware updates in μC's memory, considerably accelerated the development process an error-free code. Due to the need of getting total control of the underlying hardware, the most popular μC programming method among engineers nowadays is the *embedded C/C++* programming. Other popular programming method of our age is the *MicroPython* interpreted[2] language, which is a *Python* variant optimized for μCs. Lately, there is a tendency of moving toward *visual programming languages (VPLs)*, which run on a web browser and are intended for beginners. An identical example of VPL for μCs, nowadays, is the *MakeCode,* which is used for the code development of the popular *BBC Micro:bit* board system.

The Impact of Arduino on the μC Community

When the *low-level* gave its place to a *higher-level* programming approach, the *embedded C* thrived on μC programming, and maintained a dominant position for many years. The *8-bit* architecture was the standard microcontroller technology of the time, and two vendors that held a substantial share of the world market were the (i) *PIC* microcontrollers of *Microchip Technology* and (ii) *AVR* devices of *Atmel Corporation*. If we try to make an informal classification of the two brands, it could be assumed that the PIC devices were the favorite development tool for *professional engineers*, while the AVRs were (more likely) the best

2 In *interpreted* programming languages, executions are implemented directly without previously compiling the source code into machine language (as it is performed in the *compiled* languages, such as in C/C++).

choice for the *novices* and *enthusiasts*. On one hand, the several embedded subsystems inside the PIC μCs were providing flexibility to the *development engineer (*upon the design of a new product). On the other hand, the availability of the freeware tools for the source-code development, were increasing the motivation of *novices* and *enthusiasts* to make a choice on the AVR μCs. Among other target processor families, the familiar and free *GNU compiler collection* (*GCC*) system supports the AVR architecture, as well.

Apart from these two noble microcontroller manufactures,[3] there were (and still are) many third-party partners providing software and hardware tools for the development of μC-based applications. Each vendor merchandise different development tools without compatibility among different corporations. It could be easily assumed that this is a fair politics to "encourage" users to keep supporting the hardware and software development tools that were originally selected. However, the diverse software and hardware tools give rise to a thriving Tower of Babel, in regard to the selection of the proper development apparatus for a custom-designed system.

Under these circumstances, the Arduino company made a clever move. They created a free-of-charge and particularly simplified *integrated development environment* (*IDE*) that was used to program the company's (rather limited) μC motherboards. The latter were delivered with a preloaded bootloader, which allowed firmware updates (through a USB cable) with merely the press of a button. In Figure 1.9 the corresponding "Upload" button is automatically highlighted in white, when the user scrolls the mouse pointer over it.

The Arduino's μC motherboards, such as the Arduino Uno, use a USB to Serial converter (the drivers of which are installed upon the Arduino IDE installation) and, hence, the only preparation the user should perform before pressing the "Upload" button is to select the

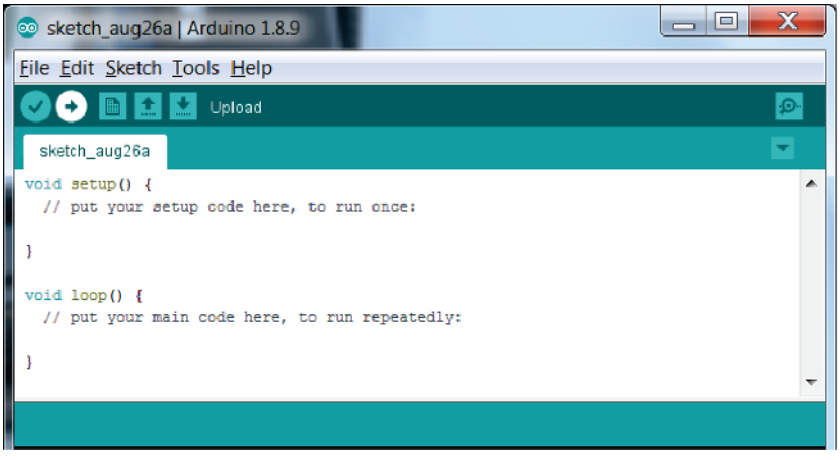

Figure 1.9 Uploading the new firmware code to μC's memory through the Arduino IDE. *Source:* Arduino Software.

3 The AVR devices acquired by Microchip Technology in 2016 and hence, PICs and AVRs now belong to the same corporation.

motherboard's name from the menu "Tools→Board:" (e.g. Arduino Uno). If not automatically selected, when the board is plugged into the USB, the user may also select the serial device of the board from the menu "Tools→Port:" (e.g. Port 3).

The Arduino company first focused on the AVR devices and was built over the GNU GCC/G++ compiler. What this means is that it could be still used by the stunts supporters of C/C++ implementation for the AVR architectures. Despite the "getting started" process, the Arduino IDE incorporates built-in functions and ready-to-run examples that accelerate the development procedure.

Other corporations that focus on the development of software platforms, also offer built-in functions, preprocessor commands and ready-to-run examples that promise a quick jump-start of a μC-based application. However, those software platforms attempt to incorporate a plethora of microcontroller devices in order to satisfy the needs of different designers and increase the potential range of the market. This decision makes the software platform more complicated, especially for the novice designer. The notion of designing with μCs constitutes a complicated and multifaceted area of study and, hence, minimizing the options available to the novice users helps them gain a more immediate access to the application development procedure.

The latter strategy of Arduino company is applied to another issue that is related with the barriers to entry μC programming. That is, the IO operations with the outside world. The simplest operation that can be performed by a microcontroller unit, in order to get access to the outside world, is to output a digital signal to one of the device's pins. Thus, the simplest example that regularly a novice user performs is to connect a LED to one of the μC pins and make it blink. While this example might be considered simple enough, the hard part for the novice learner is to make a clear link between the firmware and the hardware.

To this end, the Arduino boards are designed with a silkscreen alongside the PCB's stackable headers, which describes its header pins. In addition, the Arduino-specific libraries that control the μC pins admit definitions that are identical to naming of those headers pins, and not to the naming of μC pins. This way the user observes directly the hardware and is able to use those identical symbols and names during the code development process. Otherwise, the code developer should think at which pin of μC device is the header pin connected, and then write code that refers to that identical μC pin. For instance, to blink the LED connected to pin 13 of Arduino Uno board one would normally had to get access to pin 5 of μC's port B (Figure 1.10). With the built-in functions of Arduino, the code developer can turn *ON* the LED (connected to pin 13) with the simple syntax ***digitalWrite(13, HIGH)***; or turn the LED *OFF* using the command ***digitalWrite(13, LOW)***.

According to the aforementioned information, the μC programming and application development with Arduino integrates software and hardware tools jointly and specifically designed, so as to provide a quick jump-start and flexibility in the implementation of a μC-based project. Nowadays, the Arduino has become a viral technology, as it has created a wave of the *ready-to-use* boards and *shareable* (over the internet) libraries. Moreover, many third-party partners provide Arduino-compatible hardware tools that expand the design options available in μC-based systems. For all these reasons, this book is oriented toward the Arduino IDE platform using popular and contemporary Arduino-compatible boards.

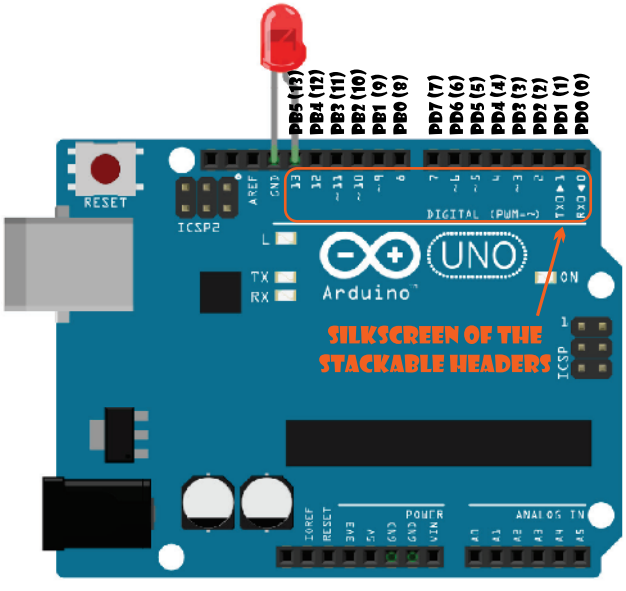

Figure 1.10 PCB silkscreen of the Arduino Uno stackable headers and the μC's digital IO pins.

In addition, the book attempts to explore creativity in μC product design with the utilization of contemporary sensor devices, as well as by the utilization of DIY culture in μC technology and the 3D printing technology of our age.

Where Is Creativity in Embedded Computing Devices Hidden?

To explore creativity in μC product design, we will consider the mean that paves the way for creativity and new solutions in one of the most popular embedded computing devices, nowadays. That is, the mobile computing devices, aka *smartphones*. The smartphones are gradually becoming powerful and complex embedded computing devices, while also giving rise to an opportunity for creativity and innovation.

Creativity in Mobile Computing Devices: Travel Light, Innovate Readily!

For all of us who have grown up in a period where *internet* was an unknown word and *gaming* a synonym to a bunch of kids kicking a ball in the neighborhood's street, the technological advancement came up as a revolution, which put extra weight on our back. Think of a person before leaving the house/apartment for a holiday trip and you will realize this extra weight is not just figure of speech. A few decades ago, one would normally check a bag full of portable gadgets for the trip (e.g. radio player, CD player along with some CDs, camera for shooting still pictures, camcorder for taking "high-quality" videos, extra memory for the cameras, alarm clock, handheld game console, as well as extra batteries and all the heavy chargers for its particular device). Nowadays, the same person would (in most of

cases) put the smartphone in one pocket and make a last check for the phone's charger (perhaps in his/her other pocket).

By the way of introduction, the sense of portability has dramatically changed the last decades. Handheld devices have been merged together to make our life easier where the smartphone constitutes, in simple words, the *all-in-one* device. A user may utilize the smartphone for everyday activities (connect to the social media, send an email, be navigated while driving the car), entertainment (play a game, read a book, listen to the music), sophisticated operations (i.e. record sport activities, be guided inside a museum, monitor personal heath, control home appliances), and sometimes perform or respond to a call. This handheld device seems to be the smartest movement of the market the last decades, as it gave users the opportunity to carry all the favorite gadgets without the extra weight. If one had to select a single handheld device before leaving the house/apartment, the mobile phone would most probably be the first choice. Apparently, the feeling of secure (i.e. to be able to call for help whenever it is needed) is above all other feelings (pleasure, entertainment, and so forth).

Due to the advancement of computing systems and the gradual cost reduction of the incorporated (hardware) electronic components over the years, mobile phones have been remodeled into powerful handheld computers available to everyone. It is thought that more than 2 billion smartphones are in use, nowadays [49]. Accordingly, smartphones lead the market and society in many different ways, but also pave the way for creativity and new solutions. The spark for creativity and innovation came up with the Android open-source platform marketed by Google, while the boost established by the release of App Inventor for Android (which rendered the mobile application development accessible to the non-computer science majors) [50]. But where is the creativity hidden in these small but powerful computing devices?

To comply with the multifaceted and increasing needs of humans (e.g. navigation, gaming), the design of mobile devices has reached to the point where smartphones and embedded computer systems look quite alike. Design and development of an embedded computer system incorporates some IO units, which allow users to interface with the computing device, where in the case of mobile phones this role is undertaken by the touchscreen. Moreover, an embedded application is commonly arranged around interfaces, sensors, and actuators. Leaving aside the actuators, smartphones are commonly delivered with Bluetooth wireless technology, which constitutes a practical interface for exchanging data among nearby devices (and of course, it could be used for the control of an actuator at distance). In addition, modern smartphone devices incorporate several sensors that render feasible the implementation of creative applications. The types of sensors incorporated in modern mobile phones seem to hold the key for creativity and innovation, and to give credit where credit is due, an important share of them has been determined by the constantly growing network of gamers.

Mobile gaming has been considerably advanced from the time of Nokia's Snake game and is currently in agreement with motion-sensing operations. Motion-sensing games (originated by Nintendo [51] and currently being found in abundant smartphones apps) obtain data from *micro-electro-mechanical systems* (*MEMS*) sensors and allow users to pilot aircrafts, steer cars, and so forth. While some people see mobile sensors as a valuable tool for game apps, others value the potential utilization of a powerful embedded computer

system with built-in MEMS sensors in research and development. Indoor navigation, gesture recognition, and virtual and augmented reality applications constitute examples that regularly apply to motion-sensing implementations. Such examples derive from data fusion of (i) the static and dynamic forces measured by 3-axis MEMS accelerometers, (ii) the angular velocity measured by 3-axis MEMS gyroscopes, (iii) the magnetic field measured by 3-axis MEMS magnetometers, and (iv) the barometric altitude determined by MEMS barometers.

Nowadays we have moved to the so called "combo" sensors that incorporate the functionality of two or more sensing elements (e.g. accelerometer and gyroscopes) into the same package, thereby minimizing the spatial requirements within the mobile device. Driven by the needs of smartphones, todays' "combo" solution are rearranged into a *system in package* (*SiP*) that incorporates an additional microcontroller device, addressed to deliver fused data from the built-in sensors. Hence, complementary to the minimization of the spatial requirements the smartphone can be relieved from complex processing operations, as well. This solution speeds up the mobile apps development process and therefore, it can be considered as being in agreement with today's maker culture in supporting creativity and innovation. Having recently announced the development of the world's smallest micro-computer (i.e. smaller than a grain of salt), IBM claims that such micro-computers (able to monitor, analyze, communicate, and even act on data) will be embedded in everyday devices within the next few years [52]. Thus, it would be wise to pay particular attention to the sensor devices when turning new and imaginative ideas (which incorporate embedded computers) into reality.

Communication with the Outside World: Sensors, Actuators, and Interfaces

Apart from the simple IO units (e.g. switches, LEDs), a microcontroller system is regularly arranged around sensors, actuators, and/or interfaces. According to the aforementioned analysis, the critical parts that give rise to an opportunity for creativity and innovation are, without doubt, the sensor devices. In the subsequent chapters of the book we will explore, in practice, how creative embedded computing may be implemented with the exploitation of the appropriate sensor devices.

Figure 1.11 illustrates a modern μC-based system which incorporates all the fundamental peripherals of a system, i.e. sensors, actuators, and interfaces. We address this example to also highlight the typical serial interfaces that are regularly used in microcontroller-based applications. That is, the UART, SPI, and I2C.

Apart from the four electric *actuators* (i.e. *DC motors*) of the drone, which are controlled either directly by a parallel interface (i.e. the μC's port pins) or indirectly through a DC motor driver board, the drone employs, at least, a *radio frequency* (*RF*) module. The latter is used to receive data from the remote controller (which also employs a similar or the same module) and it is regularly interfaced by the SPI protocol. The SPI is based on a synchronous type of serial communication. What this means is that the communication between the μC and the RF module is synchronized by a common clock line. It should be noted that each device on the SPI bus uses a different line to receive the incoming data, and a different line to transmit the outgoing information. If there is a need to connect more than one peripheral devices on the SPI bus, there is an additional line that is used to activate/deactivate

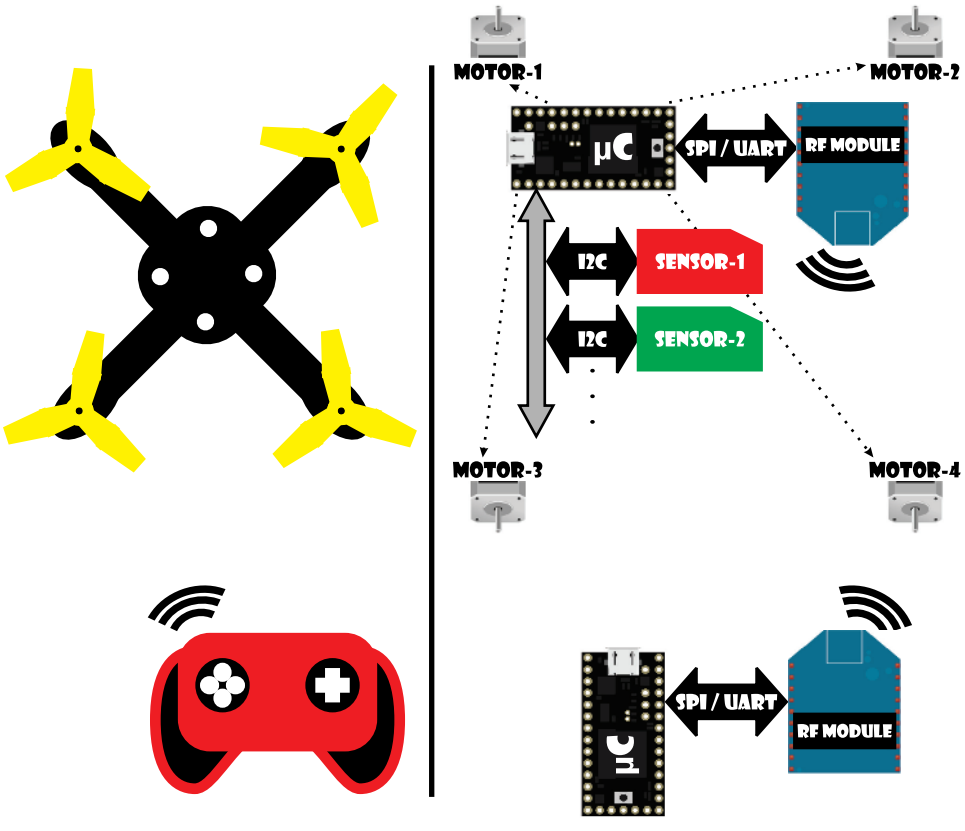

Figure 1.11 A drone is built around sensors, actuators, and interfaces.

each device (i.e. only one device should be activated in order to exchange information the system's microcontroller).

Some RF modules are interfaced by the UART protocol. The latter was the standard communication protocol between the μC and a personal computer, through the RS-232 serial port. This protocol is still very popular, nowadays, since many μC boards (such as, the Arduino Uno) incorporate a USB to UART converter in order provide backwards compatibility with older standard of serial communication. The UART applies to an asynchronous (and contrary to the SPI which uses *full-duplex*, the UART regularly addresses *half-duplex*[4]) communication. Since the μC and the peripheral device are synchronized by their own clock, they should be configured to the same clock rate.

The IC interface protocol is the most ordinary type of communication between the microcontroller and the system's sensor devices. The I2C applies also to a synchronous type of serial communication, but contrary to the SPI protocol, it applies to a half-duplex type of communication. This way, and along with the fact that the *data* line of the I2C

4 In full-duplex mode of communication data can flow simultaneously in both directions while in half-duplex; only one device at the time is able to transmit/receive data.

constitutes a bidirectional signal, this protocol uses only two signal lines (i.e. *clock* and *data*). The μC is the *master* device of the bus and each sensor device is regularly referred to as *slave* device. Each *slave* device on the I2C bus should be identified by a unique address, which is regularly a 7-bit address (i.e. $2^7 = 128$ different devices can be attached to the bus).

Conclusion

Embedded computer programming constitutes a multifaceted and interdisciplinary area of study, which falls within the disciplines of CS and EE. The programming of a regular computer encrypts the underlying operations with the hardware and, hence, the emphasis is placed on software practices (a learning approach appropriate for the discipline of CS). On the other hand, the programming and application development of an embedded computer entails more or less involvement with the hardware resources, and that involvement determines a likely position of the tutoring between the CS and EE disciplines. This chapter has recommended five types of embedded computers to help the instructor identify the educational possibilities in each category. Then, a *TPACK* analysis on the particularly interdisciplinary technology of microcontrollers was addressed.

μCs constitute a popular category of embedded computers nowadays. This technology has been experiencing, in the last decade, the widespread dissemination of DIY culture and has gradually shifted away from EE, hence, reducing the initial distance from CS discipline. It nowadays constitutes a low-cost and easily accessible technology that can be straightforwardly incorporated within a *makerspace*. Because of the *maker* movement in education, it would be wise to consider an upcoming turn to the educational research efforts related to microcomputer programming, at the expense of or complementary to the conventional computer programming courses. This attempt would raise questions such as: "What is the difference between programming a regular computer and a microcomputer? How could we arrange the content knowledge of a technical subject matter without too much focus on the specified technology?" The present chapter has addressed these issues and provided information that can be used as reference guide for further educational research in microcomputers, and embedded computers in general. Later in the chapter, the coined μCT term has explored the additional endeavor required to understand the capabilities of microcomputer technology as well as the thought processes involved in the solutions, carried out by microcomputer-based systems.

To understand the impact of μC technology on the maker industry, this chapter has also followed the advancement of μC programming and application development during the years of maturation. In consideration of the requisite time needed to learn, program, and develop a μC-based application, the author has identified the *long-* and *short-cycle* development eras. Then the chapter has explored how the modern μC tools provide quick jumpstart and flexibility in the implementation of a μC-based project. Finally, the author has considered the recent trends in sensor devices and how they pave the way for creativity and new solutions in embedded computing devices.