

1

Introduction

1.1 Motivation

The development of mobile software has often been addressed in a fashion that focuses on using some particular technologies. While this type of approach can be easily justified for the introduction of a mobile platform that is to be used as the basis of an implementation, long-term issues are harder to embed into such an introduction. Furthermore, as the number of mobile platforms has been increasing, it is becoming an option to aim at discussing the differences between workstation and embedded software and software that runs in mobile devices at a general rather than at an implementation-specific level. We believe that this leads to a longer lasting approach, which will not be outdated when a new version of some particular mobile platform is introduced, since the basic patterns and philosophy of a design are likely to remain the same even if the platform version changes.

Principally, the design of software that runs in a mobile device requires that developers combine the rules of thumb applicable in the embedded environment – memory awareness, turned on for an unlimited time, limited performance and resources in general, and security in the sense that the device should never malfunction to produce unanticipated costs or reveal confidential information even if the user behaves in an unanticipated fashion – with features that are needed in the workstation environment – modifiability and adaptability, run-time extensions, and rapid application development. For this combination, the designer must master both hardware-aware and application-level software, as well as the main principles that guide their design. In order to compose designs where all these requirements are satisfied, the designer is bound to use abstraction, which is the most powerful weapon for dealing with complexity.

1.1.1 *Leaking Abstractions*

Due to being such a powerful weapon for attacking software development, abstraction is also one of the most commonly used facilities in programming. Systems we

commonly use are full of abstractions, such as menus, databases, or file systems, to name a few. Moreover, we are good at managing abstractions we are familiar with, and know how they should be used. Therefore, the skill of programming in a certain environment implies that one recognizes the basic abstractions applied in the environment, and knows how the abstractions are intended to be used.

Unfortunately, abstractions are not problem-free. In particular, problems are imminent when we face a new application domain or environment, such as mobile devices. Commonly used abstractions of programming may no longer be solid but they can start to leak. We will study this phenomenon in more detail in the following.

In principle, as argued by several authors, including Gannon et al. (1981) and Gabriel (1989) for instance, the user of an abstraction can overlook the details of the underlying implementation. In practice, however, when composing programs, details of the underlying hardware and underlying infrastructure software used as the implementation technique of a certain abstraction sometimes become visible to the software developer or even to the user of the system. We will call this leaking abstraction.¹ However, without knowing the implementation, it is difficult to understand what happens when the program is executed and a sudden downgrade of performance occurs, for instance. This makes the design more difficult, as revealing the implementation can take alarming forms.

As a sample leaking abstraction, we next consider null-terminated strings used in the C programming language, for instance. The following procedure can be used to concatenate two such strings:

```
char * strcat(char * c1, char * c2)
{
    int i, j;

    while(i = 0; 0 != c1[i]; i++);
    while(j = 0; 0 != c2[j]; j++, i++) c1[i] = c2[j];
    c1[i+1] = 0;
    return c1;
}
```

The logic of the operation is that first, we browse all the characters of string `c1`, and then copy all the characters of string `c2` to its end. Moreover, it is assumed that `c1` is large enough to host also the characters of `c2`, which is not explicitly expressed in the procedure but is an obvious built-in assumption.

From the functional viewpoint, using this kind of an operation appears perfect. However, from the practical viewpoint, the function is far from perfect, as in some cases the implementation of strings becomes visible to the user. A problem is that if we have to carry out one million concatenations to the same string, we

¹ The term ‘leaking abstraction’ has been used in this meaning at least by Joel Spolsky (2004). Also the example we use to demonstrate such abstractions originates from the same source.

will unnecessarily go through the same characters all over again as longer and longer strings adopt the roles of c_1 and c_2 . While the execution would result in the correct outcome, the time needed for completing the execution would be considerably extended. By observing this from a completed, running system, one might be surprised since certain inputs would be slow to process, but by looking at the actual design, one would immediately learn the obvious problem of this implementation.

All non-trivial abstractions can be argued to leak to at least some extent (Spolsky 2004). For instance, let us consider the TCP/IP protocol that provides an abstraction of reliable communication; if the underlying communication infrastructure is terminally broken, there is no way the protocol can act in accordance to expectations. Similarly, although the SQL language is a powerful yet simple way to define database queries, some queries can (and usually should) be optimized by taking into account what takes place at the level of the implementation.

Fundamentally, programming languages which are commonly available in the mobile setting, such as C, C++, and Java, and their run-time infrastructures are also non-trivial abstractions. Therefore, they also have the potential to leak. In many cases, leaking abstractions of programming languages and infrastructures that are executed in mobile devices lead to problems in managing resources, memory consumption, and performance. Therefore, in order to compose programs where potentially leaking abstractions form a minimal problem, the programmer must have experience of working in a certain environment to create appropriate designs. Since expecting that all developers have experience is unrealistic, infrastructures have been introduced where the most obvious traps will be automatically treated, as well as tutorials and coding standards that aim at preventing the most obvious problems associated with leaking abstractions.

As already mentioned, mobile devices are restricted in terms of available resources. Therefore, in order to cope with leaking abstractions related to resources of the device, implied by the used programming languages and execution environments, the designer should understand what lies beneath the surface, because otherwise it is easy to use facilities of the language that are not well suited for such a restricted environment.

1.1.2 Allocation Responsibility

Programming systems in mobile devices can treat complexity in two fundamentally different ways. On the one hand, the responsibility can be given to the programmer, who then takes actions in order to manage resources, such as memory, disk space, or communication bandwidth. On the other hand, software infrastructure can be defined for handling the resources without revealing the details to the programmer, thus automating resource management from the programmer perspective. The above strategies can be considered as white-box and black-box resource management approaches in the sense that white-box resource management is visible to the

developer in full, whereas black-box resource management hides its details from the programmer and aims at an automatic deallocation at an appropriate moment.

Programmer responsibility. Fundamentally, making programmers responsible for resource allocation results in a white-box approach to resource management that is relying on programmers who are able to carry out designs in a fashion where leaking of abstractions is not possible, or, at the very least, leaking is controlled by them. An obvious language where leaking of abstractions can be a problem in the mobile environment is C++, as in many cases features of the language require thorough knowledge of the underlying facilities and implementation techniques. Implementing programmer responsibility for potentially leaking abstractions can take many forms. On the one hand, one can define a coding standard that explains why certain types of designs should not be used, or are considered antipatterns, i.e., commonly applied solutions that bear some fundamental, well-known handicap (Brown et al. 1998). On the other hand, one can introduce a coding standard that defines design guidelines for managing cases where leaking of abstractions is considered most harmful or likely. Moreover, in addition to knowing the guidelines, the programmer should also understand what kinds of problems the guidelines solve and why. This usually calls for understanding of what happens at compilation and how run-time infrastructure works.

Infrastructure responsibility. A black-box approach to resource management means that the underlying programming infrastructure is supposed to liberate the developer from considering potentially leaking abstractions. Examples of environments that operate in this way include Java and C#, which both can be used for programming mobile devices. However, in practice the developer is still able to use the properties of the infrastructure better, provided that she knows how the infrastructure works and takes this into account when composing a design. For instance, being able to compose designs that do not overly complicate the work of a garbage collector in a virtual machine environment is helpful, as garbage collecting can seemingly stop the execution of a program for a short period of time in certain virtual machine environments. Thus, while the black-box approach hides resource management from the developer and the user, this abstraction leaks when garbage collection is performed, as its side-effects may become observable by the user.

To summarize, no matter whether the programmer or the infrastructure manages resources, the way programs are designed and written has an effect on their performance and resource consumption. In the following, we discuss the most common hardware-related issues that the developer is exposed to when designing applications for mobile devices.

1.2 Commonly Used Hardware and Software

To summarize the above discussion, programming languages and their run-time infrastructures can be considered as at least potentially leaking abstractions. Such leaks mean that the properties of hardware and lower-level software are revealed to

an application programmer in some cases. Therefore, understanding their basics is a prerequisite for considering how applications should be designed for the mobile environment.

In the following, we give an overview to commonly used hardware and software facilities inside a mobile device, whose restrictions can be considered as the main technical contributors of mobility (Satyanarayanan 1997). The subsections address hardware, operating system concepts, application software, and the stack of software components that often forms the run-time environment of a mobile device.

1.2.1 Computing Hardware

The computing hardware of mobile devices can be expected to become more and more standardized. One important driver for this is the need to integrate all the subsystems into one chip. This saves development costs as well as energy consumption of the completed devices. Figure 1.1 illustrates the main elements that are significant within the scope of this presentation.

Processors and Accelerators

Fundamentally, a computer is a system that executes a program stored in memory. The processor loads instructions out of which the program is composed, and performs the tasks indicated by them. Instructions are low-level commands that discuss the execution in terms of hardware available in the system. For instance, loading the contents of a memory location to a processor's internal memory location, so-called register, adding the contents of two registers and storing the result in a third, and

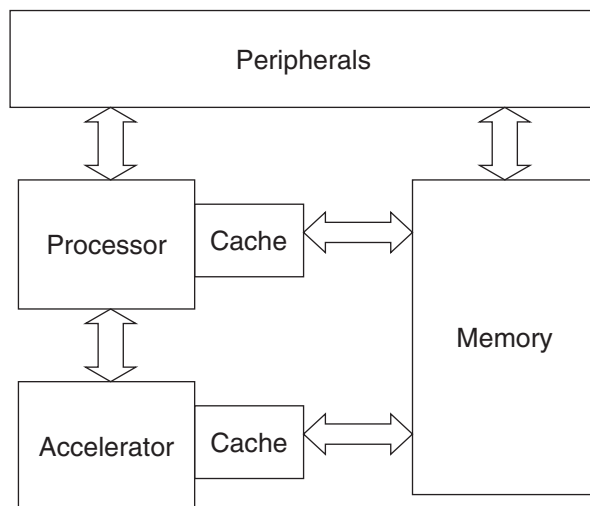


Figure 1.1 Commonly used hardware

```
int factor9() {
    int index, result = 1;
    for (index = 1; index < 10; index++)
    {
        result = result * index;
    }
    return result;
}
```

Figure 1.2 Sample source code

storing the value in a register to a particular location in memory, are commonly used instructions.

Processors have different instruction sets, i.e., primitive operations that processors are capable of executing, and their properties can vary. Probably the most commonly used main processor² in mobile devices follows ARM (Acorn RISC machine) design (Furber 2000). The ARM design is based on a 32-bit RISC³ processor that is produced by several vendors, and which has been integrated into many more complex pieces of hardware, such as OMAP architecture by Texas Instruments. On the software side, for instance Symbian OS runs on top of ARM, although also other processor alternatives have been considered in the design of the operating system.

As an example, consider the following. The piece of code given in Figure 1.2 computes the factor of 9. When it is fed to a compiler, which in this case is a GCC compiler for Symbian OS, the resulting assembly output is as listed in Figure 1.3. Similarly to most, if not all, assemblers, the output includes individual references to locations in memory and to registers, as well as load and store instructions, which form the low-level representation of any program.

In addition to the main processor that is responsible for controlling the device as a whole, many mobile devices include different types of accelerators and auxiliary processors, which in terms of hardware can be considered similar to other processors, but they play a different role in the final system. They are used for coding and decoding of radio transmissions, and, more recently, to enable more sophisticated features such as three-dimensional graphics, for instance. Moreover, in some cases a proprietary phone implementation can be extended with an application processor that is to be used by additional applications without risking the functions of the proprietary phone. Further possible processors include an access processor (or a modem) that is dedicated for executing routines associated with telecommunications.

² In the terminology assumed here, the main processor is the processor that controls other parts of the system.

³ RISC stands for Reduced Instruction Set Computer, where only a restricted number of relatively simple and quickly executable instructions are offered. In contrast, CISC, Complex Instruction Set Computer, refers to a more complex instruction set where also the execution times of different instructions can vary considerably.

```
@ Generated by gcc 2.9-psion-98r2 (Symbian build 540) for ARM/pe
.file "test.cpp"
.gcc2_compiled.:
.text
.align 0
.global factor9__Fv
factor9__Fv:
@ args = 0, pretend = 0, frame = 8
@ frame_needed = 1, current_function_anonymous_args = 0
mov ip, sp
stmfd sp!, {fp, ip, lr, pc}
sub fp, ip, #4
sub sp, sp, #8
mov r3, #1
str r3, [fp, #-20]
mov r3, #1
str r3, [fp, #-16]
.L2:
ldr r3, [fp, #-16]
cmp r3, #9
ble .L5
b .L3
.L5:
ldr r3, [fp, #-20]
ldr r2, [fp, #-16]
mul r3, r2, r3
str r3, [fp, #-20]
.L4:
ldr r3, [fp, #-16]
add r2, r3, #1
str r2, [fp, #-16]
b .L2
.L3:
ldr r3, [fp, #-20]
mov r0, r3
b .L1
.L1:
ldmea fp, {fp, sp, lr}
bx lr
```

Figure 1.3 Sample ARM code

Accelerators can be implemented using two different mechanisms, which are using multi-purpose hardware, such as a digital signal processor (DSP), and single-purpose hardware. Using the former is a more general design choice, as the hardware can be used for several tasks, whereas single-purpose hardware can only be used for the task it is designed for. However, a DSP is usually a bigger design block to be fitted in the device, and it requires more energy to run than a single-purpose piece of hardware. Still, in general, DSPs have superior power consumption to

performance rate over general purpose processors in tasks that are well-suited for them. As an example, a study has shown that a typical signal processing task on a RISC machine (StrongARM, ARM9E) requires three times as many cycles as a C55x DSP while consuming more than twice the power (Chaoui et al. 2002). The downsides of DSPs are being solved by introducing more elaborated integration schemes for off-the-shelf hardware. Such integrated systems typically include a main processor, auxiliary DSP, memory, and additional interfaces. While systems exist where everything is integrated into a single chip, it is common that devices with new features include several chips, each of which is dedicated for a certain purpose.

A continuous trend is that an increasing number of features and improved integration techniques lead to more complex designs in the sense of architecture. In addition, many hardware and platform manufacturers have been emphasizing mechanisms related to power management, which also bear an effect on the complexity of the final design. Moreover, such features are becoming more important, as ready and use times of mobile devices are meaningful properties for the consumer.

When more and more advanced features that require more processing power are introduced, clock frequency of the main processor must be increased. Unfortunately, this leads to problems due to heat generation. Therefore, one can consider that the current approach, where there is a master processor and a small number of auxiliaries, may need to be reconsidered. As a solution, two contradicting approaches have been proposed. One is symmetric multiprocessing, which injects the complexity of programming to the operating system, and the other is the introduction of more and more specialized auxiliaries that manage their internal executions themselves. Presented in Figure 1.4, both approaches can be justified with solid arguments, as discussed in the following.

Symmetric multiprocessing (SMP). A system based on symmetric multiprocessing consists of a number of similar processors that usually work in intimate connection. A common implementation is to hide them in the same operating system core, which allows them to balance load between each other. This scheme gives an opportunity to save energy by shutting down some of the processors when the load is low, and keeping all processors active when processor-intensive tasks are being performed. Moreover, as all the processors can be hidden behind the same operating system interface and have similar characteristics, a programmer can be offered an abstraction that can be conveniently used. On the downside, multiprocessing can be a relatively complex solution, when assuming that fundamentally the device can actually be a mobile phone, which by nature is a very specialized system. Moreover, the adequateness of the processor-level granularity as the basis of energy management can be questioned.

Asymmetric multiprocessing. The other alternative, the introduction of multiple specialized pieces of hardware, also offers an increased performance. Furthermore, the design of individual pieces of equipment can be eased, as they all perform some particular tasks. However, allocating tasks to different pieces of equipment cannot be implemented in a straightforward fashion as with symmetric multiprocessing.

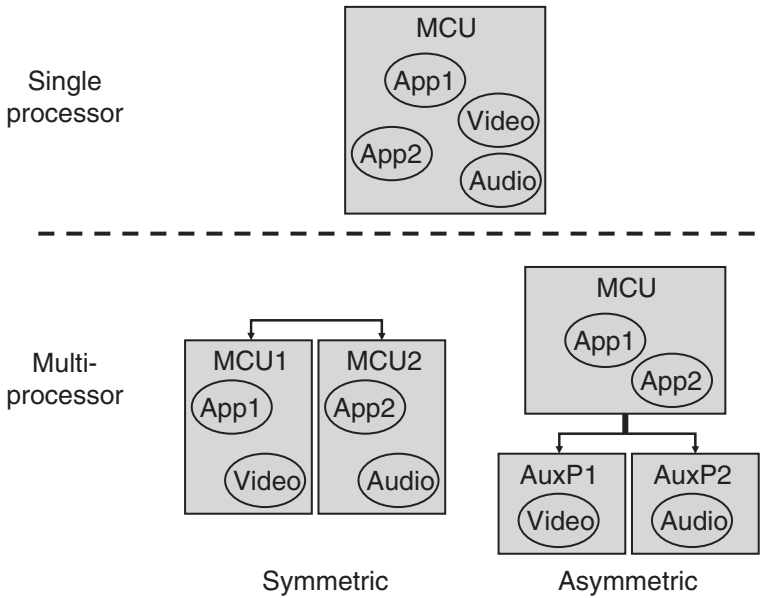


Figure 1.4 Symmetric and asymmetric multiprocessor schemes

Instead, the programmer may be forced to participate in the allocation for adequate results. As a consequence, different devices relying on different sets of auxiliary equipment must be supported. Moreover, some devices may implement some equipment with additional software running on the main processor. One possible result for this is to introduce abstractions that hide the complexity of hardware configuration in a standard fashion. Then, hardware support can be used whenever it is available, and software emulation will be used in devices that do not implement hardware acceleration. Still, a considerable development effort must be invested in the definition of interfaces that can be used with different implementations. Furthermore, standardization is required to enable interoperability. At present, the use of accelerators can be seen as a step in this direction.

Memory and Related Hardware

In current mobile devices, memory is usually internally constructed so that 32-bit memory words are used. Each word can be further decomposed to 4 bytes of 8 bits. In most cases, words are the main elements of memory in the sense that even if less than a word would be enough for a certain variable, a full word is still allocated in practice for performance reasons. There is, however, a possibility to allocate several variables to the same word, for instance. Unfortunately, this may result in degraded performance, as it is usually faster to access memory when respecting memory word boundaries. Respecting the word boundary is commonly referred to as (word) alignment.

Several types of memory are used. First, there is RAM (random access memory), which is used during the execution of programs for storing the loaded programs, and the state of the execution and variables related to it. This type of memory can be read and written. The typical amount of RAM in a mobile device has been increasing rapidly, with typical figures reaching up to 64 or even 128 Mb. Different types of RAM can be considered, including static RAM (SRAM) and dynamic RAM (DRAM). SRAM preserves its state but is unfortunately usually expensive, and it is commonly used only in memory that is to be accessed quickly, such as cache, which can be considered as an intermediate storage used for storing memory locations that the processor frequently accesses. DRAM, on the other hand, is based on transistors requiring constant attention from the rest of the system to preserve their state, which consumes some energy. DRAM is commonly used in the mobile environment, where probably the most common implementations rely on SDRAM (static DRAM). A benefit of this type of memory is that it can be run using the same clock speed as the processor.

Second, there is ROM (read only memory), which can be read but not rewritten. For instance, programs that are permanently stored in the device can be located in ROM. For execution, programs are usually first loaded to RAM for execution, although depending on the used chip set it is sometimes possible to execute programs directly from ROM using so-called in-place execution. In a mobile device, the amount of ROM can be 64 Mb or more, although flash memory, discussed in the following, can also be used for a similar role.

Third, many devices also contain permanent storage, although ROM and RAM would be sufficient for many purposes. The rationale is that if the battery is removed, it is helpful that the data stored in the device remains unaltered. Permanent storage can be implemented in terms of a hard drive or as flash memory which maintain their information even if power is switched off. Obviously, accessing disk is at least a magnitude slower than accessing RAM or ROM. Also physical characteristics of hard drives have encouraged the use of flash memory, as it is not as prone to mechanical failures.

As already mentioned, flash memory is commonly used in mobile devices. Accessing flash memory can be implemented such that it is relatively fast to read from memory, but writing is usually slow. The reason is that it is possible to turn single bits from 1 to 0, but turning 0 to 1 can only be performed in groups of 64 kb for instance, depending on the hardware implementation. As a result, even a small change in a file can result in a complete rewrite of the whole file. Two types of flash memory are to be considered, NOR and NAND flash memories. NOR flash can be used as direct memory space, and as ROM or RAM. The latter is different from SDRAM in the sense that while an access to flash-based RAM can be slower, SDRAM requires constant attention of the processor to maintain the memory active, which consumes energy. In contrast to NOR flash, NAND flash behaves analogously to a hard disk, and requires loading of code to memory before running it. In comparison to other types of memory and hard disk, an additional restriction

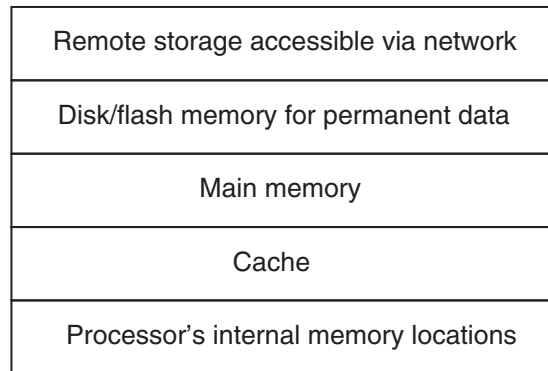


Figure 1.5 Memory hierarchy

is that only a very limited number of writes, say 100 000–1 000 000 rewrites, can be performed on flash due to its exhaustion. Similarly to hard disks, the use of flash requires the management of memory usage, which can consume a considerable amount of memory in a large flash file system (Chang and Kuo 2004).

The memory types discussed above create a hierarchy of memories (Figure 1.5), where the size of the memory and access times vary; at the bottom of the hierarchy (processor's internal registers or even cache) access is rapid and can be calculated in some nanoseconds, but only a limited amount of such storage space is offered. In contrast, accessing main memory takes place on the order of tens of nanoseconds, and accessing a value in disk is even slower, up to the order of tens of milliseconds. Furthermore, using some memory available in the network is slow, but a virtually unrestricted amount of memory can be offered.

In addition to the memory hierarchy, there is a special piece of hardware that is associated with memory, although it is not directly included in memory. The memory management unit (MMU, omitted from the figure for simplicity) is more conveniently implemented within the processor chip. The purpose of this piece of hardware is to enable using free locations in the memory for programs, disregarding their physical location. The MMU then manipulates the memory shown to the processor such that the memory image is simplified in the sense that programs usually appear to be in continuous locations in the memory even if they are distributed in the memory. Similarly, programs can be located in different parts of the memory during different execution instances of programs. MMU is also used for implementing memory protection enabled for processes, which we will discuss later in this chapter. Furthermore, MMU can also be used for implementing virtual memory, where some parts of the system are saved to disk in terms of memory pages when additional memory is needed, and when the pages are used again, they can be restored from the disk. As flash memory is not very convenient for such a purpose due to slow write operations and restricted number of write accesses, this scheme is not commonly implemented in mobile devices. However, the ability to freely

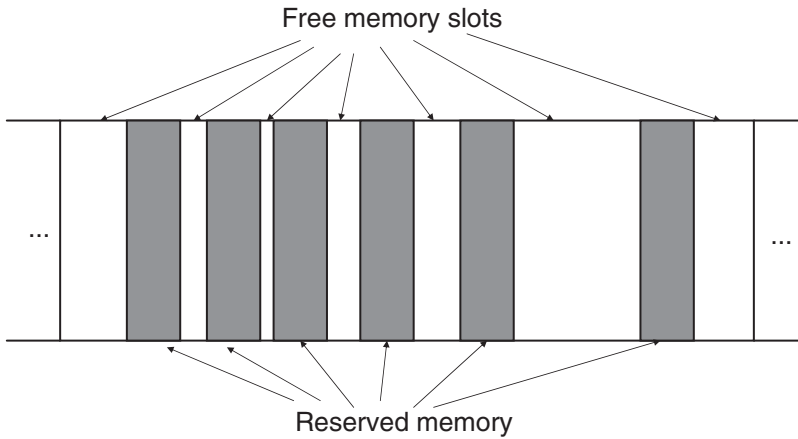


Figure 1.6 Fragmentation

locate programs in different parts of the memory as well as memory protection are appreciated, and therefore many mobile devices include an MMU. Still, there is no technical barrier to implementing a full virtual memory system.

Finally, two phenomena commonly associated with memory and its use are to be considered when composing programs for mobile devices. These are fragmentation and garbaging. The former means that memory is reserved such that some free memory remains unused between reserved blocks of memory (Figure 1.6). While fragmentation can in principle be handled by conjoining available free slices of memory to a bigger memory area, managing the lists of free and reserved slices can become a burden for the operating system, and superfluously consume precious memory and processing time. The latter refers to a situation where some data allocated to the memory can no longer be deallocated by the program that has allocated them, potentially reserving the memory locations until the execution of the program terminates, or, even worse, until the system is shut down.

Subsystems

In addition to the hardware described above, several types of auxiliary subsystems can be available, that require support from the hardware:

- Bluetooth is a short-range radio link that is used for cable replacement.
- Radio interface is used for communicating with the mobile network. Several bandwidths and protocols can be used, including for instance GSM, GPRS, WCDMA, and WLAN, to name a few.
- Keyboard (or touch screen) allows the user to input data and commands to the device.
- Screen enables the user to read the information stored inside the device.

- Additional memory for switching data between mobile devices using a memory stick that can be attached to a USB port or some other interface.
- Battery interface is used for managing the status of the energy source.

The above list is by no means exhaustive, and additional hardware is easy to imagine. In principle, anything that can be connected to a mobile device can form a new subsystem. Moreover, whether the connection is permanent or dynamic via a local area radio link, for instance, is becoming less important, as the capacity of connections is becoming comparable due to the increased coverage and bandwidth. This results in an option to use implementations where a number of devices cooperate to perform some higher-level tasks. At a lower level of abstraction, implementing this requires support from hardware in terms of I/O interfaces or the use of universal asynchronous receiver transmitters (UART), for instance.

In general, the characteristics of different subsystems are not directly visible to the application software running in the device as such but via different types of programming interfaces (or APIs, application programming interface), which makes it possible to run the same software even if the hardware is upgraded or modified in the design of a different type of device, which is common in the development of product families. However, lowest-level software must be adapted in order to benefit from the new hardware.

1.2.2 Low-Level Software Infrastructure

Kernel forms the core of an operating system. Fundamentally, it is a machine that is designed to manage the relationship of the underlying hardware and software that uses its resources.

For accessing hardware, kernels usually implement special modules called device drivers that can be used for creating a connection to the underlying subsystems and communicating with external equipment. The usual way to implement such drivers is that they send a request to the external subsystem, and the subsystem is responsible for serving the request. When the request is served, the subsystem responds with an interrupt that the kernel will acknowledge and serve with a corresponding interrupt handler, which contains the procedure for managing interrupts. Device drivers can be implemented in a layered fashion, where a physical layer handles the details of the actual hardware. A logical level can then be introduced for handling the parts that are common for all similar subsystems.

When considering the connection between the kernel and other software, the resources of the former are a necessity to create applications. Commonly used terms are processes and threads, which we will address in the following.

Processes can be considered as the units of resource reservation. This allows designs where the different resources allocated by programs are kept in isolation from one another. This, however, requires support from the hardware in the form of an MMU, which is usually included in processors used in mobile devices that

can be extended with new applications. For devices that do not allow this and only include proprietary software, this may not be an option; instead, all processes can run in the same memory space, when they can interfere with each other's execution. Examples of resources that are managed by processes include memory in particular. In addition, threads that are run within the memory space of the process are resources owned by the process. In systems where threads could not be created separately, they were commonly associated with each other.

Threads can be taken as units of execution. Each thread can be considered as a conventional program that has a control flow, and some piece of code it executes. Threads belong to processes, and they can share resources and data structures with other threads inside the same process. When the thread that is being executed is changed to another by a special part of the kernel, the scheduler, the event is called a context switch. If the operating system can force a context switch even if the thread is not ready for it, the scheduler is called pre-emptive, and if the thread must explicitly pause its execution, the scheduler is referred to as non-pre-emptive. A pre-emptive scheduling policy is more flexible, but its performance ratio is usually recommended to be kept at maximum 70% as otherwise some operations that are important for the user can be delayed in favor of other operations. Moreover, context switching can also be executed repeatedly, if the performance ratio is increased. In contrast, for a non-pre-emptive scheduler, a fixed order of executions is usually defined, which is less flexible but whose performance ratio can be close to 100%. Two types of context switches can be considered; one takes place when a thread is changed to another but the process hosting the threads remains the same, and the other when also the hosting process is altered. The former is usually a lighter operation, but even this operation can be considered expensive in the sense of performance. The reason is that all the work performed for pausing one thread and selecting and starting the other is overhead.

1.2.3 Run-Time Infrastructure

In this subsection, we connect run-time infrastructure of programming languages used in the mobile setting to the facilities of the underlying hardware and operating system software.

Allocating Memory for Programs and Variables

Perhaps the most straightforward consumers of memory are constants, which can usually be saved in ROM. This saves some valuable RAM for other use.

For variables, two locations can be imagined. When a program is being run, its modifiable data must be stored in RAM, as the program can constantly alter the data. In contrast, when the execution is completed (or interrupted) data can be saved to disk for further use in some later execution. Saving data to disk is not simple, however. Disk (or flash memory) access can be slow, which can be problematic for the user. Furthermore, it can be difficult for the user to realize when all data has been saved and it is safe to turn off the device. Therefore, the design of programs

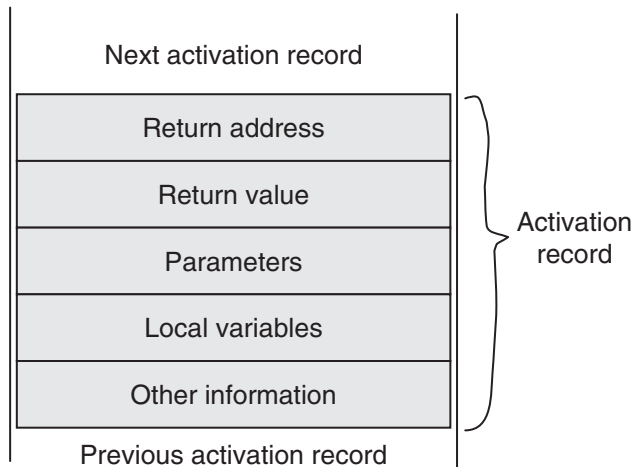


Figure 1.7 Activation record in a stack

should ensure timely saving of data in a fashion that is straightforward from the user perspective. In many mobile devices, this has been implemented using a practice where dialogs imitate transactions of database systems, offering for instance ‘Done’ selection for committing to the transaction in a fashion that is clear for the user.

Inside RAM, two fundamentally different locations can be used for storing variables, execution stack and heap. Execution stack is a memory structure that manages the control flow of a thread. Every method call made by a thread results in an activation record or a stack frame associated with the stack. The structure of an activation record is illustrated in Figure 1.7. This data structure includes information related to the management of the control flow, such as where to return once the execution of the called method is finished and what method made the call.⁴ In addition, each activation record contains method parameters, the return value (if applicable), and variables that are local to a method, which in some systems are called automatic variables that are defined within the scope of the method. As the stack is intimately associated with the thread using it, the execution stack is usually local to a thread. In contrast to stack, which is structured in accordance to the execution of a program in terms of activation records, heap is an unstructured memory pool from which threads can allocate memory. Unlike memory allocated from stack, where the execution flow manages memory consumption, memory allocated from the heap is under the responsibility of the programmer. The accumulation of heap-allocated memory areas that are no longer accessible due to a programming error for instance is a common cause of garbaging; then, a programmer allocates memory, but never deallocates it.

⁴ Exact details of activation records differ slightly in different systems. However, the principal idea remains the same.

The fashion in which the programmer composes programs determines where data structures are allocated. As an example on memory allocation from stack and from heap, consider the allocation of the following data structure:

```
struct Sample {  
    int    i;  
    char   c;  
};
```

This data structure is most likely allocated to memory in a fashion where each variable takes a different memory location in terms of words, resulting in the consumption of two 32-bit memory words, one for *i* and the other for *c*. While the actual size of the memory allocation is usually handled by the compiler and associated run-time environment, its location in the program defines how data structures are partitioned between the stack and the heap. Automatic variables, i.e., those that are local to a method or procedure, are allocated from stack, and explicitly allocated data structures, using, say, `malloc` or `new`, are placed in the heap. Figure 1.8 represents two different ways to locate the above data structure to memory using stack and heap as the locations. As already discussed, the location that is allocated for a data structure affects the way memory is to be managed; stack-based variables are automatically created as the execution advances whereas heap-based variables require explicit allocation and deletion, assuming that no special infrastructure for garbage collection is introduced. Unfortunately, in some cases compilers perform

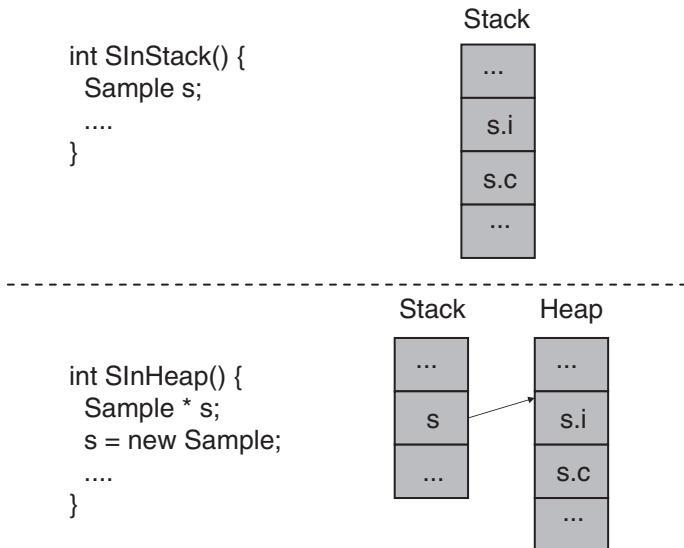


Figure 1.8 Allocating memory for an object from stack and from heap

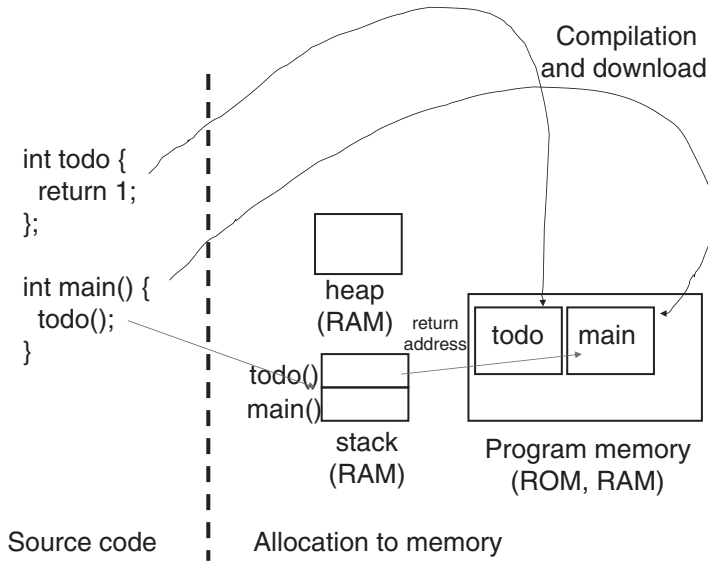


Figure 1.9 A program and associated infrastructural memory consumption

implicit allocations from heap even if a programmer assumes stack-based treatment. Such cases can lead to hard-to-trace errors that only become visible after an extended execution.

Programs are most commonly stored in ROM, flash, or disk, if such a facility is available in the device. When a program is run, it is usually loaded from its location in the storage to RAM for execution. This enables the use of updates for fixing bugs as well as the use of user-installed applications. Figure 1.9 demonstrates one way to allocate program and associated data in different memory locations.

For ROM-based programs, in-place execution can be used. This saves RAM, as there is no need to create an additional copy of the program into it. Usually this type of approach is used when a program is located in ROM, and it is beneficial to always run the program unaltered. For instance, in-place execution could be used for the introduction of a safety feature that enables (emergency) calls even if a mobile device is otherwise infected with a virus or some other malicious software. For flash memory, the type of flash used defines whether in-place execution can be used or not. As NOR flash can be used as direct memory space, programs stored to this type of memory are candidates for in-place execution, but because NAND flash behaves analogously to a hard disk, programs must always be loaded to RAM. There are also downsides to using in-place execution. First, it is impossible to upgrade the system using software modules that are downloaded after the installation. Second, it is not possible to use compression or encryption techniques, as the code must be executable as such. Therefore, in-place execution is becoming less favorable than it was in mobile devices where the option to download new applications was not available.

Generated Run-Time Elements

As already discussed, all programs and variables in them are allocated to memory locations when a program is executed. In addition to the basic cases, more sophisticated software structures introduce additional memory consumption. Such structures include infrastructure for inheritance and virtual machines, which we will discuss in the following.

In association with inheritance, an auxiliary data structure called a virtual function table is commonly generated. Its goal is to manage dynamic binding (Figure 1.10). When referring to a method during the execution of a program, dynamic binding does not refer to the method directly, but via a reference to the methods of the associated class and an offset that identifies the exact method to be called. Effectively, this implies that for all classes where dynamic binding can be used, a table is created where the cells of the table refer to different functions of the class. Then, when a program runs to a virtual function call, object type is used to select the right virtual function, and an offset included in the generated code is used to select the right function. As the outcome, all classes that potentially use dynamic binding require additional memory for the virtual function table, and, furthermore, add one extra memory word in each object for the type tag, which in reality often is a reference (or pointer) to the right virtual function table.

A virtual machine can be taken as a processor (or interpreter) implemented with software. This eases porting of programs implemented on top of the virtual machine, as the same (or similarly instructed) virtual machine can be used in different hardware environments. Fundamentally, two types of virtual machines can be considered. One is such that an interpreter is used that executes programs by simply interpreting each instruction as such. The other approach is that once the program to be run is fed to the virtual machine, the virtual machine compiles it into

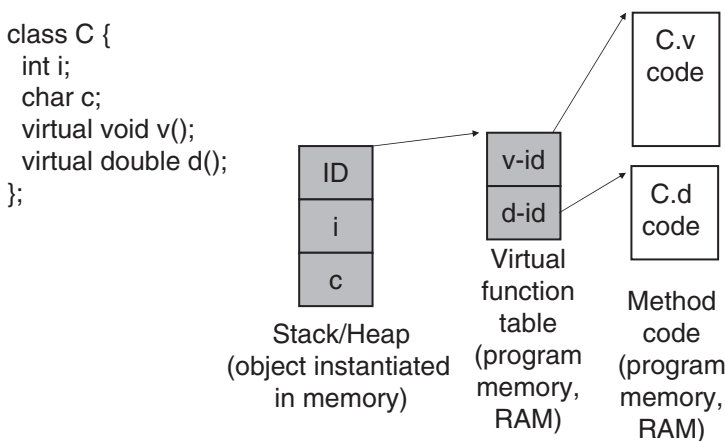
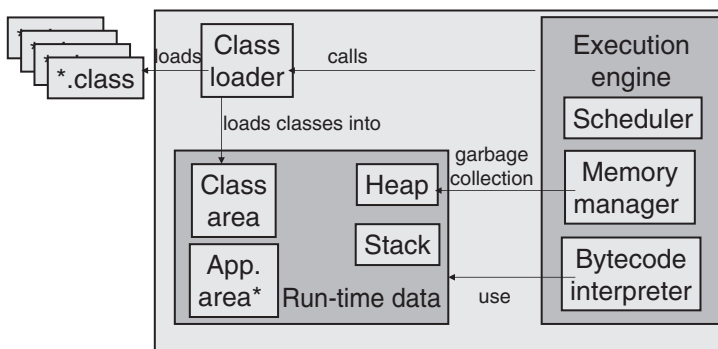


Figure 1.10 Virtual function table

an executable form that is closer to the actual machine code and can therefore be executed faster. Also a combination of the two is possible. Then, some commonly executed parts of the program for instance are compiled, whereas parts that are only executed seldomly are interpreted every time. This technique is referred to as hotspot compilation. A further issue worth considering is that since programs can be interpreted as data that are fed to a virtual machine, the machine can impose additional restrictions on programs. Moreover, it can control resource usage of programs, and manage loading of programs, an important facility to centralize in the mobile setting, to execution. A sample virtual machine, which reflects the features of a Java virtual machine, is illustrated in Figure 1.11 (Hartikainen, 2005). The roles of the different components of a virtual machine are listed in the following.

- Class loader is a component responsible for loading programs, given in terms of classes in our reference system.
- Run-time data is a container for loaded programs. It includes elements similar to those used in native executions for the purposes of the virtual machine.
- Execution engine contains a scheduler, which is responsible for selecting a thread for execution, a memory manager, which ensures that no illegal memory references are made and that memory is deallocated when it is no longer needed, and a bytecode interpreter, which executes the actual programs.

Garbage collection, i.e., freeing of resources that are still reserved but are known to be abandoned, can be implemented in two different ways using a virtual machine. Usually, a simple way is to implement garbage collection in cooperative (or stop-the-world) fashion, which stops the application while garbage is being collected. Therefore, only the garbage collector has access to data, and it can perform its task without a risk of modifying the same variables as the application at the same time. In contrast, parallel garbage collection can also be implemented, but this requires



* App. area includes e.g. Program Counter

Figure 1.11 Elements of a Java virtual machine

a more complex design because the garbage collector and the application being executed operate on the same data structures. In general, garbage collection is a wide topic with dedicated books (Jones 1999), and its detailed introduction at the level of individual algorithms, such as reference counting, mark-and-sweep, heap compaction, or more recent generational garbage collectors, falls beyond the scope of this book.

1.2.4 *Software Stack*

Building on top of kernel and other low level facilities, software used in mobile devices can be characterized using three different categories of software: applications and related facilities, middleware, and low-level software. The reason for separating these layers is that it is often desirable to consider that such levels of abstraction can evolve independently of each other, i.e., it is preferable that applications are independent of middleware version, and that middleware in general is not tied to a particular version of low-level software but can be used over several generations. Furthermore, maintained and relatively static interfaces are often defined to separate the layers. In the following, we address these categories one by one.

1. *Application-level software* is about the development of meaningful applications for an end user. In mobile devices, common application development rules of workstation software apply to a great extent, as it is often the intention of the device manufacturer to allow the introduction of additional applications. Implementation of other software components can benefit from using the provided interfaces of the lower-level components, but their implementations should not be relied on because different versions of software can be used in different devices. Moreover, a characteristic property of application-level development is usability, as otherwise users may reject the application.
2. *Middleware software* offers facilities that ease the development of applications. This often includes libraries such as support for using certain communication protocols. While an application developer is usually only using already existing systems, in some cases it is a necessity to define some new application-specific components. Then, it is common that the implementations must follow some predefined interfaces. In many ways, rules introduced for communications programming by Sridhar (2003) can be followed in the implementation of such components. The emphasis is often placed on portability in the sense that the details of the underlying hardware are not always benefited from even if this would result in improved performance. Despite this, characteristic properties include performance and memory awareness at this level of abstraction in practice despite device- and platform-specific features.
3. *Low-level software* covers kernel and device drivers, and virtual machines when applicable. Usually all such software is fixed by the device manufacturer. When developing such software, guidelines of Barr (1999) can be benefited from due

to the fact that in many ways the development of low-level software resembles the development of deeply embedded systems, with close connection to the underlying hardware. However, even at this level of abstraction, some precautions can be taken for reusability in future hardware environment, for instance. A characteristic property at this level is hardware awareness in general, but in a fashion that does not impose too harsh restrictions. In general, software associated with this level of abstraction is managed by a device manufacturer or platform developer.

To summarize the above discussion, the lower one goes in the sense of abstraction, the more one must understand the properties and restrictions of the platform and the hardware environment. Unfortunately, even at the application level, some properties of the environment remain visible, making applications a leaking abstraction. Moreover, when aiming at applications that can be used in multiple devices, also device variance can be considered as a main issue in the development.

A further problem results from the fact that the levels are seldom static but evolve as more and more devices are implemented on top of the same set of software components, often referred to as a platform.⁵ In this setting, it is common that upgrades take place and introduce new features to the platform as more hardware and processing facilities are introduced, as well as standardization advances. If performed in a careless fashion, the introduction of a new version of some low-level feature can lead to invalidation of some applications. Therefore, extra care should be paid for maintaining (binary) compatibility when composing new versions of middleware and low-level software. Otherwise, the platform can corrupt to a collection of devices in which one cannot run the same software, but must make device-specific modifications for all software that takes into account specifics of devices that are to be used for running the application.

For practical reasons, the development of a platform and the development of an application may have different concerns. For instance, for a particular application, many concerns related to variability to different types of devices – essential aspects of platform development – become irrelevant, if the application is targeted to only one type of device. Similarly, tailoring applications to run on as many platforms as possible often becomes relevant for an application developer only when the first running version is available, whereas tailoring a platform such that it will run all the applications constructed in accordance to some guidelines must be taken into account at the very beginning of the development as a compatibility requirement.

1.3 Development Process

The development process of a piece of software intended for a mobile device differs from that of conventional workstation software. The main reason is that first,

⁵ Strictly speaking one can consider that all the above levels constitute different platforms: OS platform, middleware platform, and application platform. Also other categories have been proposed.

the developed software is often tested in the development environment, usually a PC, with an emulator. Only when the software runs is it downloaded to an actual mobile device for testing, as depicted in Figure 1.12. The development workstation is commonly referred to as a host, and the final execution environment as the target.

The software tested with the emulator and the software downloaded to a mobile device can be identical, but they need not be. If the same programming infrastructure is available in both environments, like a common virtual machine for instance, the software can obviously be the same, as the case can be with Java. However, if there is no infrastructure that would enable the use of the same software, different compilations are needed. The process where a workstation is used to compile an executable which is downloaded to a different system is referred to as cross-compilation. While it is possible to emulate the low-level behavior of a system, differences in available environment sometimes lead to early use of the actual device even in the development phase.

Obviously, being able to run the same version in the development and the final execution environment eases debugging, as only one tool-chain is needed. In the best case, downloading the application to the final environment is trivial. However, if cross-compilation is needed, the phase where software runs in the emulator can be only the beginning of debugging and testing activities that are necessary until the program is completed. For instance, there can be additional requirements regarding compilation, if the mobile device does not support all the features of the emulator. For example, older versions of Symbian OS (before v.9.0) were unable to handle

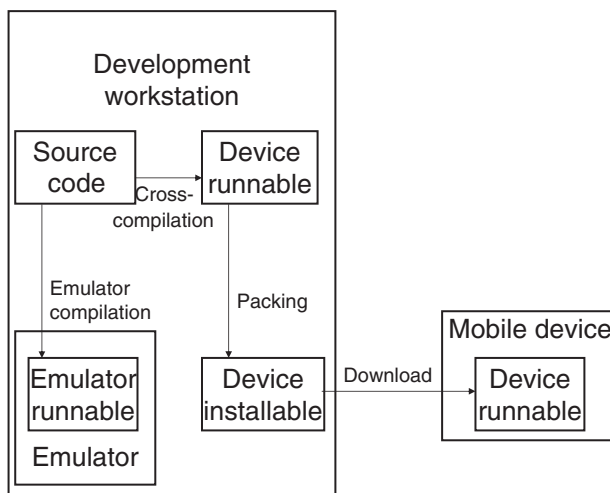


Figure 1.12 Development process and software download

global variables in dynamically loaded libraries in devices,⁶ but forced one to use a platform-specific feature called thread local storage (TLS), where thread implementation was piggy-backed with global variables (Tasker et al. 2000). However, this problem only arises in the cross-compilation phase, as it is related to the final execution environment, not to C++ that is used as the programming language or emulator, which runs on the development workstation and can deal with global variables without problems.

In addition to the actual cross-compilation, many environments require the use of installation files. In addition to plain compiled programs, such files can often be extended with additional data, which can be auxiliary data files, or sound or graphics extensions for a certain program, for instance. When a package containing all these files is fed to the installer application residing on the mobile device, the installer then unpacks the files and places them to convenient locations as defined by the package.

1.4 Chapter Overview

The chapters of this book introduce the main concerns of the design of software for mobile devices. Unlike many other introductions, we have organized the presentation in accordance to concerns, which are memory usage, concept of applications, modularity and available mechanisms, concurrency, generic resource management, networking, and security. Individual chapters and their contents are listed in the following.

Chapter 1 has introduced the basics of mobile devices in terms of underlying hardware and software. The goal of the chapter is to introduce the basic concepts on top of which further chapters will build on.

Chapter 2 discusses memory management, which a designer must master in the mobile setting. As memory is one of the restricted elements of a mobile system, it is important to understand the basic principles of managing its use. The chapter introduces some design patterns, i.e., reusable design decisions that solve certain problems in a predefined context (Gamma et al. 1995), for memory-aware software, and gives examples on memory management in mobile devices. Similar topics have already been addressed by Noble and Weir (2001), although their focus is been wider.

Chapter 3 introduces the concept of an application. While in principle, all software can be implemented from ground up, mobile software platforms usually introduce a prescribed architecture for applications. Furthermore, another important issue is packaging applications for delivery to a device.

⁶ The reason for not using global variables in dynamically linked libraries lies in the fact that at least one additional memory page, i.e., memory allocation unit, would be reserved for the library if global variables were allowed. This would lead to a considerable overhead.

In Chapter 4 we discuss the use of dynamically linked libraries, and demonstrate how available implementation techniques become visible to a programmer in a mobile environment. Again, the purpose is to address how the selected design and implementation decisions are unveiled to programmers.

Chapter 5 introduces the most important mechanisms of concurrency applicable in the mobile setting. The chapter addresses issues like whether to use threads, which are a common technique when programming desktop systems, or to use other means of serialization, which can offer less resource consuming solutions. Moreover, the chapter considers reusability of such designs.

Chapter 6 integrates concurrency with the management of different resources that must be handled in a mobile device. The chapter also discusses problems associated with the fact that it is not uncommon that even seemingly similar devices can include some differences in their hardware, which can sometimes lead to complications in the development of associated software.

Chapter 7 defines how a mobile device can interact with networks. The goal is to describe how mobile devices can be used in a networking application, and how software residing in a mobile device should be designed, and general topics associated with distributed systems and their implementation are mainly overlooked. Furthermore, the chapter also discusses two cases, one on using Web Services with a mobile device, and another on ad-hoc networking over short-range wireless protocols using Bluetooth as an example.

Chapter 8 is dedicated to security properties of a mobile platform. The chapter discusses both design patterns for secure designs as well as approaches adapted in existing platforms. The rationale of locating this important topic towards the end of the discussion is that security is an issue that contributes to all the previous techniques in a fundamental fashion. Without first discussing the basic implementation, it would be impossible to introduce a suitable security concept for the different issues.

In each chapter, we will use mobile Java (Riggs et al. 2001; Topley 2002) and Symbian Operating System (Edwards et al. 2004; Harrison 2003; Tasker et al. 2000) as examples on real-life mobile platforms. The reason for using these systems is that in many ways they contradict each other. In mobile Java, the underlying mindset is that it can be introduced as an external facility to all proprietary phones, and that the run-time infrastructure will adopt a major portion of the complexity of dealing with scarce resources. In contrast, in Symbian OS, the complexity is explicitly made visible to the programmer, together with certain patterns and idioms that are to be used when dealing with resources. When used with care, this in principle yields improved performance, because the whole power of the device's hardware can be used.

In addition, we also give some exercises on the topic discussed in the chapter. However, the purpose is not to focus on these particular platforms but to maintain a more abstract approach, but examples merely characterize some design solutions made in real platforms.

1.5 Summary

- Programming of mobile devices is fundamentally the same as programming of desktops. However, design flaws are more prone to cause problems, as resources are scarce in mobile devices. Moreover, available facilities are usually not as advanced as when programming a workstation, but developers must sometimes rely on practices more commonly associated with programming of embedded systems.
- Understanding the characteristics of the underlying hardware and software infrastructure is a practical necessity for dealing with leaking abstractions in the mobile setting.
- While there are several types of devices, they share the same basic architecture.
 - Processor, a hierarchy of memory facilities, and auxiliary subsystems for interacting with the environment and enhanced features.
 - Operating system, middleware, and applications, whose reliability requirements may differ. This has implications for software development as well.
 - Ability to reuse existing infrastructure as a common platform in multiple devices is preferable.
- A fundamental design decision of future mobile devices is whether to use symmetric multiprocessing versus a collection of specialized pieces of hardware.
 - Symmetric architecture eases the development of applications in the sense that all facilities appear similar to the programmer.
 - Asymmetric architecture can be more specialized in its resource use, which often hardens design and adequate use of available facilities.
- Run-time software infrastructure can introduce overhead; examples of such overhead include virtual function table used for implementing dynamic binding associated with inheritance, and virtual machines.

1.6 Exercises

1. What solutions would be applicable for defining an abstraction that does not leak for strings? Which programming infrastructures (languages, operating systems, etc.) use them?
2. What properties of applications have potential for leaking in the mobile environment, assuming that a programming language familiar to you is available? How should the developer address these properties?
3. What differences can you find in your own software programs when considering the way in which they use stack and heap? Why?
4. Consider a calendar application running in a mobile device. Which services does it use from low-level, middleware-level, and application-level components?
5. Which features of programming languages, such as C++ or Java, can be problematic when programming mobile devices? What kinds of coding rules or idioms

could be given for a developer to ease the development, assuming that for instance memory usage and resource management are to be controlled better?

6. In-place execution enables the execution of programs directly from ROM. What rationale is there not to implement all features of a mobile device using this technique?
7. What functions could be separated from the main processor to be executed by some auxiliary unit? What would be a reasonable level of abstraction to offer to the application developer?
8. What parts of the hardware should be standardized for a mainstream workstation operating system to be assumed as the platform for mobile applications? What would this mean for application developers?