MiniOS

SAM4S Cortex-M4 version

DRAFT

Rafael Roman Otero, Conan Veitch, Alex Aravind

2018.

Contents

Preface

MiniOS is an instructional microcontroller operating system. It was originally written to be used in calsrooms, but we wanted to make it available to the public, whether they are comp sci students, engineering students, or self taught. In Operating Systems, like in most other computer science subjects, those studying it ought to learn it by writing their own OS. However, the size and complexity of both modern OSes, as well as modern CPU architectures, render the possibility of a custom-built OS impractical. To this problem, MiniOS attempts to offer an unorthodox solution.

How? First, MiniOS design is minimal, which means we only implemented those features that are essential for the system to keep its *OS-ness*. So the reader will find that, although MiniOS has no concept of virtual memory, surprisingly it does have minimal versions of memory protection, multitasking, events, and asynchronous IO. Moreover, these minimal versions are functional. That is, they function and they can be used to build things. Second, compared to Desktop computers, MCU have relatively simpler architectures, have friendlier development environments, and have off-the-shelf debugging capabilities. These three things together, we think, make it easier for the reader to build the system. After all, no one likes to program black boxes, we like to *see* as we write code. Lastly, the last part of our approach is this book. This book contains a step-by-step explanation on how the entire system is built from scratch. So detailed, in fact, that its first part is dedicated solely to the Cortex-M4 architecture.

What we hope

It is our hope with this material to show in a small scale, how OSes do what they do, and—more importantly—why they do it; while, at the same, learn the mysterious inner workings of the computer machinery we call computers. We also hope to provide the reader with enough basics and intuition as to encourage her to explore beyond what our limited imagination let us demonstrate in this book.

A word of advise

Often readers of a tutorial are able to skim through the contents and still manage to get what they need out of it. While this is a very handy ability to have, the writer would like to warn that low-level programming is a task that one must approach (as Dijkstra put it in his EWD340) very humbly.

Good luck!

About this book

This book is split in two parts: an architecture part and a system part. The architecture part discusses general principles of computer architecture, and introduces the reader to the inner workings of the ARM Cortex-M4 and the SAM4S microcontroller board. Notice this part alone could very well be (and it has been) used to teach principles of computer architecture (with an emphasis on working knowledge). The latter part is dedicated to the construction of MiniOS from the ground up. Those interested in only building the system can skip the architecture part, and use it instead as reference material.

The material required to follow this book are the Atmel SAM4S Xplained Pro Starter Kit, and a Desktop or laptop computer for cross-platform development. Complementary online material is often referenced, so a connection to the internet is also necessary.

The material is intended to be self-contained to a point. Specifically, througout the book it is assumed prior experience with C and a basic understanding of digital circuits, data structures, and computer systems.

Part I

Architecture

Chapter 1

Introduction

1.1 Computers of different shapes

We generally associate the term computer with laptops, desktop computers, or even mobile devices. This is only a very small sample of computers, however. Most computers exist as part of other systems (say mechanical or electronic systems), and they range from simple coffee makers to incredibly complex aircraft navigation systems. These computers are referred to as *embedded*, and are different in that they are dedicated special purpose devices. Unlike general-purpose computers, embedded ones tend to serve a single purpose throughout their life-time. Consider an everyday printer. While it is possible to replace printer software with software of a different type, only more printer software would make sense.

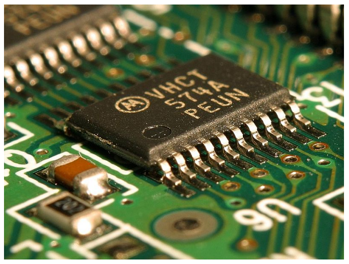


Figure 1.1: IC or Microchip on a circuit board

Embedded computers can be built from either microprocessors or microcontrollers. Let us look explore the differences.

1.2 Microprocessors

A *processor* or *central processing unit* (CPU) is the hardware that executes instructions in a computer, and it is composed of circuitry such as registers, the arithmetic logic unit, and the control unit. All these components can each be obtained as *integrated circuits* (ICs), also known as *microchips*, and placed together in a circuit board as shown in Figure 1.1. In fact, that is how computers were built in the past. However, for other than educational purposes, there is absolutely no reason to do this any more, as processors themselves are now built as ICs. To be more precise, they are built into a very specific part of an IC, a single piece of semiconducting material known as *die*[1].

Microprocessors (MPUs) are processors built in a die; the die integrates all the transistors that form all the components of a CPU. Figure 1.2 shows the Eniac-on-a-chip, a die replica of the first general-purpose computer, which weighted over thirty metric tons.

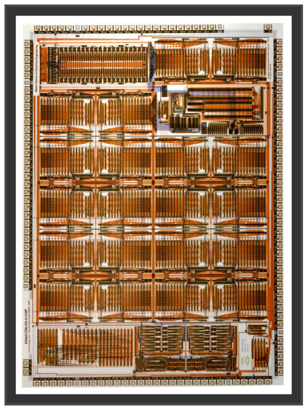


Figure 1.2: *Eniac-on-a-chip* from *(attribution missing)*

A microprocessor by itself is not a computer. It would require at least some form of memory and I/O to make one. In the case of a dektop computer or laptop, all these components are soldered as part of the *motherboard*. Is it possible to make computer boards even smaller? The short answer is yes, and in the next section we discuss how microcontrollers achieve this.

1.3 A computer in a die

The next step in this shrinking process is to integrate CPU, memory, and I/O into a single die; and this is exactly what microcontrollers do. A *microcontroller* (MCU) is a computer in a die. From the outside, they look just as any other regular IC, and often are indistinguishable from MPUs.

[1] A microchip consists of a die enclosed in a package and connected to external pins via wires. See Figure 1.3

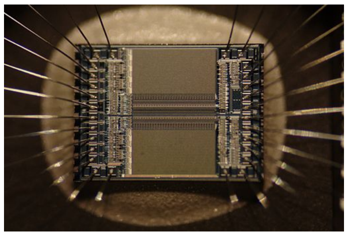


Figure 1.3: Die in a microchip package. Photo by Zephyris / CC BY

MCUs and MPUs serve different purposes. The former ones are easier to connect with the world because I/O peripherals are built in. Also, there is no need for a memory controller, nor the memory itself to be placed in a motherboard together with the MPU. This means the effort that goes into designing MCU-based computer boards is less. Different MCUs have different I/O peripherals, from timers and clocks to Ethernet and USB controllers. The fewer external components required by a MCU allows for smaller computer boards and reduced production cost; which is good for, say, a washing machine manufacturer.

It is no surprise MCUs are typically the choice in small electronic systems such as computer keyboards, hard drives, refrigerators, anti-theft alarms, IoT objects, and even small robots. Modern cars, in particular, are a very good example of how ubiquitous MCUs have become given that modern cars are shipped with no fewer than fifty of them.

Figure 1.4: EDBG chip debugger in the back of the SAM4S board

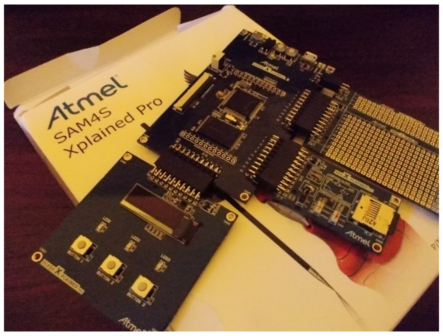
Nevertheless, MCUs are constrained in memory and CPU sophistication. The capacity of main memory seldom surpassess 512 KB; cache sizes go from none to a few KB; the fastest of them run at no more than a few hundreds of MHz; and certain more advanced CPU features such as memory management unit are not present. This is why MPUs are used in applications that demand more resources or more sophistication. Applications such as laptops, tablets, mobile phones, teller machines, WiFi routers, industrial and military robots, and air-craft systems. Incidentally, all these computer systems typically have microcontrollers for secondary processing. In other words, computers often have smaller computers alongside it doing specialized subtasks.

Now let us introduce the SAM4S board, which will serve as computer for the remaining of this book.

1.4 MiniOS target hardware platform

The *Atmel SAM4S Xplained Pro Starter Kit* shown in Figure 1.5a is composed of the SAM4S board (the main board), and three more daughter boards: IO1, OLED1, and PROTO1. The SAM4S board uses an Atmel *ATSAM4SD32C* MCU for processing. This MCU has 160 KB of SRAM as main memory, 2 KB of cache, an ARM Cortex-M4 as processor, and a max clock frequency of 120 Mhz.

In addition, the Atmel BNO055 expansion board in Figure 1.5b (an orientation sensor), is used in some exercises in an attempt to make things more interesting. Yet it is optional and the reader may continue without it.



(a) SAM4S Starter Kit

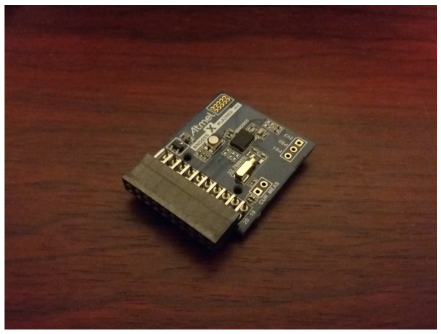

(b) BNO055 expansion board (optional)

Figure 1.5: The hardware platform

1.5 Development Environment

To write software for the SAM4S board a cross-compiler is needed. A *cross-compiler* runs on one computer platform (the host computer), and creates executables for a different platform. Particularly, we will be using the GNU Toolchain for ARM, which, among other tools, includes a C compiler and an assembler.

In the host computer, source code is first compiled into an executable for the ATSAM4SD32C MCU, and then it is transferred to it using a flashing tool. Once the executable is deployed, an on-board chip debugger (Figure 1.4), interfaces with the host to enable debugging of the MCU.

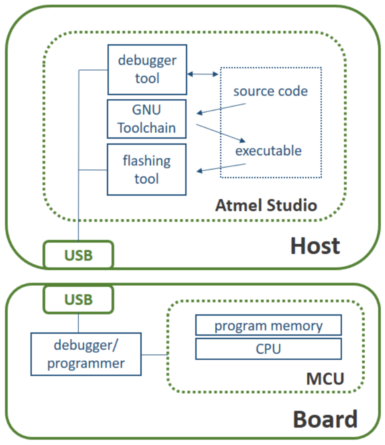


Figure 1.6: Development Environment

All of the different host-side software tools needed for development are integrated in Atmel Studio. Communication between the target platform and host is via USB. Figure 1.6 depicts, in a simplified manner, the described programming environment.

Additional video material: Introduction—Plugging the SAM4S board

1.6 Installing Atmel Studio

The latest version of Atmel Studio (version 7 as of today), can be downloaded from Atmel Studio's official website.

1.7 Hello World

The code base for this chapter is SAM4S XPro Bare Metal. It is a zipped Atmel Studio project, and it contains the hello world assembly program in Listing 1.1.

```
.syntax unified

.section .vectors
.skip 4
.word main

.section .text
.thumb
.type start, %function
start:
   mov r0, #0x01
   mov r1, #0x02
   mov r2, #0x03
   b .
```

Listing 1.1: Hello World SAM4S Bare Metal

To run it extract the zip file and open the solution in Atmel Studio. Then assemble the source code and flash it to the MCU.

Additional Video Material: Introduction—Running and debugging the SAM4S hello world

Resources on the web

- ENIAC Computer History Archives Project - remastered original 1946 Film & Narration Army Military
- (Video) Programming my 8-bit breadboard computer

Exercises

1. Using the *mov* instruction, write a program that swaps the contents of any two registers. Using Listing 1.1 as start point, replace lines 11 to 13 with your own instructions.

In your own words, answer the following questions:

2. What is the difference between an embedded computer and a general purpose computer?
3. What is the difference between processor, microprocessor, and microcontroller?
4. What is a cross-compiler and why is it needed?

Advanced exercises:

8. Using the *xor* instruction, write a program that swaps the contents of two registers. To do $a = a \text{ xor } b$ you have to say, for example, *eor r1, r2*, where *r1* holds a and *r2* holds b. The mechanism to swap them is explained in XOR swap algorithm.

Chapter 2

Instruction Set Architecture

People who are really serious
about software should make
their own hardware

Alan Kay

Instruction Set Architecture (ISA) is the interface between hardware and software. Particularly, the interface between the processor and programmers, or compilers for that matter. That is because an ISA, or *architecture* for short, specifies all the details concerning the behaviour of a processor from a software perspective.

Processor and architecture are not the same thing. To understand the difference between them consider the x86 architecture, which has been implemented in Intel as well as AMD processors, for instance. Although internally both processors are quite different, from an external point of view they are identical since they all implement the same instruction set. In other words, they are *compatible*.

The ATSAM4SD32C MCU uses an ARM Cortex-M4 as processor core. The following section elaborates on its architecture.

2.1 ARM Cortex-M4 ISA

The Cortex-M4 is a 32-bit processor core[1], licensed by ARM Holdings[2], and it implements the *ARMv7E-M* architecture. The ARM documentation details this ISA in the document *ARMv7-M Architecture Reference Manual* (note the missing *E*).

A very important part of any architecture are its registers. They are discussed in the following section.

2.2 Registers

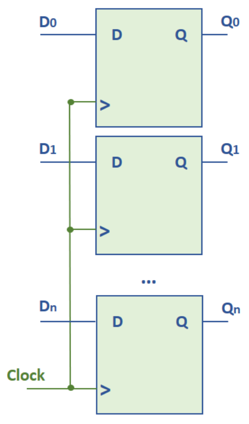


Figure 2.1: n-bit register

Physically, a register is a collection of flip-flops, with one flip-flop storing one bit of information (Figure 2.1). These “memory cells” are built within the processor core itself, and thus very close to where the execution of instructions takes place. This allows for very short travelling time of the electrical signal between registers and the arithmetic logic unit. Shorter travelling distance means shorter accessing time, which is important because **CPU instructions operate on registers**.

The Cortex-M4 has over twenty registers (Figure 2.2): thirteen general purpose registers (R0-R12), three registers with specific purpose (R13-R15), and five special registers. For now, only general purpose registers are considered, while the remaining are discussed in subsequent chapters.

[1] Roughly speaking a *core* is the part of the CPU that does the actual execution of instructions. A core needs extra circuitry to make a complete CPU. For example, cache, more cores, and video processing facilities.

[2] ARM does not manufacture microprocessors, instead the ISA and core designs are licensed to third party companies, which may manufacture microprocessors, microcontrollers, or other ICs. Examples of such companies are Atmel, Texas Instruments, and NXP Semiconductors.

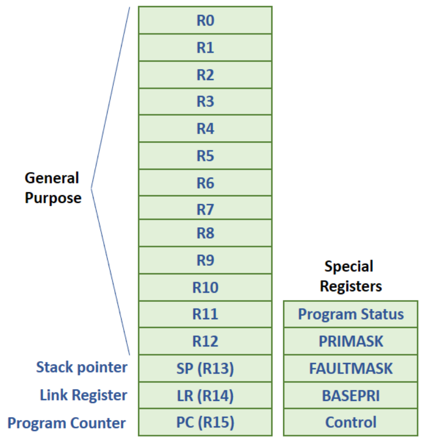


Figure 2.2: Cortex-M4 Registers

General purpose registers are used precisely for that: any purpose. For storing intermediate values of a computation or passing parameters between functions, for instance. Let us consider the former case. Say the reader wishes to compute the the arithmetic expression 1 + 7 + 19 in her head. To compute the result mentally one has to first add two of the available operands together, and then add its result to the third remaining operand. Note that during the computation, the (intermediate) result of the first addition is "temporarily kept in memory". In the same manner, it is possible to continue to do longer computations by writing intermediate results in paper. That is what registers are in modern computers: **a place for temporary storage where operations (instructions) can act upon.**

The same program in assembly is shown in Listing 2.1. The actual instructions appear in lines 11 to 15. This program stores (or moves) the value 1 in r0, 7 in r1, and 19 in r3. Then it adds the contents of r0 and r1 and stores the result in r2. Lastly, it adds the result of the previous addition in r2, with the contents of r3, and the result is stored again in r2.

```
 1 .syntax unified
 2
 3 .section .vectors
 4 .skip 4
 5 .word start
 6
 7 .section .text
 8 .thumb
 9 .type start, %function
10 start:
11     mov r0, #1
12     mov r1, #7
13     mov r3, #19
14     add r2, r0, r1
15     add r2, r2, r3
16     b .
```

Listing 2.1: Program using registers

Now it is time to introduce the assembly language used to program the

Cortex-M4. Please note that, due to the interdependencies of concepts, the following section is not meant to be understood in one pass.

2.3 Assembly Language

Assembly or assembler language is a language that is very close to machine instructions. For most assembly instructions there is a one to one correspondence between them and machine code (in binary). Assembly programs are said to be "assembled" by an assembler (as opposed to compiled by a compiler).

2.3.1 Assembler directives

Assembler directives are specified by a dot "." at the beginning of each command, and they are **instructions or information for the assembler** to consider during assemble time. They are not part of the final executable, although they alter its final form.

In the example in Listing 2.1, *.section .text* tells the assembler that any following instruction should be placed in the *.text* section in memory[3]. Similarly, *.thumb* tells the assembler that the instructions that follow should be assembled using Thumb encoding[4]. Because the Cortex-M4 only supports Thumb-encoded instructions, they must appear preceded by one *.thumb*.

Now, at runtime the CPU can be in one of two modes, ARM mode or Thumb mode[5]. Switching of the CPU from one mode to another takes place on every function entry; and once the CPU is placed in one mode, it remains so until told otherwise at another function entry. It is difficult to explain at this early point why *mov r0, #0x01* is a function entry point[6].

[3] Memory is organized in sections. For now it is enough to know that instructions live in the *.text* section

[4] Instruction encoding refers to the binary produced when assembling an instruction. ARM has two available encodings: ARM and Thumb.

[5] To execute Thumb instructions the CPU must be in Thumb mode

[6] Technically *start* is an interrupt entry point (i.e. an interrupt handler), but because interrupt handlers are at a low level indistinguishable from functions, the mechanism

So, for now, it suffices to know that because execution begins at *mov r0, #0x01*, the program must tell the CPU that the instruction at address *start* is a function entry by saying *.type start, %function*. It can be seen as the program telling the CPU "hey do not forget to change to the right mode at this specific point". What mode? Thumb mode, since the instruction at "that point" is preceded by a *.thumb* directive. In fact it is possible to replace both *.thumb* and *.type start, %function* with a single directive (*.thumb_func*) as shown in Listing 2.2.

```
 1 .syntax unified
 2
 3 .section .vectors
 4 .skip 4
 5 .word start
 6
 7 .section .text
 8 .thumb_func
 9 start:
10    mov r0, #1
11    mov r1, #7
12    mov r3, #19
13    add r2, r0, r1
14    add r2, r2, r3
15    b .
```

Listing 2.2: Hello World using the thumb_func directive

As for *.syntax unified*, it tells the assembler that the program contains Unified Assembly Language (UAL). UAL was introduced by ARM as a common assembly capable of generating both ARM and Thumb encoded instructions from the same assembly language, hence its name. UAL is the assembly we will be using throughout this book. So every assembly file must begin with a *.syntax unified* directive at the top.

2.3.2 Labels

Labels are human friendly names for memory addresses. You can identify them by the colon ":"" at the end of them. In Listing 2.2 *start* is a label. How exactly are addresses assigned to labels? In the same example, due to line 7, the name *start* corresponds to the address in memory where the *.text* section begins, wherever that is. Another way of looking at it is that *start* corresponds to the address where *mov r0, #1* is stored in

is the same. (Do not worry if this does not makes sense. Interrupts are covered later.)

memory. How so? Again, due to *.section .text*, we know *mov r0, #1* is the very first instruction in the *.text* section, wherever that is.

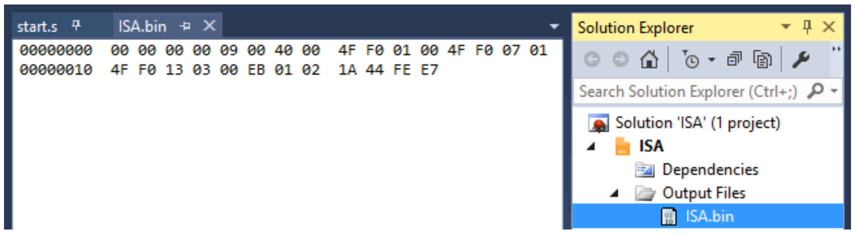
So, for instance, in the example in Listing 2.3, *some_label* corresponds to the address in memory of *mov r1, #7*. It is important to notice that **the address in memory of an instruction is the location in memory of the first byte of the instruction**, as Cortex-M4 instructions occupy at the very least two bytes.

```
...
start:
   mov r0, #1
some_label:
   mov r1, #7
...
```

Listing 2.3: Program using registers

2.3.3 Instructions

Instructions in an assembly program must appear one per line. Every consecutive line in the source code represents binary instructions stored at consecutive locations in memory. While some instructions are translated into 16-bit values, other are translated to 32-bit values. Altogether they are commonly referred to as **machine code**. Consider, for instance, the machine code in Figure 2.3 generated from assembling Listing 2.2.

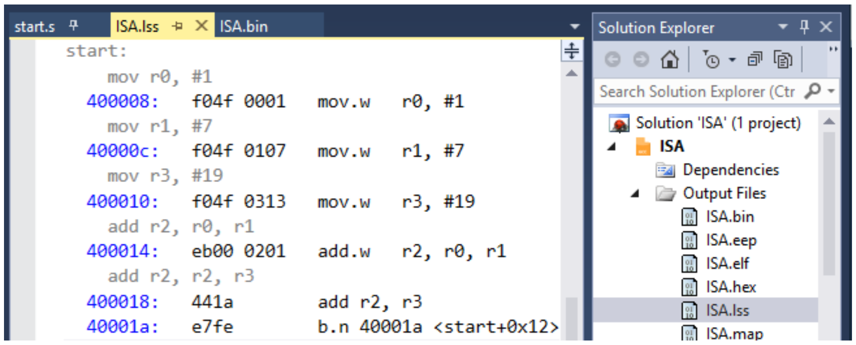


```
00000000  00 00 00 00 09 00 40 00  4F F0 01 00 4F F0 07 01
00000010  4F F0 13 03 00 EB 01 02  1A 44 FE E7
```

Figure 2.3: Sample machine code generated by the GNU assembler

The actual mapping of assembly instructions and machine code is made explicit in the disassembly *ISA.lss*, as shown in Figure 2.4.

The numbers on the left column are the memory addresses where instructions will reside in memory, once they are flashed to the MCU. The



```
start:
   mov r0, #1
 400008:    f04f 0001    mov.w    r0, #1
   mov r1, #7
 40000c:    f04f 0107    mov.w    r1, #7
   mov r3, #19
 400010:    f04f 0313    mov.w    r3, #19
   add r2, r0, r1
 400014:    eb00 0201    add.w    r2, r0, r1
   add r2, r2, r3
 400018:    441a         add r2, r3
 40001a:    e7fe         b.n 40001a <start+0x12>
```

Figure 2.4: Sample disassembly

column on the center is machine code. The column on the right is the assembly equivalent of each instruction in the center. Take into consideration that the disassembly is generated from the translated binary and it is not shown in UAL. So its syntax may differ slightly from the original[7]. Also, the disassembly is explicit on whether a 16-bit or a 32-bit instruction was used (some instructions are available in both versions). The postfix *.w* denotes 32-bit version, whereas the lack of it denotes 16-bit. Typically the assembler picks the smallest version of the instruction that can be used, unless we explicitly tell it to.

To conclude this section let us visualize (Figure 2.5) the machine code for the sample programs as it is usually depicted in classrooms.

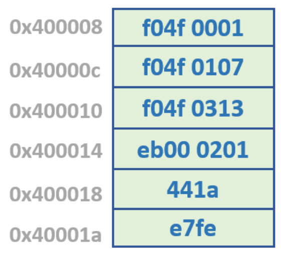


Figure 2.5: Visualization of sample machine code in memory

Note the granularity of the memory addresses shown is per 4 bytes, i.e., 0x400008 followed by 0x40000c, and so forth. To show it with, say, a resolution of 1 byte (i.e 0x400008 followed by 0x400009, followed by 0x40000a, and so forth) it is necessary to introduce the concept of **endiannes**. This is, however, left as an exercise.

Wait! There is one piece missing. What about *.section .vectors*, *.skip 4*, and *.word start* ? Enough confusion for one section. They will be covered in later chapters.

[7] For instance, the instruction in UAL *add r2, r2, r3* is shown as *add r2, r3*

2.4 Working with ARM's documentation

The set of available instructions for the Cortex-M4 is varied. Some of them are very simple and intuitive, and some of them are not. Luckily, only a handful of simple instructions is all that is required to start programming in assembly. In general, instructions are similar, they take zero or more operands and do some operation with them. When reading documentation, this is the notation used:

- Rd—the destination register
- Rn—the register holding the first operand
- Operand2—a second operand, which can be a constant or a register.

This is a simplified version of what is in the documentation. Full documentation of instructions is available in Chapter 3 of the document Cortex-M4 Devices. Generic User Guide.

In the following it is assumed the reader has the ability to look up in the documentation any instruction mentioned (say *b* or *ldr*). Beware, though, that documentation is written in a very detailed manner, and it may be difficult to follow (specially at the beginning). Most details can be ignored, however. Use the debugger to your advantage. It may be useful to take the examples in there provided and play with them. Visually inspect what effect they have on the machine.

Assembly instructions by themselves are of little use, unless we can somehow combine them to write programs. In the next section we look at how to combine instructions to describe repetition and conditions in a program.

2.5 Loops and conditionals in assembly

By now the reader understands instructions are executed in sequence, one after another. Internally it works as follows. CPUs, including the Cortex-M4, typically have a *program counter* register. **The program counter (PC) holds, at all moments, the address of the next instruction to be executed**. Internally, in the CPU, every time a new instruction is fetched for execution from memory (from where in memory? from whatever address *PC* holds), the value of the program counter is automatically incremented by one instruction. In this manner execution happens sequentially.

This sequential model of execution is rather limiting, however, as often it is necessary to express conditional or repetitive behaviour in programs. The only way to disrupt that sequential flow of execution is with **branch instructions**. A branch instruction (with syntax *b reg* where *reg* is a register) copies the memory address in *reg* to the program counter, thus effectively branching (jumping) to that memory address. Thereafter execution continues in a sequential fashion.

As example, consider expressing the condition *if(r1 == 17) r1=0;* in assembly. The code that does it is in Listing 2.4

```
...
start:
   cmp r1, #17       /* if( r1 == 17 ) */
   bne end           /*                */
   mov r1, #0        /*     r1 = 0 ;   */
end:
    b .
```

Listing 2.4: Sample conditional

Now consider adding an else condition to it. Say, *if(r1 == 17) r1=0; else r1+=1;* in assembly. The corresponding code is in Listing 2.5

```
...
start:
   cmp r1, #17       /* if( r1 == 17 ) */
   bne else          /*                */
   mov r1, #0        /*     r1 = 0 ;   */
   b   end           /*                */
else:                /* else           */
    add r1, r1, #1   /*     r1++;      */
end:
    b .
```

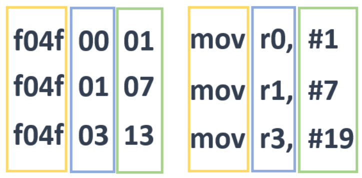
Listing 2.5: Sample conditional plus else

Additional Video Material: Introduction—Conditional execution in assembly

2.6 Immediate values

When specifying a constant operand in a *mov* instruction, say *mov r0, #7*, the value of 7 has to exists somewhere. This is so that, when the *mov* instruction is fetched and executed, the value #7 is available to the CPU. *The CPU cannot create values out of nowhere*. Certainly, the value can be placed in memory and then we can instruct the CPU to load that value from memory into a register. That is exactly what the instruction *ldr r0, =address* does[8]. Okay problem solved—not quite. Think for a second of the concept of asking someone to add two numbers. Say Mariana and her friend are texting and Mariana gets the text: "Hey quick how much is 2 times 21 ?" Mariana quickly replies "It's 43" (yes, that is wrong). The question here is—where were the values 2 and 21 stored? Where did Mariana get them from? They were encoded as part of the question itself. When Mariana reads the question she has all the information she needs to carry out her friend's instruction. Analogous to the *ldr* instruction that loads from memory, Mariana's friend could have written "Hey go check you inbox. I have sent you two numbers, can you please multiply them for me?". But there is absolutely no reason to over-complicate things. Specially not when the operands are two constant values whose value was decided at the moment the program was being written, or the moment Mariana's friend was asking the question, for that matter.

Giving instructions to a computer is no different. **Constant values in instructions are encoded as part of instructions themselves**. Such values are know as **immediate values**. To see a clear example consider the three *mov* instructions in Listing 2.2, and its corresponding machine code in Figure 2.6 (immediate values appear in the green box).



```
f04f 00 01    mov r0, #1
f04f 01 07    mov r1, #7
f04f 03 13    mov r3, #19
```

Figure 2.6: Encoding of assembly instructions in machine code

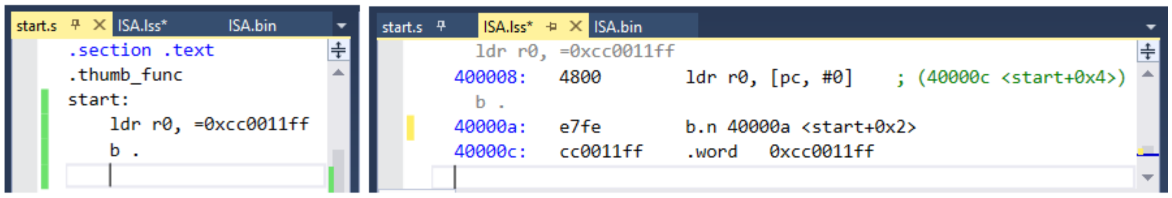
2.7 Moving 32-bit immediate values to registers

Given that Thumb instructions can be encoded in at most 32 bits, moving large immediate values to registers is a problem. Say we want to move

[8] assuming *address* contains the value 7.

the 32-bit value *0xcc0011ff* to a register using *mov*. If the value alone occupies 32 bits, how many bits are left available for encoding the rest of the instruction? Definitely not 32. So if we attempt to assemble a program containing the instruction *mov r0, #0xcc0011ff*, the assembler will complain with the message **invalid constant after fixup**.

The solution is to use the pseudo instruction **ldr *reg, =immediate***, where *reg* is a register and *immediate* is an immediate value. A pseudo instruction is an instruction that is presented as an instruction, but it is not. In this case, the assembler creates at assemble time a **memory pool of constants**, and generates one instruction that will load that constant value from the pool at runtime. It sounds complicated, but it is simply replacing the original *ldr* (the one with the immediate value) with a *ldr* that reads the constant value from where it was stored in memory. In the case of our example, the instruction *ldr r0, =0xcc0011ff* is translated to *ldr r0, [pc, #offset]*. That just means the constant pool was placed *offset* bytes away from the current value of *pc*, and *ldr r0, [pc, #offset]* is just loading the constant from there[9]. Figure 2.7 shows the example alongside its generated machine code.



```
.section .text
.thumb_func
start:
    ldr r0, =0xcc0011ff
    b .
```

```
    ldr r0, =0xcc0011ff
400008:   4800        ldr r0, [pc, #0]    ; (40000c <start+0x4>)
    b .
40000a:   e7fe        b.n 40000a <start+0x2>
40000c:   cc0011ff    .word    0xcc0011ff
```

Figure 2.7: Loading constants with *ldr* and *pc*-relative addressing

This is something to keep in mind when looking at the disassembly of an assembly program that uses *ldr* to load constants. Question—why is the *ldr* offset 0 and not 4 in Figure 2.7? Answer left as exercise.

Additional Video Material: Introduction—Loading constants to registers with LDR

[9] This technique of specifying a memory address using the program counter as base is known as **pc-relative addressing**

Resources on the web

- (Video) sHow transistors work - Gokul J. Krishnan
- (Video) Bus architecture and how register transfers work - 8 bit register - Part 1
- (Video) Stepping through a program on the 8-bit breadboard computer
- x86 Disassembly/Loops

Exercises

Base code for this chapter is available at ISA Base Code

1. The document First Draft of a Report on the EDVAC was one of a series of documents describing what we now know as the Von Neumman architecture. It was written by John Von Neumann and distributed in 1945 by H. Goldstine. Pick one section that you find interesting and explain it. Mention why you think it's interesting.
2. Briefly explain how multiplying and dividing by 2^n can be achieved using logical shift operations. Demonstrate with a few examples for Cortex-M4 assembly.
3. Pick one of the rules from De Morgan's laws and write a program that verifies it.
4. Let r0 and r1 hold the addresses in memory[10] of two strings of your preference, respectively. A string is just a set of contiguous bytes[11] that end with the null character *0x00*. Write an assembly program that concatenates them and writes the result somewhere in RAM.
5. Write a program that multiplies the contents of two registers. Pick the contents of the two registers so that using *mov* to set the contents result in an *invalid constant after fixup error*.

[10] RAM in the Cortex-M4 begins at 0x20000000

[11] Hint: *ldrb* and *strb* instructions are the versions of *ldr* and *str* that work on bytes

6. Translate the program explained in Programming Fibonacci on a breadboard computer to Cortex-M4 assembly.

7. For the string concatenation program above, create a depiction similar to the one in Figure 2.5. Show the contents per group of one byte, though, not per group of four bytes[12].

[12] To solve this exercise you will need to determine the endianness of the Cortex-M4. An option is to use the debugger to inspect memory.

Chapter 3

Memory

On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" In one case a member of the Upper, and in the other a member of the Lower House put this question. I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage

3.1 Memory Organization

In computer systems data and instructions are transferred between CPU and memory via one or more communication buses[1]. When an architecture defines one single memory (and bus) for both instructions and

[1] A bus is simply the wires that connect components in a computer. For instance, the wires that connect the CPU and memory.

data, it is referred to as *Von Neumann architecture*. Conversely, when an architecture defines two separate memories and buses, one memory and bus for instructions, and another separate memory and bus for data, it is referred to as *Harvard architecture*. From a programmer's perspective, the former architecture defines one single address space, shared by both code and data, while the latter one may define one or more address spaces.

The Cortex-M4's architecture is what is known as **modified Harvard**, which can be thought of as a modern Harvard architecture. Figure 3.1 shows a simplified schematic of the **CPU-memory interconnection**. The *System Bus* allows for both fetching of instructions and data access from SRAM, IO, and external RAM. This configuration by itself represents a Von Neumann architecture, and is no different from what is typically found in MPU-based computers. On the other hand, MCUs typically have a separate **program memory** where instructions are stored. This program memory is implemented as on-chip *Flash* and instructions are fetched from it via the *I-Code Bus*. Moreover, those instruction can be treated as data[2] and be accessed via the *D-Code Bus*. Having separate buses for instructions and constant data (*I/D-Code*), and data (*System Bus*) is desirable as it allows for simultaneous access, which may result in increased memory bandwidth.

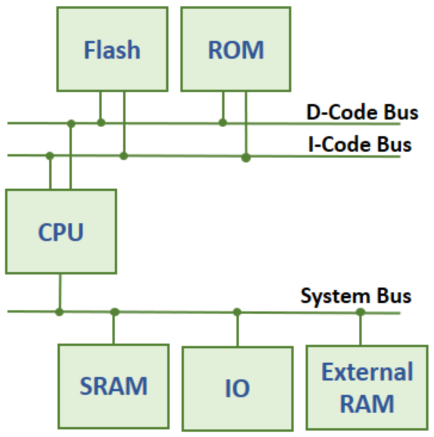


Figure 3.1: Cortex-M4 CPU-Memory interconnection

An abstraction over the memory interconnection depicted above is the **memory map**. The Cortex-M4 ATSAM4SD32C memory map defines one address space split into **regions**, as shown in Figure 3.2. Notice not all

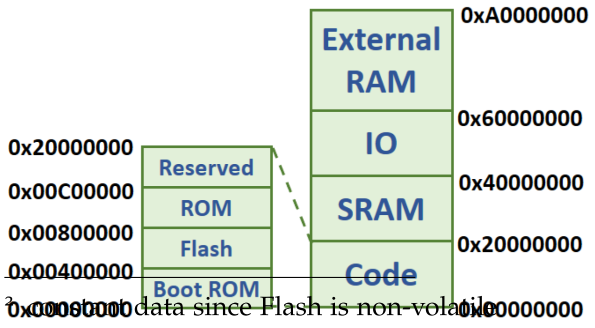


Figure 3.2: ATSAM4SD232C Memory Map

[2] constant data since Flash is non-volatile

memory specified in it is actually implemented. From the 500 MB of addressable SRAM specified in the layout (0x20000000 to 0x3fffffff) only 160 KB are physically present. This is because regions are specified as part of CPU designs, but actual amount of memory implemented is part of computer (or in this case MCU) designs.

The next section discusses how memory's address space is further divided in segments or sections.

3.2 Memory Segmentation

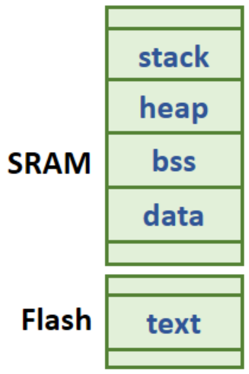


Figure 3.3: Example of segmented memory

For programming convenience, memory regions are split into **segments** or **sections** (Figure 3.3). Although they are arbitrary, compilers typically assume certain de facto segments such as text, data, bss, stack, and heap. In particular, C compilers will place code in the text segment, initialized static variables in the data segment, uninitialized static variables in the bss segment, the stack in the stack segment, and the heap in the heap segment.

This additional indirection is convenient because it allows programmers and compilers to disregard the exact memory addresses of instructions and data. Programmers writing assembly or compilers translating to assembly are only responsible for specifying what goes in what segment. The mapping of segments to physical memory is the linker's responsibility, and we discuss it in the following section.

3.3 Linker

To map segments into memory addresses, knowledge of the memory map is required by the toolchain. More specifically, this information is passed to the linker in the form of a **linker script**.

A software development toolchain is the set of tools used to transform source code into executables and libraries. They are composed of a compiler, an assembler, a linker, among other tools. To better understand the role of the linker in the process of creating an executable (Figure 3.4) let us look at an example.

Consider a project consisting of three files: *main.c* (a C file), *interrupts.s* (an assembly file), and *memory.ld* (a linker command file). When the project is built, first the compiler runs and translates *main.c* into *main.o*[3] (object file). Similarly, the assembler runs and translates *interrupts.s* to *interrupts.o*. The two generated object files contain **relocatable machine code** i.e. machine code containing relative addresses, hence not ready to be executed. Then the linker takes both object files and combines them into a single executable file (.elf, .exe, or other). To combine them the linker must a) perform symbol resolution (replace all references to symbols[4], across different object files, for actual addresses in memory); and b) replace relative memory addresses by the actual addresses they will have in the target machine. This is because compilers and assemblers do not have knowledge neither of other object files (except for the one they are generating), nor of the memory layout, and therefore generate isolated code starting at address 0. It is the linker's tasks to put it all together with finalized memory addresses ready to be loaded into memory and executed.

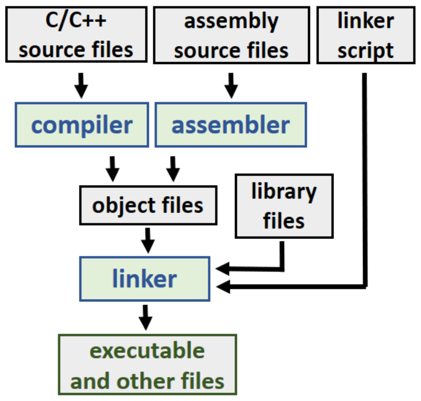


Figure 3.4: The process of generating an executable

This process is demonstrated with an example in Memory—Object File and Executables.

[3] Some C compilers may first compile to assembly

[4] e.g., labels in assembly, or function names and variables in C.

3.4 Linker Scripts

The linker script is a file containing the target machine's memory layout. Listing 3.1 shows a basic linker script, and the one used thus far. Although there is no need to be an expert on linker scripting, some basics are necessary when dealing directly with the machine.

```
...
MEMORY
{
  rom (rx)  : ORIGIN = 0x00400000, LENGTH = 0x00200000
  ram (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00028000
}

SECTIONS
{
    .text :
    {
        . = ALIGN(4);
        KEEP(*(.vectors))
        *(.text )

    } > rom

    .data :
    {
        . = ALIGN(4);
        *(.data);

    } > ram
}
```

Listing 3.1: Basic linker script

In Listing 3.1 the MEMORY directive declares two memory regions: rom and ram. rom, with rx (reading and executing) permissions, corresponds to the MCU's internal Flash. ram, with rwx (reading, writing, and executing) permissions, corresponds to the MCU's internal SRAM. ORIGIN specifies the start address for a given memory region, while LENGTH specifies its available physical memory in bytes. This information is detailed in the document Atmel SAM4S Series Datasheet.

The SECTIONS directive is used to create output[5] memory sections. When declaring a section, one has to declare one output section, as well as all the input sections that will be included in it. The linker file in

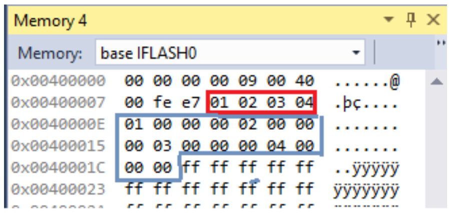
[5] Output sections are those in the final executable. Input sections are those in object files. **They are input and output from the linkers perspective**.

Listing 3.1 defines a *.text* output section in rom. This is specified with the expression *.text:* {...} >*rom*. At the same time, it specifies that the *.text* output section be composed of all *.vector* sections in all input object files, and all *.text* sections in all input object files. This is due to the expressions **(.vectors)* and **(.text)* respectively. "*" character means all object files; instead one could simply specify the object file names, e.g., *main.o(.text)*.

The order in which output sections and input sections appear in the script matters. All sections will appear contiguously in memory, and those appearing first (top to bottom) will have a lower memory address. For example, input *.vector* has a lower memory address than input *.text*. More specifically, the first byte of input *.vector* will appear at 0x00400000, while the address of the first byte of input *.text* will depend on what the size of *.vector* is.

KEEP is used to tell the linker that a specific input section must be included in the output section, even if not referenced from other source files. To save space, the linker does not include in the executable any data that is not referenced in code (unreachable data). In the above linker script, KEEP is specifically used to mark the entry point of the executable (the reset vector). If not explicitly told, the linker will optimize away the entire vector table in *.vectors*, since the vector table is never reference from code.

The line . = *ALIGN(4);* forces the first byte of the section that follows (i.e. visually below) to be word-aligned or 4-bytes aligned. This means the address in memory of the input section specified right after an ALIGN command will be multiple of 4. In general, a memory address is said to be *n*-byte aligned when it is multiple of *n* bytes, and *n* is a power of two. This is required as some instructions may generate an alignment fault if access to memory is not aligned.



```
0x00400000  00 00 00 00 09 00 40   ......@
0x00400007  00 fe e7 01 02 03 04   .þç....
0x0040000E  01 00 00 00 02 00 00   .......
0x00400015  00 03 00 00 00 04 00   .......
0x0040001C  00 00 ff ff ff ff ff   ..ÿÿÿÿÿ
0x00400023  ff ff ff ff ff ff ff   ÿÿÿÿÿÿÿ
```

Figure 3.5: Array of bytes and words in memory

Additional Video Material[6]: Memory—Custom Input Section and Alignment, Memory—Misaligned Memory Access.

[6] Some content of this chapter assumes knowledge of the vector table. A complete

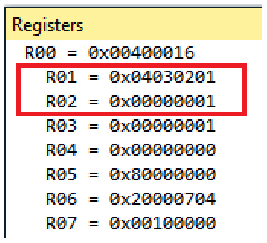
3.5 Variables

Unlike higher level languages, assembly does not have variables per se. Labels (i.e. addresses) can be treated as variables. Consider the program in Listing 3.2, where there is an array of bytes and an array of words being placed in the text segment, *a* and *b* respectively. After flashing the MCU, the program memory holds the two arrays as shown in Figure 3.5.

```
...
.section .text
.thumb_func
start:
   b .

.section .text
a: .byte 1, 2, 3, 4 /* array */
b: .word 1, 2, 3, 4 /* array */
```

Listing 3.2: Data in .text

However, one has to be careful as **labels are simply names for the address of first byte of each array**. Listing 3.3 demonstrates this, and it also shows how thinking of addresses as variables can lead to a bug. When the program in Listing 3.3 is executed, the value 0x04030201 is loaded into r1, while the value 0x00000001 is loaded into r2 (Figure 3.6). This is because **labels have no type, and thus there is no size associated with them**. LDR will load a word starting at the specified address, regardless of its contents. To load only one byte it is possible to replace LDR for LDRB (B stands for byte).



```
Registers
 R00 = 0x00400016
   R01 = 0x04030201
   R02 = 0x00000001
   R03 = 0x00000001
   R04 = 0x00000000
   R05 = 0x80000000
   R06 = 0x20000704
   R07 = 0x00100000
```

Figure 3.6: LDR on array of bytes and words

```
...
.section .text
.thumb_func
start:
   ldr r0, =a
   ldr r1, [r0]   /* r1 = a[0] */
   ldr r0, =b
   ldr r2, [r0]   /* r2 = b[0] */
   b  .
```

understanding of the vector table is in the Chapter Interrupts. Meanwhile, here is a short introduction: Memory—The Vector Table (MISSING. TO DO).

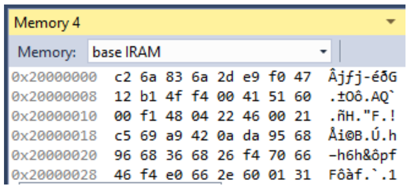
```

.section .text
a: .byte    1, 2, 3, 4 /* array */
b: .word    1, 2, 3, 4 /* array */
```

Listing 3.3: Loading the first element of two arrays

Labels are not variables. Many times it is convenient, however, to treat them as such when they hold addresses for data, as long one is aware they are not.

Next it is illustrated the use of RAM to hold data.

3.6 Data in RAM



```
0x20000000  c2 6a 83 6a 2d e9 f0 47  Âjƒj-éðG
0x20000008  12 b1 4f f4 00 41 51 60  .±Oô.AQ`
0x20000010  00 f1 48 04 22 46 00 21  .ñH."F.!
0x20000018  c5 69 a9 42 0a da 95 68  Åi©B.Ú.h
0x20000020  96 68 36 68 26 f4 70 66  –h6h&ôpf
0x20000028  46 f4 e0 66 2e 60 01 31  Fôàf.`.1
```

Figure 3.7: Empty RAM

Consider the program in Listing 3.4, which specifies a string literal must be placed in the data section (SRAM as per the linker script). After compiling this program, one would expect to find all the characters from *msg* at address 0x20000000. However, this is not the case (Figure 3.7). The question is, then, why is the linker failing to place data where instructed to?

```
...
.section .text
.thumb_func
start:
   b .

.section .data
msg: .ascii "Hola Mariana. ¿Quieres café?" /*data in SRAM*/
```

Listing 3.4: Placing data in RAM

The linker cannot place data in RAM—RAM is volatile. Yes, the executable (built by the linker) contains information of the data that must live in RAM, but the linker is a "compilation" tool, and cannot place data in SRAM at compile time. For an application running on an OS environment, the operating system's loader is responsible for loading an executable into RAM. Every time an application is executed, the loader reads the program image from permanent storage, and loads the code and data in their respective memory segments in RAM where it is subsequently executed. Even in the lack of an operating system, there has to be a loader responsible for moving code and data from permanent storage into RAM.

In the scenario depicted above, the need for a loader is obvious. However, on MCUs, the situation is slightly different. While **data in RAM does need to be loaded, code does not**. This is because during the flashing process, *.text* is placed in program memory (permanent storage) from where it is directly executed. This is why there has not been any issues executing code without a loader—until now.

How is data loaded into RAM? It must be manually loaded on power up, in the same way it occurs in a typical MPU-based environment. Take in consideration, though, that not all data goes in RAM; data such as constants can too be stored in the text section.

3.7 Data Loader

Code that set ups *.data* and other segments in RAM must have a) knowledge of where in permanent storage[7] they are, as well as their size; and b) the target location in RAM. To this end linker script variables are added to the current linker script (Listing 3.5).

```
...
    .text :
    {
        . = ALIGN(4);
        KEEP(*(.vectors))
        *(.text )

    } > rom
   . = ALIGN(4);
    _etext = .;

    .data : AT( _etext)
    {
        . = ALIGN(4);
```

[7] Data that will end up in RAM has to be stored somewhere. In the case of a MCU, data is stored somewhere in program memory so that it can be loaded at runtime.

```
        _sdata = .;
        *(.data )
        . = ALIGN(4);
        _edata  = .;
    } > ram

    _data_size = _edata - _sdata;
}
```

Listing 3.5: Basic linker script

The location counter symbol "." is a variable containing the address at that specific point of the linker script. The symbol *_etext*[8], therefore, will hold the address of the next free byte after the text section, whatever that address is. Similarly *_sdata* will point to the first byte of the data section, while *_edata* will point to the next free byte after the data section, irrespective of its size. In every case, the symbol definition comes after the command . = *ALIGN(4);*. This is to enforce word alignment. Specifically, *ALIGN(4)* will do some arithmetic to assign the closest greater address that matches the requirements i.e. will add 1, or 2, or more bytes to the location counter as to match the alignment requirements.

Memory sections have two addresses: load and runtime. By default they are the same unless otherwise stated. In the case of the text section both addresses are the same. The data section, on the contrary, will have an address in program memory, known as *load address*, and one in RAM, known as *runtime address*. It is thus the loader's task to copy the data segment from its load address to its runtime address in RAM. The distinction is important, as **instructions refer to data using its runtime address**. Both addresses are specified in the linker script. While the runtime address is calculated based on the linker script's layout, the load address is set via the *AT* keyword. *AT(_etext)* is thus telling the linker to place the data segment at address *_etext* (which is Flash).

To test the new linker file we use the program in Listing 3.6. When debugging one can see msg was successfully written to its load address (Figure 3.8).

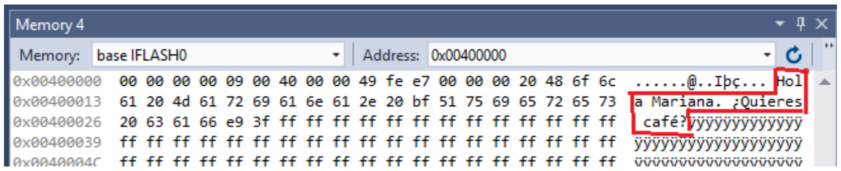
```
...
.section .text
.thumb_func
start:
    /* msg needs to be referenced or
```

[8] Linker script variables can be accessed from source files, say LDR r0, =_etext

```
      else it's not included in the elf*/
   ldr r1, =msg
   b .

.section .data
msg: .ascii "Hola Mariana. ¿Quieres café?"
```

Listing 3.6: Placing data in RAM part 2

Now that *.data* is in permanent storage, and the linker script has made available its runtime address and its size. With this information it should be straightforward to write a loader. This, however, is left as exercise.



```
0x00400000  00 00 00 00 09 00 40 00 00 49 fe e7 00 00 00 20 48 6f 6c  ......@..Iþç... Hol
0x00400013  61 20 4d 61 72 69 61 6e 61 2e 20 bf 51 75 69 65 72 65 73  a Mariana. ¿Quieres
0x00400026  20 63 61 66 e9 3f ff ff ff ff ff ff ff ff ff ff ff ff ff   café?ÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x00400039  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x0040004C  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

Figure 3.8: *msg* at the end of *.text*

Video material: Memory—Load and Runtime Addresses

Resources on the web

- (Video) IBM researchers store one bit of magnetic information in just 12 atoms
- (Video) How computer memory works - Kanawat Senanan
- (Video) Mechanical Turing Machine in Wood
- (Video) SRAM and DRAM

Exercises

The Base code for this Lab is Memory Base Code.

1. Write a loader for the SAM4S that initializes the *data* segment in memory. Test it with a program that given the string literal "¡Qué pobre memoria es aquélla que sólo funciona hacia atrás!" as a variable in RAM, turns it into upper case.

2. For the above task, show the loaded segment in memory using the debugger, and in the executable using objdump.

3. Do some investigations and extend the loader to also initialize the *bss*[9] segment. With some sample program and the debugger, demonstrate that it works.

4. Read the article Data alignment: Straighten up and fly right and explain why aligned memory access may increase performance.

5. Watch the video Lest We Remember: Cold Boot Attacks on Encryption Keys and briefly explain what it is about.

6. Read the article Historical Reflections. Actually, Turing Did Not Invent the Computer . What are your thought on it?

[9] The *bss* segment is used for holding non-initialized and zero-initalized data. Hint: NOLOAD directive. Another hint: there is no need to store content whose value is known beforehand.

Chapter 4

Input-Output

Why, sometimes I've believed as many as six impossible things before breakfast

Lewis Caroll

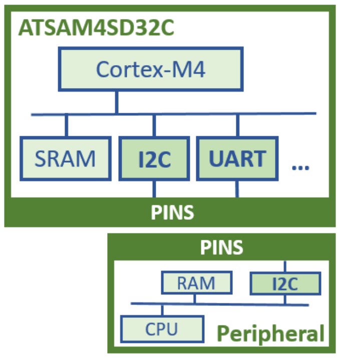


Figure 4.1: Sample MCU-to-Peripheral communication

The term input-output (IO) is used to describe the interaction between the part of a computer that computes and anything outside of it, excluding memory. To understand the need for IO, consider what would be the use of a computer without a means to interact with the outside world. Clearly, a computer without IO can be useful in itself. Imagine, for instance, a closed computing environment hosting some form of universe. Interactions within such universe have an effect (perhaps of benefit) in whatever exists in the same universe. However, for any computer to have an effect outside of it, there has to be some means of communication. In other words, a computer without the ability to cause an effect on our world, from our point of view, is as beneficial as a rock (one that gets warm).

4.1 IO devices

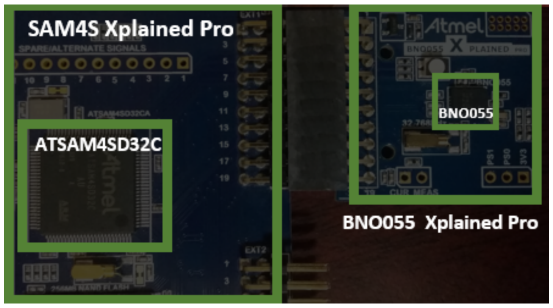


Figure 4.2: SAM4S-to-BNO055 connection

The term IO device or IO peripheral is used to denote **a physical part of a computer that does IO**. A plethora of peripherals exist: sensors, actuators, communications devices, storage devices, timers, among others. Yet from the MCU's perspective, there are only a limited number of them. That is because **peripherals either connect directly to the system bus or connect indirectly via other peripherals that connects to the system bus**. Figure 4.1 depicts an example of an IO device connected to the ATSAM4SD32C via I2C (I2C is a communication bus used in MPUs and MCUs).

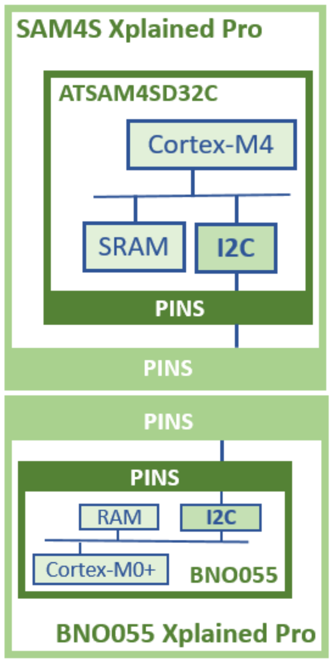


Figure 4.3: SAM4S-to-BNO055 connection

There are a few things to realize from Figure 4.1. First, some peripherals, such as I2C and UART, are peripherals built in the MCU's silicon die. Since a MCU is a computer on a chip, this configuration is analogous to IO devices built in the motherboard of, say, a laptop computer.

Second, given that built-in peripherals are accessed via the system bus, they have a range of addresses assigned (as if they were memory), and are thus referred to as **memory-mapped IO**. Specific memory addresses (say 0x400001cc) are mapped to specific registers in specific peripherals (say the data register in the I2C IO Device).

Third, **IO devices themselves are computers**. Smaller, surrogate computers, but computers nonetheless. To see how "main computers" interact with surrogate ones, consider the BNO055 Xplained Pro board in Figure 1.5b.

It plugs to the SAM4S's EXT1 port as shown in Figure 4.2, and its logical connection is depicted in Figure 4.3. The BNO055 sensor is a chip that uses a Cortex-M0+[1] for processing. Notice how the ATSAM4SD32C MCU is not plugged directly to the BNO055 sensor, but instead both are part of a board, and then those boards plug together. Except for on-board peripherals, this is the default configuration between the SAM4S Board and any other peripheral.

Before diving into how to talk to the BNO055 and other *fancy* devices. Let us first demonstrate the rather complex process of driving peripherals with the simplest of the examples—an LED. LED0, in particular (Figure 4.4).

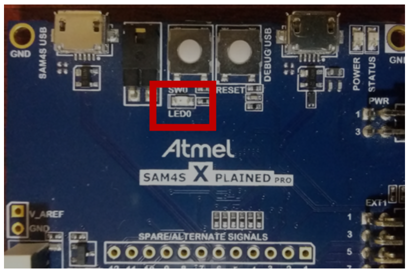


Figure 4.4: LED0

4.2 Driving an LED (example)

Unlike the BNO055, LEDs do not do local processing. They are "passive". They are simply connected to a physical pin. When the pin goes to high level the LED glows, when the pin goes to low level the LED is off.

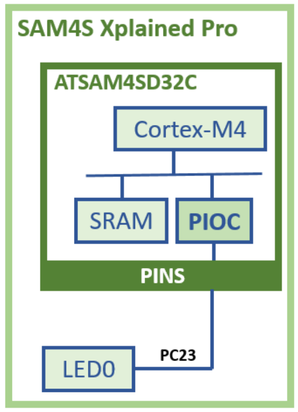


Figure 4.5: CPU to LED0 connection

We begin by investigating what physical pin drives LED0. Because that information pertains board design, it should be in the board's documentation. Namely, the SAM4S Xplained Pro User Guide. In section 4.2.5 (LED) we find that LED0 is wired to pin *PC23* in the MCU. In the MCU, all physical pins are mapped to logical IO lines. IO lines can be multiplexed to a specific built-in peripheral (e.g. I2C, USB, Ethernet). Here the important part is to find out the internal peripheral that controls *PC23*. The internals of the ATSAM4SD32C MCU are detailed in the document SAM4S Series Datasheet. In that document, (table 3.13, pg 14) we find the on-chip peripheral

[1] A smaller sibling of the Cortex-M4

responsible for driving *PC23* high and low is the *Parallel IO Controller C*. Now we know how LED0 is reached from the CPU, as shown in Figure 4.5.

PIO Controllers can drive up to 32 IO lines each (see Section 31, page 567, of the same document). Particularly, ***PC23* is the 23rd IO line of the *PIO Controller C* (*PIOC*)**. At its most basic, IO Lines can be set, cleared, and read. Again, since LED0 is physically wired to PC23, by setting and clearing PC23 from *PIOC* we drive LED0.

Now the question is, how do we drive *PIOC*? Using its interface registers: PIO Enable Register, PIO Disable Register, PIO Status Register, and others (Section 31.6 pg 585 of the same document). Everything that *PIOC* can do is controlled from its interface registers. **Because interface registers are memory-mapped, ultimately driving *PIOC* is translated to *ldr* and *str* instructions**. The tricky part is to write the right registers, in the right order, and without missing any step. The code in Listing 4.1. shows minimal code for clearing[2] *PC23*, and thereby driving LED0 on.

```
/* Enable peripheral clock line to PIO Controller C */
ldr r0, =0x400E0410 /* PMC_PCER0 */
mov r1, #1
lsl r1, r1, #13
str r1, [r0]

/* Set PC23 direction as output */
ldr r0, =0x400E1210 /* OER */
mov r1, #1
lsl r1, r1, #23
str r1, [r0]

/* Enables writing on PC23 */
ldr r0, =0x400E12A0 /* PIO_OWER */
mov r1, #1
lsl r1, r1, #23
str r1, [r0]

/* Clear PC23 (inverted logic) */
ldr r0, =0x400E1234 /* CODR */
mov r1, #1
lsl r1, r1, #23
str r1, [r0]
```

Listing 4.1: Driving LED0

[2] Some IO lines's logic is inverted. When they are cleared, the physical pin is high and vice versa.

In the above code, the first step is to enable the clock line in the *PIO Controller C*. This line comes from the *Power Management Controller* (*PMC*) (Section 29, pg. 515), which is another peripheral. To enable the clock line we set the 13th bit in the *Peripheral Enable Clock Register 0*. To set a bit we uses the pattern *mov r1, #1* followed by a *lsl r1, #13* to make it obvious that we are setting the 13th bit. Another way would be *mov r1, #0x1000*, we save one instruction but the intention is less obvious.

The second step is to set *PC23* as output, for which we set the 23rd bit of the *Output Enable Register*. Then we continue to do the same with two more registers. The important question here is—how can one know what registers to write to, and in which order, and what other peripherals must be configured first? By reading highly technical and chaotic documentation.

For a video demo with extra details see IO Demo—Driving LED0

4.3 Device Drivers

Device drivers are software that serves as bridge between IO devices and some other piece of software. The previous example is almost a driver, but it needs some improvements. First, instead of writing *PIOC* code, every time we want to drive LED0, it is preferably to write drivers for the *PIO Controller C* i.e. **code capable of doing with the peripheral anything that can be done with it**. Then based on these lower-level drivers write drivers for other devices that are controlled through *PIO* lines, such as LEDs. For example, pio.s shows a very simple driver for the *Parallel IO Controller C*. Notice the “interface functions” of the PIO Controller have been put into routines. Listing 4.2 shows a sample program using them to drive LED0 (these are not LED drivers, but just sample code).

```
...
.equ INPUT_DIR,    1
.equ OUTPUT_DIR,   0
.equ LEVEL_HIGH,   1
.equ LEVEL_LOW,    0
...

/* init pioc */
bl pioc_init

/* configure PC23 as output */
```

```
mov r0, OUTPUT_DIR
mov r1, 23
bl pioc_dir_set

/* set PC23 high (logic is inverted) */
mov r0, LEVEL_LOW
mov r1, 23
bl pioc_level_set

...
```

Listing 4.2: Driving LED0 with PIOC drivers

Now, based on these *low-level* PIOC drivers, one could write LED drivers whose interface functions or routines are, say, *led_on* and *led_off*.

By now one thing must be clear beyond doubt—writing drivers is difficult. Writing device drivers is a tedious, error-prone, hard-to-debug, device-specific, and time-consuming task. There are a few reasons for that. Communication with a device requires knowledge of the device's domain knowledge. For instance, writing drivers for a 802.11 wireless card requires certain degree of expertise in the 802.11 protocol. That by itself can take months to obtain. It also, requires knowledge of the interface in use. If we were to write drivers for the BNO055 we would need some basic knowledge and devices drivers[3] for I2C. Because drivers must be written in some language, certain fluency in the implementation language (customarily C or assembly) is also a must. Writing drivers while learning a language is not a good idea. Not in this case, but if the drivers were to be integrated to an OS, then certain fluency in the OS driver framework (e.g. Window's Kernel-Mode Driver Framework) would also be necessary. Debugging can be difficult, because at such low-level, often there are limited or no facilities that allow us to debug. Moreover, debugging breaks the time requirements of some protocols, and may make debugging impossible. Say, for example, attempting to breakpoint in the middle of a USB transaction would be possible, but depending on where the breakpoint is, the transaction may not be able resume after the breakpoint. Also, peripheral documentation explaining what registers controls what[4] are lengthy and often unintelligible. It is extremely easy to miss a detail. Lastly, after all the struggle of writing a driver one realizes that those drivers only work for a specific device on a

[3] I2C is also a peripheral. In this case, I2C drivers are typically known as low-level drivers, and the BNO055 drivers are higher-level drivers

[4] e.g. the *value* register contains the value of the last sensor reading

specific hardware and a specific software platform.

Luckily, except for those willing to torture themselves voluntarily (hats off to them), the reader may never have to write a single driver in her life. Still, the process of writing the simplest possible driver was demonstrated. That is, with the hope of providing a better understanding of how they *drive* hardware in reality.

4.4 Third-party drivers

Third-party drivers are the solution to the problem of having to write our own drivers. This is part of the success of open source platforms like Arduino that allow to reuse code. The SAM4S does not have the community support of Arduino, but it has a manufacturer-supported software platform called the Atmel Software Framework, which is introduced in Appendix A.

Web Resources

- (Video) CACM Jan. 2017 - Computing History Beyond the U.K. and U.S.
- Holographic Displays Coming to Smartphones.
- (Video) Holographic Acoustic Elements for Manipulation of Levitated Objects.
- (Video) Manipulation of levitated objects using holographic acoustic elements.

Exercises

IO is the base code for this chapter.

1. Using pio.s as base, write LED drivers (leds.s) capable of controlling LED1, LED2, and LED3 in the OLED1 Xplained Pro Board.

2. Use the new functionality in *led.s* to write a program that blinks LED1, LED2, and LED3. (Delays in assembly are loops that waste cycles)
3. Create a new C ASF project and write a small single program demonstrating the use of two peripheral of your choice.
4. Read the article A DSL Approach to Improve Productivity and Safety in Device Drivers Development. Explain their work.
5. Investigate about Alan Kay's *Dynabook*[5] and Douglas Engelbart's *The mother of all demos*[6]. What are your thoughts on it?

Project Ideas:

1. The idea of generating random numbers from a deterministic device can be interesting. Use IO data to write a random number generator function. For more information on *randomness* see here: here and here.
2. Use the bno055 Xplained Pro board to build a rover capable of moving in straight lines, in any direction (say 60 degrees).
3. SAM4S boards can be set to talk to each other (see here). Use that ability to make a demo of the Trickle algorithm on a SAM4S-Xbee network. Something like this.

[5] Try: Alan Kay's Dynabook – Rare NHK video

[6] Try: The Mother of All Demos, presented by Douglas Engelbart (1968)

Chapter 5

Stack

Low-level programming is
good for the programmer's
soul

John Carmack

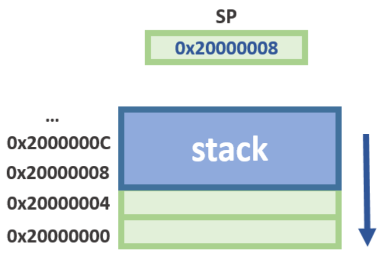


Figure 5.1: Stack grows downwards

A stack is an abstract data type with two operations: push and pop. Push adds one element to the stack, pop removes one from it. Operations are performed in a way that, given a stack in state S, a push operation followed by a pop operation will leave the stack in the same state S.

Visually, push adds an element to the top of the stack, and pop removes an element also from the top. Data is thus added and removed in a last-in-first-out (LIFO) manner. Except, perhaps, for a research prototype, all modern architectures have built-in support for one or more stacks. To relate this with previous knowledge on data structures, think of how a stack can be implemented as a linked list, or an array, or even two queues. In the machine's case it is implemented as raw memory bytes.

Like any abstract concept, there is gap between its theory and implementation. With a hardware stack, it is a considerable gap, and one has to be careful, when reasoning about it, to keep the implementation details in mind. Specifically, a stack as such does not exist in the architecture. We limit ourselves to utilize the facilities provided to ”pretend” there is one. The author uses the word pretend—since memory locations are never completely abstracted as stack elements. General memory instructions still can be applied to them, and so we are forced to keep memory details present.

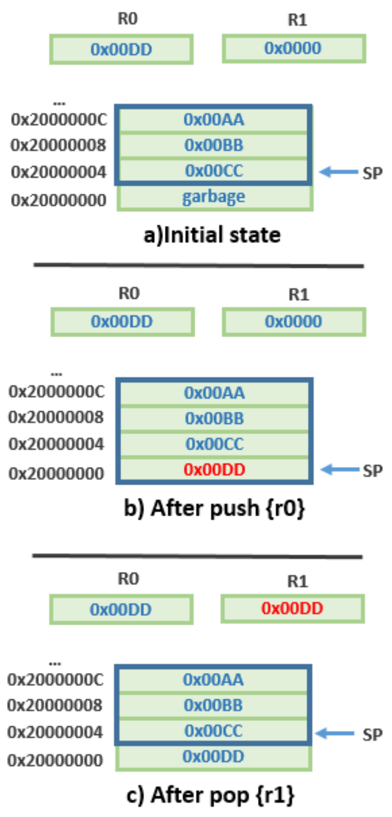


Figure 5.2: Stack use example

Facilities vary among architectures, but they typically include a stack pointer register, and special instructions to push and pop element in and out.

5.1 Cortex-M4 Stack

The stack pointer register (*SP*) is a special purpose register pointing to the top of the stack. Visually we think of the stack as growing upwards (as a pile of plates), but in the Cortex-M4, like in many other architectures, the stack grows downwards. In other words, the more data the stack has, the lower the memory address the stack pointer holds. This is depicted in Figure 5.1.

(Keeping this confusing-non-intuitive-against-gravity way of visualizing the stack, let us continue.)

A push instruction, with syntax *push reg*[1], moves *SP* down one word, then stores the value in *reg* to the memory location pointed by *SP*. Contrariwise, a *pop* instruction, with syntax *pop reg*[2], loads *reg* with the value stored in

[1] The real syntax is push *reglist*, where *reglist* is a list of registers; yet for simplicity the case of a single register is considered.

[2] The real syntax is *pop reglist*, where *reglist* is a list of registers; yet for simplicity the case of a single register is considered.

the memory address pointed by *SP*, then it moves *SP* one word up, thus effectively ”removing” the element previously pointed by SP.

Consider a sample scenario, where a stack starting at address 0x2000000c holds three words as shown in Figure 5.2a. After executing *push r0*, the stack grows in one element now holding 0x00dd at the top (Figure 5.2b). Then the instruction *pop r1* is executed, loading 0x00dd into *r1* and removing it from the top of the stack (Figure 5.2c).

Now, where is the stack in memory? Again, the machine does not provide a stack as such. It is just a set of contiguous memory locations, which we have agreed with ourselves to treat as stack. We do not need to, but to make this agreement more difficult to violate, we define a dedicated memory segment.

5.2 The stack segment

The stack segment is no different from other memory segments. It is simply an output section defined in the linker script, as demonstrated in Listing 5.1 (where 16 KB of SRAM are being allocated). The result is the memory layout in Figure 3.3, except for the lack of a heap.

```
__stack_size__ = 0x4000; /*16KB*/

...
SECTIONS
{
    ...
    .stack (NOLOAD):
    {
        . = ALIGN(8);
         _sstack = .;
        . = . + __stack_size__;
        . = ALIGN(8);
        _estack = .;
    } > ram
}
```

Listing 5.1: Creating a stack segment

Then all is left to do is initialize SP to hold the address *_estack*. Why *_estack* and not *_sstack*? Answer is left as exercise.

5.3 Initializing SP

The processor initializes SP automatically taking the value from the stack pointer vector in the vector table, at reset time. Namely, the first vector (offset 0) in the vector table is dedicated to hold the initial stack pointer. We thus modify our vector table as shown in Listing Figure 5.2.

```
.syntax unified

.section .vectors
.word _estack   /*1st entry*/
.word main      /*2nd entry*/
```

Listing 5.2: Setting SP's initial value

Upon completion the stack should be ready for use.

Video material: Stack—Basic stack usage demo.

5.4 Procedure Calls

The main purpose of the stack is to support calls to subroutines. A subroutine or procedure call is simply a set of contiguous instructions in memory—indistinguishable from data and instructions from other subroutines. Like with variables, we may define a label to "name" a subroutine. More precisely, we define a label to name the first byte of the first instruction of what we consider to be a subroutine. Then to transfer execution to it we use branch instructions. (See example in Listing 5.3).

```
...
main:
    b my_subroutine1 /* call my_subroutine1 */
    b my_subroutine2 /* call my_subroutine2 */
    b .
my_subroutine1:
    ...
my_subroutine2:
    ...
```

Listing 5.3: Branch to a subroutine

The above example shows how execution is transfer from *main* (the caller) to *my_subroutine1* and *my_subroutine2* (the callees). What it does not show is that callees need to eventually transfer execution back to the caller. Architectures typically support the need for return with special branch instructions. The Cortex-M4 provides the instruction branch with link (*bl*). *bl* does the same as *b*, except the address of the next instruction, before the branch, is stored in the link register (*lr*). So, when a callee routine needs to return, the return address is available in *lr*. This is demonstrated in Listing 5.4.

```
...
main:
    bl my_subroutine1 /* call my_subroutine1 */
    bl my_subroutine2 /* call my_subroutine2 */
    b .
my_subroutine1:
    ...
    bx lr
my_subroutine2:
    ...
    bx lr
```

Listing 5.4: Branch with link to a subroutine

This works so far—but what happens to *lr* in the case of nested subroutine calls? Does each subroutine get its own *lr*? No, the contents of *lr* are overwritten and lost. The same problem occurs with other registers. Callees can modify registers that are "in use" by caller subroutines, thereby corrupting any intermediate-computation value stored in them. The stack solves this problem. Register values are temporarily stored on subroutine entry, and restored on subroutine exit. Listing 5.5 demonstrates this pattern. (*pop {r4, pc}* is equivalent to *pop {r4, lr}* followed by *bx lr*. Can the reader tell why?)

```
...
main:
    mov r4, #1      /* r0=1 */
    bl my_subroutine
    add r4, r4, #1 /* r0=2 */
    b .
my_subroutine:
    push {r4, lr}    /* store used registers */
    ...
    /* somewhere in here R4's
       content is altered     */
    ...
    pop {r4, pc}    /* restore used registers (and return) */
```

Listing 5.5: Use of the stack in a subroutine

Why are *r4* and *lr* not stored and restored by the caller instead? They could be. That would be perfectly fine. This format is the one used by gcc (GNU C Compiler) to compile functions. They are called function prologue and epilogue. In the function prologue, registers are saved in the stack. In the epilogue, the stack is restored to how it was before procedure entry.

Like this little decision there are many others involving how parameters are passed and returned, and how data types are aligned. This little decisions matter when we want to interface our code with other code; say a C function in a different source file, or a procedure in a different binary (e.g. a library). For procedure calling to be compatible, all code must agree on one single way— convention.

5.4.1 Procedure Call Conventions

Every architecture defines an application binary interface (ABI). ABIs define procedure calling conventions, sizes and alignment of data types, among other things; and they are followed by compilers and operating systems. Their purpose is to ensure compatibility among binaries.

The ARM convention for procedure calling (part of the ARM ABI) is defined in the document Procedure Call Standard for the ARM Architecture. This document states many things, but at this point we are only interested in two of them: a) parameters are passed in registers *r0*-*r3*, and if more than four parameters are needed, they can be passed in the stack; and b) on function return, a data type the size of a word or smaller is returned in *r0*.

Video Material: Stack—Procedure calls conventions demo.

5.4.2 Code type

When branching in and out of a subroutine, *pc* is loaded with an address. From this address loaded to *pc*, the least significant bit (LSb) is reserved to tell the machine the *code type*. If the address corresponds to the address of a Thumb instruction, the LSb must be set to 1. On the contrary, if the instruction is an ARM instruction, the LSb must be set to 0. In other words, the code type of a subroutine is encoded in the address given to *pc*. Since the Cortex-M4 only supports Thumb instructions, all addresses loaded to *pc* on subroutine entry, or return, will be the address in memory + 1. Setting of the LSb is handled automatically by the assembler. In particular, the assembler does it for every subroutine pre-ceded with the directive *.thumb_func*. If a subroutine is not marked as such, branching to it, or returning to it, may result in a CPU fault.

This is something to keep in mind when dynamically loading code to memory and executing it (by saying *bl code_address*). In such scenario one must manually set the LSb before branching to it.

Video Material: Stack—Procedure calls demo and code type,

5.5 Stack frames

We have already seen how *lr* and other registers are pushed onto the stack so as to save their values for later use. The return address, and other values, that are pushed onto the stack by each *procedure instance* is referred to as stack frame. On subroutine entry, a stack frame *F* is pushed, and on subroutine return *F* is popped. Because each subroutine callee deletes its own "traces" from the stack, caller subroutines can continue *where they left off* without further considerations. Moreover, it gives the impression of having dedicated stacks for each procedure instance—quite an elegant mechanism.

This simple mechanism is an idea on its own. It seamlessly enables nested procedure calls as well as re-entrant (recursive) procedures. The principle behind it is that of postponing the completion of an action until a second action is completed, and in turn postponing the completion of that second action until a third action is completed... and so on. A stack simply mimics the order of events (LIFO) in such situations.

Video material: Stack—Stack frames demo. (Question: In this demo, *main*, *f1*, and *f2*'s compiled code modifies registers *r3* and *r7*. Yet only *r7*'s value is stored and restored—why? Hint: scratch registers)

5.6 Automatic Variables

In higher-level programming languages automatic variables are those variables whose allocation and deallocation is automatic on procedure[3] entry and exit, respectively. Their place is in the stack frame of a given procedure instance, for as long as the procedure instance is active.

In the C programming language, automatic variables are all variables that are local to a function, except for static local ones (static variables live in the data segment and they survive function return). When compiled, allocation and deallocation takes place in the function's prologue and epilogue, in that order. Allocating space for them is as simple as moving *sp* to make space for them in the *active* stack frame. For instance, in a stack that grows downwards, to allocate space for a 32-bit integer we subtract space from *sp* (*sub sp, #space*).

Video Material: Stack—Automatic variable demo

5.7 Call Stack

A call stack is composed of all the stack frames existing in the stack at any given time, for some program in execution. In memory, the call stack would be all the consecutive bytes between *_estack* and *sp* at any given time.

Web Resources

- What does logic have to do with Java?
- A brief history of the stack (by Sten Henriksson)
- XKCD HeartBleed Explanation
- From Missingno to Heartbleed: Buffer Exploits and Buffer Overflows

[3] This is inaccurate, but for simplicity, and given the context of this explanation, we will use the term procedure instead of scope.

- Understanding Vulnerabilities 1: C, ASM, and Overflows: Computer Security Lectures 2014/15 S2
- Buffer Overflow Exploit Demo (SAM4S Xplained Pro + MiniOS + Xbee)

Exercises

The base code for this Chapter is in Stack.

1. Using any two peripherals, and their Atmel ASF C drivers, write a small assembly program that does something.
2. Write a recursive function[4] in C. Use the debugger to take a screen-shot of the call stack and its contents (in RAM) at the point when the call stack has the most number of stack frames, i.e., in the base case. Then do the following.
   (a) In the screen-shot identify every stack frame.
   (b) In each stack frame identify the stacked returned address and any automatic variables.
   (c) In each stacked returned address highlight the LSb. Mention its code type.

   Then take a screen-shot of the disassembly, and do the following.
   (a) Identify the prologue and epilogue.
   (b) Within the prologue identify those instructions responsible for stacking the return address, and those responsible for creating automatic variables.
   (c) Within the epilogue identify those instructions responsible for unstacking the return address, and those responsible for deallocating automatic variables.
3. Read Section 6 Outline of Logical Control of the document Proposed Electronic Calculator, by Alan Turing (48 pp.) undated*. What are your thoughts on it[5]?

[4] Choose a function that allows answering of all questions

[5] For some context see Alan Turing Proposal For 'ACE' Automatic Computing Engine

4. Explain the Heartbleed bug, and what it has to do with the stack.
5. Explain the exact mechanisms by which buffer overflows occur.

Project Ideas:

1. Write a demo of a buffer overflow exploit
2. Mimick the Heartbleed bug and write a demo of an exploit

Chapter 6

Interrupts

It was a great invention, but also a Box of Pandora.

E.W . Dijkstra

Interrupts are *CPU events* triggered as response to internal and external conditions that require attention... (I DONT LIKE THIS DEFINITION) When an interrupt is triggered, execution is transferred to its associated interrupt handler. From a high-level point of view interrupts are no different to the concept of events in programming languages. What is more important is the purpose they serve. In particular, interrupts are the mechanism whereby the CPU can respond to stimuli while passively[1] waiting for it. Interrupts can be generated by I/O devices, the CPU itself, and software being executed by the CPU.

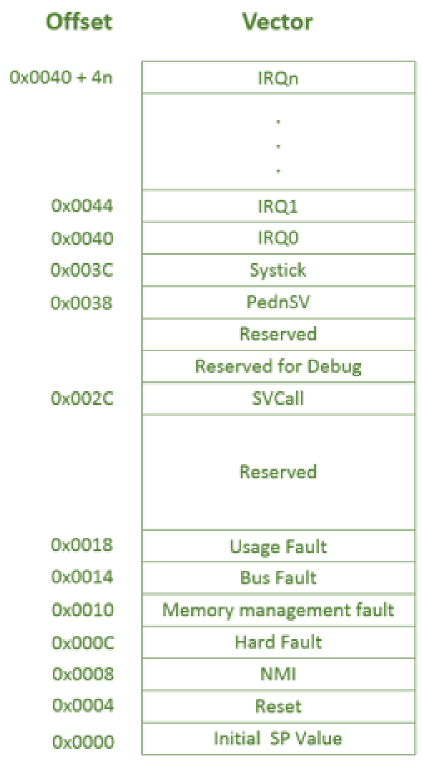
[In progress] See here

6.1 Vector Table

[In progress]

[1] As opposed to actively checking for it

6.2 Reset vector



Offset	Vector
0x0040 + 4n	IRQn
	. . .
0x0044	IRQ1
0x0040	IRQ0
0x003C	Systick
0x0038	PednSV
	Reserved
	Reserved for Debug
0x002C	SVCall
	Reserved
0x0018	Usage Fault
0x0014	Bus Fault
0x0010	Memory management fault
0x000C	Hard Fault
0x0008	NMI
0x0004	Reset
0x0000	Initial SP Value

Figure 6.1: Vector table

[In progress]The reset vector is special.... execution begins here

6.3 The Nested Vector Interrupt Controller

The Cortex-M4's exception model defines priority-based nested exceptions. In a very simplified manner, without losing resemblance to reality, this is more or less the exception model explained:

An exception waiting to be serviced by the processor is called pending. An exception being serviced by the processors is called active. When an exception is not active and not pending is said to be inactive. An exception entry occurs when there is an exception pending, and there is no active exception; or when an exception is active, and another exception of higher priority is pending, case in which the active exception is pre-empted—these exceptions are said to be nested. A return from the exception handler occurs when the exception handler is completed.

[In progress]

6.4 The Stacking/Unstacking process

When an exception is attended by the processor, the processor pushes eight data items of register size into the active stack. This is referred to as stacking, and the pushed data items are called a stack frame. In particular, registers R0-R3, R12, LR, PC, PSR, and SP (active stack pointer) are pushed on the stack as depicted in Figure 3.13. Immediately after stacking SP points to the top of the stack (which is in a lower address than it was before).

NOTE: Sometimes extra ”aligner” registers are pushed onto the stack to make the stack 8-byte aligned. This operation is handled by the compiler, however.

In regular procedure calls the link register (LR) contains the procedure’s return address, but in interrupt handlers it holds some special value with information about which CPU mode and stack pointer were active before interrupt entry, so that the processor can go back to them on exception return. In exception handlers, on the contrary, the return address is stored in the stacked program counter (PC). In parallel with stacking the processor fetches the exception handler address from the interrupt vector table, and once stacking is completed the processor starts executing the exception handler.

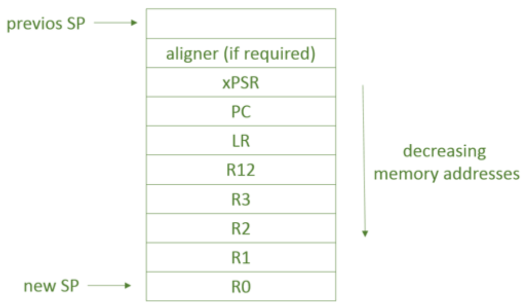


Figure 6.2: Stacked registers

[In progress]

Based on this we can write an SVC handler that can distinguish 256 different supervisor calls and is capable of receiving up to four arguments passed in the stacked registers R0-R3, and with some additional changes up to n registers.

6.5 Software Interrupts

In Cortex-M terminology they are referred to as supervisor calls (SVCs). Upon execution of an SVC instruction (in thread mode) the CPU is automatically switched into privileged handler mode and the SVC exception handler takes control of the machine. It is from this one point (the SVC handler) where the OS runs with full access to the computer. Eventually control of the machine is returned back to the application by returning from the SVC exception handler. In other words, all the process of attending user code's requests occurs within an exception handler entry and return. In fact, handler mode receive its name from the peculiarity that this mode is only active within exception handlers; except for the reset handler, as we have already seen.

The Cortex-M4 instruction set provides a SVC instruction with the syntax SVC #imm, where imm is an 8-bit immediate value used to distinguish one supervisor call from another. A valid SVC call is, for instance: SVC #0xAC. Once in the SVC Handler, finding out the immediate value used to make the supervisor call requires reading the SVC instruction itself (the one responsible for the exception) from its location somewhere either in code section or ramfunc section. We can then specify a number with each supervisor call, but what if we need to pass arguments? To pass arguments we take advantage of an operation called stacking, which we describe next.

[In progress...]

- Breaking the x86 Instruction Set

Exercises

Part II

System

Chapter 7

Introduction

What I cannot create, I do not understand

Richard Feynman

7.1 Accessing IO Peripherals

To access IO peripherals we need software that knows how to talk to these peripherals. This is why OSes are shipped with their own device drivers. In fact, as much as 70% of the code of a typical OS is driver code. Of our special interest is, however, that at boot time, before the OS has been loaded into memory, there is a need for accessing IO. **This is an egg-chicken type of problem, for how can the OS image (which includes IO drivers) be read from disk without disk drivers?** It is not only a matter of disk drives. Often there are other type of operations that may be needed before the OS is loaded into memory. Printing to the screen or initializing the keyboard, for instance.

Because of this problem, **computer systems typically have some form of firmware[1] for "basic" IO access**. An example of this is the BIOS (or

[1] Firmware is the name given to software that is stored in non-volatile memory and is executed directly from there itself. Unlike other software, firmware is not intended to be replaced, hence the term *firm*; e.g., printer firmware is not intended to be replaced, except perhaps for more printer software (i.e. an update).

the more recent UEFI), present in PC-compatible computers. The functionality offered by these firmware IO subsystems is somewhat limited (i.e. basic), yet there are OSes which rely uniquely in it. Meaning OSes that do not have their own drivers. In our case we want something similar: **we want (and need) some form of third-party IO subsystem (or something) that will free us from having to write device driver code**[2].

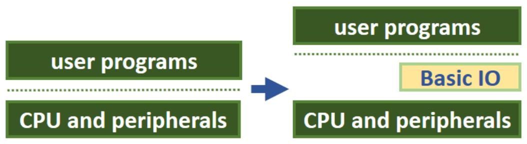


Figure 7.1: Architecture goal for this this chapter

Now, the SAM4S, being a MCU platform, is different from general purpose computers. There is no additional chip present in the board containing driver support as firmware. That is, the SAM4S has no BIOS, nor anything similar. Moreover, **technically speaking, software running on the SAM4S is firmware**, since it is stored and executed from non-volatile memory (see chapter 3—Memory). Luckily, another difference is that MCU vendors (Atmel in this case) offer device driver support for their boards. In particular, Atmel offers open source firmware as part of their Atmel Software Framework (ASF). ASF code can be more or less mechanically integrated into Atmel Studio projects. **So for MiniOS we will use Atmel's ASF for Basic IO purposes**. It is up to the reader to judge how "basic" the ASF really is. From an architecture perspective, this chapter is concerned with the transformation in Figure 7.1.

Booting in a MCU is a straightforward process, and is detailed next.

7.2 Booting

A bootloader is the name given to the piece of software in charge of loading the OS from where it is stored, say disk, to where it can be executed (i.e., memory). Yet again a MCU is different. Namely, there is no bootloader in a MCU. Why? because **OS code can be executed directly from where it is stored: *program memory***. Program memory is non-volatile Flash memory and is known as *internal Flash*.

[2] The author has nothing against writing device drivers per se, he is simply slightly traumatized after spending part of his youth writing drivers for no pay in a dungeon.

Since OS code is already ”loaded”, the only thing left to know is how to execute it. When the SAM4S is powered up, a *reset exception* is triggered. In response, the CPU will automatically load the program counter (PC) with the address in the *reset vector* (chapter 6–Interrupts) and begin execution. From a code point of view, this means **the reset handler is the execution entry point for the machine**. This is, of course, provided the vector table has been defined adequately.

Once in the reset handler, a software loader known as *start up code* (see chapter 3—Memory) initializes CPU and memory, and then calls *main()*. ***main()* is thus the entry point for programs, and where OS code will eventually be placed**.

While the vector table and start up code can be written manually, the ASF provides templates with vector table definitions and start up code, in addition to a linker script.

For complete details on how to create ASF template projects see Appendix A.

Web Resources

- ARM Cortex M3/M4 Processor Reset Sequence
- A software design process for the Atmel Software Framework
- Lecture 13: Booting Process
- Computer Boot Process animation
- Linus Torvalds - Nvidia F_ck You!

Exercises

If this is the reader’s first encounter with the SAM4S platform, it is strongly recommended to go through chapter 1 (Introduction), and chapter 4 (IO).

1. Create a new C ASF project and write a small single program demonstrating the use of two peripheral of your choice.

Chapter 8

Hardware Abstraction Layer

Computer Science is a science of abstraction—creating the right model for a problem and devising the appropriate mechanizable techniques to solve it.

Alfred Aho

Very simplistically, systems are a *collection of coexisting parts*, from which some overall behaviour arises. This behaviour is the result from the interactions of the parts, and it is important to notice that the behaviour of the parts by themselves is very different from that of the whole. From this follows that **a software system is a collection of coexisting *abstract parts*, from which some overall behaviour arises**. When it comes to operating systems, it is more or less well established what these abstract parts are[1]: file system, scheduler, console, and others the reader is sure familiar with.

The first abstract part to be implemented in this chapter is the hardware abstraction layer (HAL). **The idea behind the HAL is to hide away**

[1] Not because the establishment is the best suited (whatever best means), but simply because of how operating systems happened to be conceived in the early days of computing; together with the *incremental* nature of how changes take place in human endeavours.

many of the details of the machine and bare-metal drivers from upper layers. The HAL thus serves as a foundation for the rest of the system, and makes upper layers more platform independent that they would be otherwise. In terms of architecture the goal for this chapter is the transformation depicted in Figure 8.1.

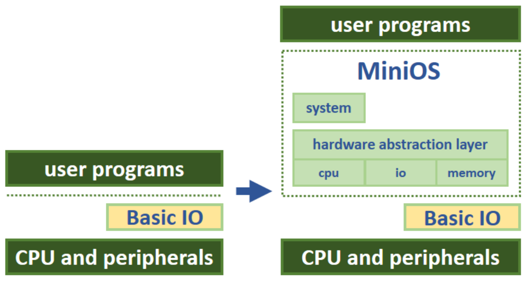


Figure 8.1: Architecture goal for this this chapter

In other words, we want to go from writing programs that run on the bare machine to writing programs that run on the HAL. To show this more concretely, let us begin with a simple example.

8.1 LED Example

Let us say we want to expose upper layers the ability to control LEDs (LED1, LED2, and LED3). With the current functionality from Basic IO, we would need to use the code shown in Listing 8.1.

Listing 8.1: Turning LEDs on via Basic IO

```
#include <asf.h>

#define LED1 IOPORT_CREATE_PIN( PIOC, 20 )
#define LED2 IOPORT_CREATE_PIN( PIOA, 16 )
#define LED3 IOPORT_CREATE_PIN( PIOC, 22 )

int main( void ){
        sysclk_init();
        board_init();
        ioport_init();

        //set pins directions
        ioport_set_pin_dir( LED1, IOPORT_DIR_OUTPUT );
        ioport_set_pin_dir( LED2, IOPORT_DIR_OUTPUT );
        ioport_set_pin_dir( LED3, IOPORT_DIR_OUTPUT );
```

```
    //set level high
    ioport_set_pin_level( LED1, false );
    ioport_set_pin_level( LED2, false );
    ioport_set_pin_level( LED3, false );
}
```

This code is hardware- and driver-specific, and uppers layers are unnecessarily exposed to Parallel IO controller concepts, such as pin number, pin direction, and pin level. Instead we would like to write the HAL so that it supports the code in Listing 8.2

Listing 8.2: Desired LED abstraction

```
int main( void ){
        led_set( Led1, LedOn );
        led_set( Led2, LedOn );
        led_set( Led3, LedOn );
}
```

This second piece of code is shorter, simpler, and enables reasoning in terms of actual LEDs. Moreover, as simple as it may seem, it makes upper layers, to an extent, hardware-agnostic. Should the application in Listing 8.2 be ported to, say, an Arduino platform, it would suffice to rewrite *led_set*, while the rest of the code remains intact. Clearly this is a trivial example, but as the system grows the benefits will become more evident.

As mentioned above, writing an OS can be quite challenging. In an attempt to help this, we follow a few specific coding practices that are outline next.

8.2 Coding practices

With the purpose of keeping code manageable and preventing complexity from exploding, we use a very specific set of coding practices.

First, non-static[2] global variables within a module are discouraged. It is difficult to keep track of external modifications to non-static variables

[2] *Non-static* variables can be accessed from outside the file they were declared in (using the keyword *extern*), whereas *static* ones cannot be.

in a module, and as such is a source of bugs. Instead modules have an *interface* and interaction among modules occur only via interfaces.

Code is organized in C modules[3], which have a one-to-one correspondence to modules in the architecture. If the architecture shows a module *M*, then there exists an *M* module in code. Every module has a set of non-static functions that serve as interface to the module. The header file for a module includes data types, constants, and function prototypes for a given module.

No hard-coded values. The larger the code gets the more difficult it becomes to keep track of all the places where a certain value was hard coded—the search functionality of the IDE is no replacement! So hard coded values are another source of bugs.

When a known data structure or algorithm is to be used (e.g. tree traversal), it is better to use a known algorithm and document which one. Often code is difficult to understand, but if the code is simply translating a known pseudocode or textbook version this becomes much simpler.

The naming conventions are: (a) functions of a module *module* begin their name with *module_*; (b) function names and variable names are all lower case and words are separated by low dash "_", e.g. *this_is_a_function()*, *this_is_a_variable*; (c) constants declared with the *const* keyword and/or part of enumerations are written in upper camel case, e.g. *ThisIsAConstantVar*; (d) defined data type names follow the same convention as constants, except they start with the lowercase letter "t", e.g. *tThisIsAType*.

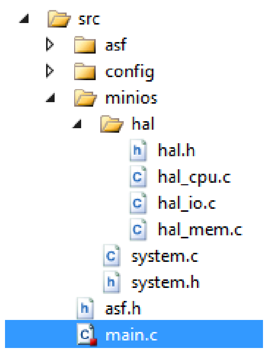


Figure 8.2: Source code organization for HAL

Every module will have an *init* funtion. So the module *console* will have, for instance, a *console_init()* function which initializes the console.

For purposes of portability, standard data types, such as *uint8_t*), are used instead of their platform-specific[4] counterparts, such as *int*.

[3] A module is composed of a *.c* file and a header (*.h*) file

[4] Different platforms may have *int* types of different sizes. For example, an *int* could be a *32-bit int* in some platform, and a *16-bit int* in some other. This can lead to disastrous bugs. Really it can!

Last but not least, choose good names. *tree3* is not a good name for a tree, specially not if there is no *tree1* and *tree2*!

Why bother so much? Well, certainly exposure to good coding practices is desirable in itself. That aside, we do it with the purpose of keeping complexity under control (or, at least, trying). That or the control freak nature of any kernel developer (the author included). With this in mind, let us continue with the LED example.

8.3 LED Example (continued)

Now we need to write the code that will enable us to execute the LED application shown in Listing 8.2. The first step is to organize code as to match the HAL architecture (Figure 8.1). So we add a *System* module and three HAL modules: *hal io*, *hal cpu*, and *hal mem*. The resulting file organization appears on Figure 8.2 (notice how all HAL modules share the same header file *hal.h*).

The System module defines all functions related to the system, such as, system clock and board initialization. HAL modules, on the other hand, define all the functions that access hardware and Basic IO. In the particular case of LEDs there is really no functionality to add, besides what Basic IO already has, so **it is mostly wrapping the already existing functionality**, as one can see in listings 8.3 to 8.7.

Listing 8.3: HAL IO module C file

```c
/**
 * @file         hal_io.c
 * @author
 * @version
 *
 * @brief IO part of the Hardware Abstraction Layer
 */

#include <asf.h>
#include 'hal.h'

void hal_io_init(void){
    //Whatever needs to be initalized from lower layers
    //goes here
        sysclk_init();  //Sysclk is technically HAL CPU (move later).
        board_init();
        ioport_init();
}
```

```
void hal_led_start(void){
        //LEDs specific initializations.
        //No need to configure anything...
        //If GPIO is configured so are LEDs

        //set LEDs pins directions
        ioport_set_pin_dir( IOPORT_CREATE_PIN(PIOC, 20),
            IOPORT_DIR_OUTPUT);
        ioport_set_pin_dir( IOPORT_CREATE_PIN(PIOA, 16),
            IOPORT_DIR_OUTPUT );
        ioport_set_pin_dir( IOPORT_CREATE_PIN(PIOC, 22),
            IOPORT_DIR_OUTPUT );
}

void hal_led_write( tLedNum  lednum, tLedState state ){
        uint32_t IOLine = 0;

        switch(lednum){
                case Led1: IOLine = IOPORT_CREATE_PIN(PIOC, 20); break;
                case Led2: IOLine = IOPORT_CREATE_PIN(PIOA, 16); break;
                case Led3: IOLine = IOPORT_CREATE_PIN(PIOC, 22); break;
                default: /* Error */ ;
        }

        //write pin
        ioport_set_pin_level( IOLine, !state );
}
```

Listing 8.4: Common HAL header file

```
/**
 * @file        hal.h
 * @author
 * @version
 *
 * @brief Header file for the HAL (IO, CPU and MEM)
 */

#ifndef HAL_H_
#define HAL_H_

#include <stdint-gcc.h> //defs for size-specific primitive data types
#include <stdbool.h>    //defs for true and false

typedef uint32_t tLedNum;
typedef bool tLedState;

enum tLedState  { LedOff = false, LedOn };
enum tLedNum    { Led1 = 1, Led2, Led3 };

//Init
void hal_io_init( void );

//LED
void hal_led_start( void );
void hal_led_write( tLedNum, tLedState );
```

```
#endif /* HAL_H_ */
```

Listing 8.5: System's module C file

```
/**
 * @file          system.c
 * @author
 * @version
 *
 * @brief System module
 *
 * System-related definitions, and definitions that are common
 * for different layers of the system.
 *
 */
#include ‘‘system.h’

void system_init(void){
        //Initializes the HAL and all other modules
        hal_io_init();

        //Starts IO Devices
        hal_led_start();

        //Sets IO initial state
        hal_led_write(Led1, LedOff);
        hal_led_write(Led2, LedOff);
        hal_led_write(Led3, LedOff);
}
```

Listing 8.6: System's module header file

```
/**
 * @file          system.h
 * @author
 * @version
 *
 * @brief System Module header file
 */

#ifndef SYSTEM_H_
#define SYSTEM_H_

//   ---------   SYSTEM OPTIONS    ---------
// (Configure system by editing this values)
#define SYS_VERSION     3


void system_init(void);

#endif /* SYSTEM_H_ */
```

Listing 8.7: Sample application

```
#include ’MiniOS/system.h’
```

```
#include 'MiniOS/HAL/hal.h'

int main(void){
        system_init();
        hal_led_write( Led1, LedOn );
}
```

Now with an idea of how the code will be structured we continue with another IO device.

8.4 Real-Time Clock (RTC) Example

In this case the functionality we want to expose to upper layers is very similar to the one offered in Basic IO, except we arbitrarily integrate time and date as a single concept: time. Also, we do not include the week parameter as it is not of any use for our purposes. Finally, since now time is composed of too many elements (seconds, minutes, hour, day, month, year) we integrate everything into a *struct*. Needless to say we do not want any setup to be part of the interface either. The resulting code is in Listings 8.8 and 8.11 (*system.h* is not shown as it remains unchanged). For a demo see HAL – RTC Debugging Demo.

Listing 8.8: HAL IO module C file

```
...

static void rtc_setup(void); //Q: Why static?!

...

void hal_clock_start( void ){
        rtc_setup();
}

void hal_clock_write( tTime* t ){
        rtc_set_date( RTC, t->year, t->month, t->day, 0 );
        rtc_set_time( RTC, t->hours, t->minutes, t->seconds );
}

void hal_clock_read( tTime* t ){
        uint32_t dummy_week = 0;

        rtc_get_time(RTC, &(t->hours), &(t->minutes), &(t->seconds));
        rtc_get_date(RTC, &(t->year), &(t->month), &(t->day), &dummy_week
            );
}

static void rtc_setup(){
        pmc_switch_sclk_to_32kxtal(PMC_OSC_XTAL);
        while (!pmc_osc_is_ready_32kxtal());
        rtc_set_hour_mode(RTC, 0); //24-hrs mode by default
}
```

Listing 8.9: Common HAL header file

```
 ...

/**
* A structure to represent time
*/
typedef struct{
        uint32_t seconds;
        uint32_t minutes;
        uint32_t hours;
        uint32_t day;
        uint32_t month;
        uint32_t year;
}tTime;

//Clock
void hal_clock_start( void );
void hal_clock_write( tTime* );
void hal_clock_read( tTime* );

...
```

Listing 8.10: System's module C file

```
void system_init(void){
        ...
        //Starts IO Devices
        ...
        hal_clock_start();

        //Sets IO initial state
        ...

        //Beginning of time was at 00:13:00 July 14, 1991
        tTime time = {
                0, 13, 0, 14, 7, 1991 //
        };
        hal_clock_write(&time);
}
```

Listing 8.11: Sample Application

```

#include 'MiniOS/system.h'
#include 'MiniOS/hal/hal.h'

tTime time;

```

```
int main(void)
{
        system_init();

        time.seconds = 0;
        time.minutes = 30;
        time.hours = 20;
        time.day = 27;
        time.month = 11;
        time.year = 2017;

        hal_clock_write(&time);

        while(1){
                hal_clock_read(&time);

                if( time.seconds % 2 == 0 )
                        hal_led_write( Led2, LedOn );
                else
                        hal_led_write( Led2, LedOff );
        }
}
```

At this point it should be clear that IO devices functionality for a given *device* is exposed via *hal_device_start*, *hal_device_write*, and as we shall see later *hal_device_read*. It should also be clear that all these functions are defined in *hal_io.c*, the IO part of the HAL.

8.5 Final remarks—on low-level programming

Programming low-level code is considered difficult and is often quite frustrating. There are a not small amount of technical details that must be considered when being so close to hardware; one must have a working understanding of the processor, memory, and peripherals in order to use them; bugs manifest in very very very oddly manners; high-level software/programming facilities to help in the development process are not available; C pointers are often difficult to use correctly; among a few other reasons.

So when the reader is getting frustrated after debugging seamlessly correct code for hours—because it will happen—he should consider that unfortunately this is how low-level programming is. **Computers are complex machines, and their direct manipulation is plagued of minutia. Yet it helps to be methodic, thorough, and careful**. For instance, write code by building on code that has been tested and is known to work. Build a small piece, test, build another small piece, test, and so forth. Suspect every line of code! Also, use the debugger to your advantage. Are there things that are supposed to be happening? Use the debugger and verify them!

8.5.1 Debugger problems

The debugger is not perfect. Sometimes the debugger likes to behave in *strange manners*. When that happens, the reader should remember she is visually stepping through C code, which is in fact machine code in the MCU's program memory. So there may be a mismatch between the two. Moreover, the CPU is paused and letting us inspect and modify memory and registers after every step—that is outstanding!

How is that possible? CPUs are built to be debugged. See: ARM Cortex M series Debug and Trace (the first minute).

Resources on the web

- (Video) Abstraction from physics to applications
- (Video) HAL and Driver Compatibilty
- C2 Wiki – Modular Programming

Exercises

This is the code base for this chapter. It has the structure for what was discussed in this chapter, and partial solutions to the exercises. Please note the programming pattern whereby callbacks are used to pass data coming from a lower layer (Basic IO) to an upper layer (demo apps).

1. Add the following functionality to the HAL: millisecond timer, sensor, display, led, clock, button, system timer, and non-volatile mem-

ory. A correct solution must make these three examples work: sample_app1.c, sample_app2.c, and sample_app3.c[5] .

For reference, these are the solution demos: demo app 1, demo app 2, and demo app 3.

2. Add any other two functionalities of your choice to the HAL, and make a demo for it.

[5] Important! Please consider that at the moment of writing this chapter, the latest version of the ASF has a bug that prevents the Micro SD Card on the IO1 Board to be read properly, unless there is an empty SD Card adapter inserted in the slot on the back of the SAM4S board. See here

Chapter 9

Executing applications

It's 5.50 a.m.... Do you know where your stack pointer is ?

Anonymous.

Up to this point, writing applications that run on MiniOS requires that applications and the system itself[1] are compiled together; and subsequently deployed as a single binary, or as a single piece of firmware[2]. Although this is the norm when developing embedded systems, we do not want that. Instead we want a system, compiled separately, with the capabilities of executing arbitrary applications, in the same way general-purpose computer do. Before getting into the details of how to achieve this, we need to put this idea in context.

9.1 Firmware in MCUs

In Sections 1.1 to Sections 1.3 we looked at the differences between MCUs and MPU-based computers from a high-level perspective. We discussed how MCUs are computer in a die, and how they are typically used in

[1] At the moment the system is just the HAL

[2] Recall we call the software running in MCUs *firmware*

embedded systems, such as printers and sensors, where they fulfil a single task throughout their lifetime. Then in Sections 3.1 we discussed how MCUs typically implement a modern Hardvard architecture with physically separated memories. While one memory is for holding data, the second one is for holding instructions i.e., code. This memory dedicated for instructions is typically implemented as non-volatile Flash memory and receives the name of **program memory**. In the ATSAM4SD32C, it is the memory starting at address 0x00400000. So, every time we *flash* (or program) the device with a new *binary*, that binary is written to program memory in Flash, thereby replacing the old firmware. Lastly, at runtime, on power up, a *loader* (Sections 3.6) loads the Data, BSS, and other segments into main memory; and once finished, it transfers execution to *main* (again, *main* is somewhere near 0x00400000). This, in a nutshell, is the development-deployment process that takes place in the ATSAM4SD32C and in any other MCU in general.

From the above explanation there is one important thing to realize: except for updates, **firmware is not meant to be replaced**. Why would anyone want to run a coffee maker application in a microwave? In any case, loading of applications at runtime for execution is not something for which MCUs were designed.

Before continuing the reader must note that in the context of firmware, the term application is used to specify the "control logic" part of the firmware. Meaning, all the code that are not libraries or frameworks such as the ASF or even the HAL. So the demos in the previous chapter are considered *apps*.

Now we briefly discuss how memory (RAM) may be used to hold apps.

9.2 OS as firmware, apps as software

It is possible to modify the system as to enable apps to be loaded from permanent storage and executed. That is, like they do in general-purpose computers. In fact, we could do the same with the system itself, but there is no point in leaving program memory empty and waste it. So, we will let the system continue to execute from Flash, and simply enhance it with app-loading capabilities. In other words, we want: **MiniOS as firmware**

and applications as software.

The exact mechanisms are detailed next.

9.3 Running apps on MiniOS

Running apps is a straightforward task, but it requires some precise understanding of how the machine executes instructions and how the compiler generates code. As usual, let us demonstrate with an example.

Say, we already have MiniOS (up to the HAL) living in program memory. Now what? How do we execute an application? Well, first we need an application. Moreover, it must be a stand-alone application that is not compiled with the HAL. Listing 9.1 shows a simple stand-alone application that uses the drivers in pio.s to turn LED0 on. *pio.s* was introduced in the IO chapter (see Section 4.3).

Listing 9.1: Sample App

```
#define OUTPUT_DIR  0
#define LEVEL_LOW   0

void main(void){
        pioc_init();
        pioc_dir_set( OUTPUT_DIR, 23 );
        pioc_level_set( LEVEL_LOW, 23 );
}
```

Because it is a stand-alone application, creating an Atmel Studio project to compile it is different from what the reader has done before. The exact steps are demonstrated in Executing Applications—How to create a stand-alone C project.

Now we have an application. What is next? We want this application to run from RAM. *How* do we tell that an application must run RAM? More importantly, *who* do we tell that to? The *who* is the linker. The linker is the one responsible for address resolution (Section 3.4). Therefore, the *how* is the linker script (Section 3.5). The linker script for our sample stand-alone C project is split in two files, *sam4s_flash.ld*, and *sam4sd32c_flash.ld*. (For some reason Atmel decided to split it, it makes no difference.) Part of the linker script structure is shown in Listing 9.2.

```
MEMORY{
  rom (rx)   : ORIGIN = 0x00400000, LENGTH = 0x00200000
  ram (rwx)  : ORIGIN = 0x20000000, LENGTH = 0x00028000
}
SECTIONS{
    .text : {
        ...
    } > rom
    ...

    .ARM.exidx : {
        ...
    } > rom
    ...
}
```

Listing 9.2: Sample app's linker script

From inspection, it is clear that two segments are specified to reside in Flash. So this is the first thing we change. Telling the linker to run a segment from RAM is very simple (See Listing 9.3.).

```
MEMORY{
  rom (rx)   : ORIGIN = 0x00400000, LENGTH = 0x00200000
  ram (rwx)  : ORIGIN = 0x20000000, LENGTH = 0x00028000
}
SECTIONS{
    .text : {
        ...
    } > ram
    ...

    .ARM.exidx : {
        ...
    } > ram
    ...
}
```

Listing 9.3: Fixed sample app's linker script

Okay, that should fix it right? The linker will generate code that uses only addresses within main memory. Variables, code, constants, and everything will have a runtime address in RAM. Given a binary image with this information, a loader should have no problem loading it into main memory. For verification, we inspect the dissambly (*.lss*) file for the sample application[3]. Part of the dissasembly is shown in Listing 9.4.

[3] In the folder *Output Files* accessible from Atmel Studio's solution explorer

```
Disassembly of section .text:

20000000 <__do_global_dtors_aux>:
20000000:    b510        push  {r4, lr}
20000002:    4c05        ldr   r4, [pc, #20]   ; (20000018
    <__do_global_dtors_aux+0x18>)
...
```

Listing 9.4: Dissasembly for sample app

Effectively, all addresses are in the RAM area, but there is a problem. *main* is not to be found anywhere! Where is *main*? Well, it turns out the entry point for any bare-metal application is the vector table, which is specified as *.vectors* in our application's linker script (Listing 9.5).

```
 .text : {
   . = ALIGN(4);
   _sfixed = .;
   KEEP(*(.vectors .vectors.*))
   *(.text .text.* .gnu.linkonce.t.*)
    ...
```

Listing 9.5: Sample app's entry point

The linker is "smart" and does not include, in the executable, code or data that is not reachable from the "execution tree" (think of function calls as edges). The keyword *KEEP* is used to force the linker to "keep" code or data even if they are not reachable. One of the places where KEEP has to be used, is to mark the executable's entry point (the root of the tree). In bare-metal executables, the root of the tree is the reset vector (in the vector table), and execution continues to branch thereon. Nonetheless, our application is not bare-metal and it has no knowledge of reset vectors. There are no vector tables in stand-alone applications! Instead the root of the tree should be *main*, because *main* is where execution begins. **Who transfers execution to main? The OS**. With this knowledge, there is one simple fix that enables the linker to continue doing its optimizations and keeps "extra code" to a minimum. Namely, create a custom entry point.

Creating a custom entry point is as simple as creating a new input section, say *.entry_point*, and telling the compiler to place *main* in it. Listing

9.6 and 9.7 show the corresponding code.

```
.text : {
  . = ALIGN(4);
  _sfixed = .;
  KEEP(*(.entry_point))
  *(.text .text.* .gnu.linkonce.t.*)
   ...
```

Listing 9.6: Custom entry point (linker script side)

```
#define MINIOSAPP __attribute__ ((section('.entry_point'))) void

MINIOSAPP main(void){

}
```

Listing 9.7: Custom entry point (application side)

__attribute__ ((section(".entry_point"))) is simply the way we tell the C compiler that *main* is part of the *.entry_point* input section (its assembly equivalent is *.section .entry_point*). On inspection of the dissasembly file in Listing 9.8 we can see *main* is back.

```
Disassembly of section .text:

20000000 <main>:
20000000:    b508         push  {r3, lr}
20000002:    4b05         ldr   r3, [pc, #20]   ; (20000018 <main+0x18>)
...
```

Listing 9.8: Dissasembly for sample app

By now you may have already guessed that this is not over—there is likely something else that we have not considered. That is correct. Applications cannot be starting at address 0x20000000. Why? Because the system is also using that same address space. Loading an application would thus corrupt OS data. Fortunately, the solution is also quite simple. **It suffices to split RAM into two parts, one for application code and data, and one for OS data (code is in Flash)**. Physically, there are roughly 160 KB of SRAM present. The linker script specifies that RAM starts at 0x20000000 and has a length of 0x00028000 (See Listing 9.3.). Based on this, we modify the linker script as shown in Listings 9.9 and

9.10.

```
MEMORY
{
  rom (rx)  : ORIGIN = 0x00400000, LENGTH = 0x00200000
  ram (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00014000
}
```

Listing 9.9: Fixed OS' linker script

```
MEMORY
{
  rom (rx)  : ORIGIN = 0x00400000, LENGTH = 0x00200000
  ram (rwx) : ORIGIN = 0x20014000, LENGTH = 0x00014000
}
```

Listing 9.10: Fixed app's linker script

Once again we inspect the sample app's dissassembly, and we find that *main* now lives in its allocated part of memory. See Listing 9.11. (The system disassembly should not change.)

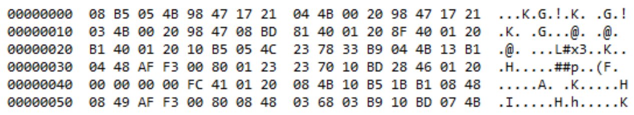
```
Disassembly of section .text:

20014000 <main>:
20014000:    b508        push  {r3, lr}
20014002:    4b05        ldr   r3, [pc, #20]   ; (20014018 <main+0x18>)
...
```

Listing 9.11: Dissasembly for sample app

Note that the entire memory is still accessible at runtime. By changing the length of the specified available memory, we simply made sure that no memory past 0x20013FFF is allocated at compile time for the system. As far as the linker knows our target device has 80 KB of RAM available.

Okay. Is that everything?! At least from the application side that seems to be everything. All is left is to locate the final binary file, which is in *./Debug/MyApp.bin*[4] and has a size of 2 KB. A small part of its contents are shown in Figure 9.1. (Any *Hex editor* can be used to inspect binary files.)

[4] Make sure to use the right binary! Depending on which configuration you have active the new binary will be placed in the *Debug* folder, or in *Release*. See here and here.



```
00000000  08 B5 05 4B 98 47 17 21  04 4B 00 20 98 47 17 21  ...K.G.!.K. .G.!
00000010  03 4B 00 20 98 47 08 BD  81 40 01 20 8F 40 01 20  .K. .G...@. .@.
00000020  B1 40 01 20 10 B5 05 4C  23 78 33 B9 04 4B 13 B1  .@. ...L#x3..K..
00000030  04 48 AF F3 00 80 01 23  23 70 10 BD 28 46 01 20  .H.....##p..(F.
00000040  00 00 00 00 FC 41 01 20  08 4B 10 B5 1B B1 08 48  .....A. .K.....H
00000050  08 49 AF F3 00 80 08 48  03 68 03 B9 10 BD 07 4B  .I.....H.h.....K
```

Figure 9.1: Sample app's binary image

If the reader noticed, the content seems to start at address 0x0000000. That is because **binaries contain "raw *binary*" with no extra information**. Other executable formats like ELF or HEX files may contain, besides the content per se, addresses, memory segments, and even checksums for integrity verification. That is why it is possible to inspect an ELF and see its contents laid out in segments. In any case, **binaries simplify loaders and are thus used as executables in MiniOS**. Provided the binary image is already in a buffer in memory, Listing 9.12 shows code for loading an app—ta-da!

```
void load_app( uint8_t* dest_mem, uint8_t* buffer, uint32_t size ){

   for(uint32_t i = 0; i<size; i++)
      dest_mem[i] = buffer[i];

}
```

Listing 9.12: OS loader

As for the rest of the code, we prefer a more *proper* implementation that follows the system's architecture. In particular, the loader must sit on the HAL, and be as hardware-agnostic as possible. To this end, two more abstractions were added to the HAL: *memory regions* and *non-volatile memory*. The final results are in hal_memreg.c and hal_nvmem.c. The final loader is in main.c.

For a demo see: Executing Applications—App.bin Demo.

Now apps are supported, but how good is a system that can only run a pre-defined app? An user interface is missing! Next the idea of a console is discussed.

9.4 Console

A console is a text-based user interface for an operating system. The author is not old enough to really understand the distinction between command-line, shell, terminal, and console. Yet, it seems clear that a console is a more general term. Apparently, terminals and command-lines are consoles. Shells are peculiar because their purpose is to offer kernel services, and they are not part of the kernel itself. If all that is true, then **a text-based interface that prompts the user for input within (internally from) MiniOS can safely be called a console**. Something much simpler, but analogous to: The Linux Console, which is also implemented as part of the kernel.

In either case, with the new application-executing capabilities, writing a console is straightforward. Because its implementation is left as exercise, the exact details are not discussed. Yet here is some non-implementation related information that may help the reader in conceiving the actual code: **the console never returns**[5]. The console runs an application, and on termination, the console continues to execute by prompting the user for input. This is/was the behaviour of MS-DOS and any GNU/Linux distro that does not starts the GUI from the beginning. Where exactly does the code for the console goes? Recall the console is internal to the system, it is not an application. Another hint: after what would we expect to see the console show up? Lastly, Figure 9.2 shows how the console fits in the current architecture.

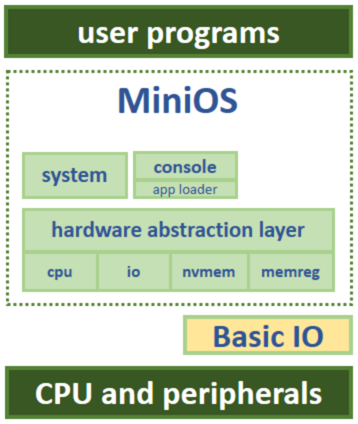


Figure 9.2: System's architecture for this chapter

[5] Has the reader ever written a text-based program that shows a menu and never exits unless 'q' is selected? Something like that

9.5 Final Considerations

Support for application loading and execution in MCUs is not common. As far as the author knows, there are only two other OSes that support this: StratifyOS and uCLinux (the Linux version for MCUs). The approach from this chapter is different, though. The exact details for StratifyOS are discussed here, while the ones for uCLinux are here.

One drawback from our approach is that constants live in RAM, software could modify a constant by mistake (i.e. corrupt it) and it would be possible because constants are in RAM (and not in Flash where they belong). A counter argument is that even if that constant was not there, other data would be there and get corrupted. So, one way, or another, that application was doom to crash.

Because we like putting things in context, here are some other similar OSes: FreeRTOS, ChibiOS, RiotOS, Contiki, TinyOS.

Last, but not least, **the debugging capabilities have been lost for applications**. The system can still be debugged, but not apps. For good or bad, with no debugging, and no ASF, there is no benefit in using Atmel Studio for writing apps. Any other lighter IDE that supports the ARM GNU C compiler can serve well. (The author likes emIDE).

Resources on the web

- Birth of the Microsoft DOS
- My First Line of Code: Linus Torvalds
- How Linux is Built
- Linux History
- AT&T Archives: The UNIX Operating System
- The UNIX Time-Sharing System

Exercises

The base code for this chapter is ExecutingAppsBase.zip and MyApp.zip. Feel free to use PuTTY or any other serial console as console.

1. In the base code, the actual code for executing the app is missing. Write it. (Hint: Section 5.2.2).

2. Write a console that supports executing applications by name, in addition to the commands *ls* and *cat*. For a demo of the solution see Executing Apps—Solution Demo.

Provided exercises 1 and 2 have been solved:

3. Add to the console any two commands of your choice.

4. Enable the console to pass command-line arguments to applications. Hint: use registers to pass arguments (See Section 5.2.1).

5. Enable the console to execute a group of commands, one after another stored in a .bat file.

6. Generating pseudo-random numbers is difficult on computers. Use the randomness from on-board sensors to generate a random number. Create a command to access it.

Chapter 10

System Calls

All problems in computer science can be solved by another level of indirection

David J. Wheeler

So far we've seen how to create a process. There is one problem we have overlooked: **processes have full access to OS code and data**. This means apps are capable of altering the OS itself, either intentionally or by mistake. When modification is intentional the problem falls under security, and the idea is to prevent malicious code from messing with the system, e.g., crashing it or leaking sensible information[1]. On the contrary, when it is unintentional, it is a matter of fault tolerance, and the idea is to prevent an application bug to crash the entire system and cause unwanted side effects[2].

In either case, granting applications arbitrary access to the kernel is a bad idea. To show this more concretely, let us look at an example.

Consider the code in Listing 10.1. This piece of code executes OS code and modifies OS data, as shown in Demo—messing with the system.

[1] See, for instance, Android Bug Exploit

[2] See, for example, Toyota's unintended acceleration case

```
MINIOSAPP main(void){
   //Accessing OS code (calling hal_led_write in memory)
   for( uint32_t i=0; i<5; i++ )
      ((void(*)(uint32_t, uint32_t))(0x004008a4 + 1))(i, true);

   //Accessing OS data (changing the prompt message)
   strcpy( (uint8_t*)0x20008FE0, 'mariana(at)minios:~\$' );
}
```

Listing 10.1: Malicious App

To prevent this type of situations CPUs typically have a **Memory Protection Unit** (MPU) and support for various **CPU modes**, each mode with different levels of privileges. Together they enable the system to enforce access prohibitions on specific memory regions. In a typical OS environment, the kernel's address space is protected by the MPU, and attempting to access it from application code results in a *segmentation fault*.

Now, the MPU is complex enough to be a chapter for its own, so MPU protection is left for the Memory Protection Chapter. Meanwhile, this chapter is concerned with adding support for privileged and unprivileged execution via CPU modes.

10.1 CPU Modes in the Cortex-M4

The Cortex-M4 supports two CPU modes, kernel mode and user mode. **Kernel mode**[3] is by default privileged, and it is thus the mode intended for the system. **User mode**[4], on the contrary, can be configured to be either privileged or unprivileged, and it is thus the mode intended for applications.

Once privilege separation is in place, stacks must too be separated, or else it becomes difficult to determine what stack frame belongs to whom and how access must be restricted. The Cortex-M4 has support for two stacks: the **main stack** and the **process stack**. By default kernel mode can only use the main stack, whereas user mode can be configured to use either the main or process stack. **So, at any moment, the CPU is in**

[3] *handler mode* in Cortex-M jargon

[4] *thread mode* in Cortex-M jargon

one specific mode with one specific active stack. In particular, we want the system to run in privileged kernel (handler) mode using the main stack, and applications to run in unprivileged user (thread) mode using the process stack.

Next we look at how to configure the machine so that each mode is set up as specified above.

10.2 Configuring user mode

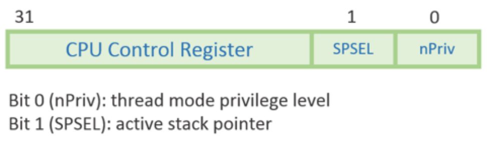

31
1
0
CPU Control Register
SPSEL
nPriv
Bit 0 (nPriv): thread mode privilege level
Bit 1 (SPSEL): active stack pointer

Figure 10.1: Control register

The level of privilege and the active stack for user mode is controlled by the **Control Register** (Figure 10.1). The *nPriv* bit specifies privilege level (1 is unprivileged, and 0 is privileged). The active SP is controlled by the *SPSEL* bit. 0 indicates *main stack pointer* (MSP), and 1 indicates *process stack pointer* (PSP).

To write/read to/from registers we normally use the *MOV* instruction. The Control Register, however, is a **special register**, and cannot be written by regular *MOV* instructions. Instead it requires the use of special instructions: **MSR** and **MRS**. *MRS* moves the content of a special register to a general purpose register, while *MSR* does the opposite. Since these two instructions are used to access special registers, they can only be executed by privileged code.

(Figure 10.2 shows all Cortex-M4 registers, both general and special, and the level of privilege required to access them.)

Based on this we write the assembly code shown in Listing 10.2 for setting user mode as

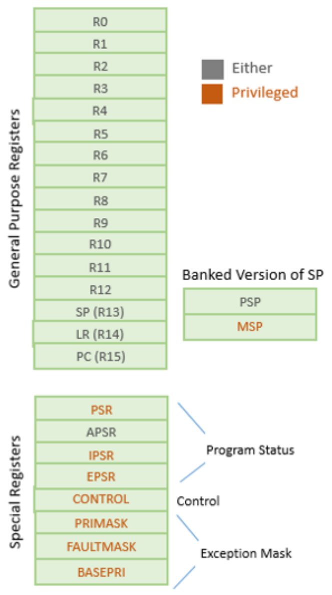


Figure 10.2: Cortex-M4 Registers

unprivileged. Notice the presence of an *isb* (instruction synchronization barrier) instruction. This is to flush the CPU's pipeline so that any subsequent instruction is fetched from memory or cache (documentation states *isb* should be used after updating the control register to ensure the new configuration is used by following instructions).

```
.thumb_func
.global hal_cpu_set_unprivileged
hal_cpu_set_unprivileged:
   mrs r3, control
   orr r3, r3, #1
   msr control, r3 /* control |= 1 */
   isb

   bx lr
```

Listing 10.2: Routine for setting user mode as unprivileged

To comply with the system's architecture, we make this new CPU functionality part of the HAL (in *hal_asm.s*). Also, because we want applications to run without privileges, we have it called before executing an application, as shown in Listing 10.3.

```
uint32_t loader_exec_app( uint8_t* app_name, uint8_t* param, uint32_t
    num_params, uint32_t* ret_code  ){

    ...
    //Remove privileges
    hal_cpu_set_unprivileged();

    //run!
    *ret_code = ((uint32_t(*)(uint32_t*, uint32_t))app_memreg.base +
        1)(param, num_params);

    return LOADER_EXEC_SUCCESS;
}
```

Listing 10.3: Executing unprivileged apps

For a demo see System Calls—Testing unprivileged [Check this demo, it feels backwards]. User mode is now configured to run as unprivileged[5], as we wanted it. Next we consider setting a stack for each mode.

[5] Be careful not to confuse this with switching to user mode, we simply told the CPU that user mode runs unprivileged—no less, no more.

10.3 The two stacks

We said above that we wanted kernel mode to use the *main stack*, and user mode to use the *process stack*. As usual, let us begin with a basic question—where are these stacks? The *process stack* is in the application's address space, and the *main stack* is in the kernel's address space. What about "the stack", the one we have been using so far, are there three stacks in total? No, there are only two, **we have been using the *main stack* unknowingly, as it is the default stack on power up**. Okay good. Now, how does code know what stack to use? Stacks as such do not really exist, we say code uses one stack or another depending on what value the *SP* holds when *push* and *pop* are executed. Therefore to support two stacks, the machine implements two stack pointers the *process stack pointer* (*PSP*) and the *main stack pointer* (*MSP*). Then, on CPU mode switching, SP "gets linked" to either *PSP* or *MSP*, depending on the *Control Register* and from what mode to what mode is the switch to. The important part here is that this is handled automatically. **The only thing we need to do ourselves is to give *MSP* and *PSP* their initial values**. That is, make *MSP* = end of the *main stack*[6] and *PSP* = end of the *process stack*. Because the machine starts with the *MSP* as active, giving *MSP* its initial value was completed when initializing *SP* (See here and here).

Now the real problem—what is the initial value for *PSP*? Applications are not compiled together with the system. All MiniOS gets from an application is its binary, nothing more. **So MiniOS has no idea where the end of the stack is for any application**. What if we say that all applications must place their stack in the same place; could we agree on some address, say, 0x20014C00? We could but that is rather ugly, and hard-coded values tend to complicate things in the long run. What do other OSes do? Other OSes add additional information in executables to communicate the OS loader initial values of specific registers such as the SP. ELF executables have support for that, for instance (see Startup state of a Linux/i386 ELF binary). Could we do something similar without having to give up raw binaries[7]? Yes we can. **Specifically, we dedicate**

[6] Recall the stack grows downwards, see Chapter 6—Stack

[7] Writing an ELF parser is not a trivial task

the first 4 bytes in the binary to hold the initial *SP*[8], and the remaining stays the same. The exact code is shown in Listing 10.4 and 10.5.

```
...
#define MINIOSAPP __attribute__ ((section('.entry_point'))) uint32_t
#define STACKINFO __attribute__ ((section('.stack_info'))) const uint32_t

extern uint32_t _estack;
STACKINFO sp  = &_estack;

MINIOSAPP main(void){

   return 0;
}
```

Listing 10.4: New Application template for MiniOS

```
SECTIONS {
     .text : {
          . = ALIGN(4);
          KEEP(*(.stack_info))
          KEEP(*(.entry_point))
          ...
```

Listing 10.5: New linker script for MiniOS

The App memory region defined in the HAL must too be updated to match the new binary layout (Listing 10.6).

```
#define MEM_REGION_APP_BASEPTR      (uint8_t*)0x20014004    //App mem
    region, ptr to start address
```

Listing 10.6: hal_memreg.c

The loader is also modified accordingly (Listing 10.7). Notice *hal_cpu_set_psp* and *hal_cpu_set_psp_active* must be called before *hal_cpu_set_unprivileged*. Can the reader guess why?

```
    ...
    //Get App region details
    hal_memreg_read( MemRegApp, &app_memreg );

    //reads app
   uint32_t bytes_read = hal_nvmem_fat_file_read( app_name, binary,
      SYS_APP_MAX_SIZE );
```

[8] Apps have no concept of *PSP*, they only knows about *SP*

```

    //Extract app's initial SP and application from binary
    uint32_t app_sp = *((uint32_t*)binary);
    uint32_t* app_buffer = (uint32_t*)binary + 1;

    //load app in App memory region
    load_app( app_memreg.base, app_buffer, bytes_read );

    // Set up CPU for user mode execution
    hal_cpu_set_psp( app_sp );
    hal_cpu_set_psp_active();
    hal_cpu_set_unprivileged();

    //run!
    ...
}
```

Listing 10.7: Setting CPU for user mode execution

Listing 10.8 shows the definitions for *hal_cpu_set_psp* and *hal_cpu_set_psp_active*.

```
.thumb_func
.global hal_cpu_set_psp
hal_cpu_set_psp:
     msr psp, r0 /* return PSP */

     bx lr

.thumb_func
.global hal_cpu_set_psp_active
hal_cpu_set_psp_active:
    mrs r3, control
    orr   r3, r3, #2
    msr control, r3 /* control |= 2 */
    isb

    bx lr
```

Listing 10.8: Routines for setting PSP

For a test of everything see System Calls—Setting PSP. Finally, it is time to look at how the actual switching takes place,

10.3.1 Mode switching

It turns out that we cannot manually change from user mode to kernel mode. We can only specify the privilege levels and what stack is used for each mode—the switching occurs automatically on exception handler entry and exit. **On exception entry the CPU goes into kernel mode, and on exception return it goes back into user mode**. So, kernel (handler) mode only takes place from within exception handlers, hence the name *handler mode*. Exception handlers like *Systick_Handler*, *SVC_Handler*, *UART1_Handler*, and any other handler. **So, after initialization, once control of the machine is transferred to the application, the kernel only gets it back on an exception.** Then how does the kernel ever gets to run? If the app attempts to access an area protected by the MPU, a *MemFault* exception occurs and the kernel regains control of the machine, within the *MemFault* handler. If the app is executing and the scheduler timer ticks (to signal that a different thread should run), execution is transferred to the *SysTick* handler; because the *SysTick* handler is kernel code, we say the kernel now has control of the machine. If an application is running and data arrives from the *UART1*, the *UART1* handler executes i.e. control is transferred from the application to the system, and the system can store the received data and return to let the application continue. Well, the reader gets the idea. Something else that happens on exception entry and exit are the switching of stacks, as per the Control Register.

Lastly, how is the machine initially placed in user mode? **The machine comes out of reset in user mode**. When execution begins in the reset handler (see Listing 10.9), the machine is in user mode. It stays the same when *main* is executed, and later when an application is executed.

```c
void Reset_Handler(void)
{
    /* Initialize the relocate segment */
    ...
    /* Clear the zero segment */
    ...
    /* Set the vector table base address */
    ...
    /* Branch to main */
    main();
    /* Infinite loop */
    while (1);
}
```

Listing 10.9: Reset Handler at *startup_sam4s.c*

There is still one question—if applications are not allowed to access OS code, OS data, memory-mapped IO, nor special registers, how can they do anything at all? Simple, they request it to the system. For humanity's sake, applications must be treated like toddlers who will kill themselves or steal all our money if we let them. The exact mechanism is discussed next.

10.4 System calls

System calls are the interface between applications and the operating system. They are the interface whereby applications request OS services and access to hardware. However, from a system calls perspective, privileged execution presents a challenge, for how can applications possibly call OS services without access to OS code i.e., without function/subroutine calls? Like many other problems in computer science, the solution is—**indirection**. Meaning, *place something* in between apps and system so that they do not have to be in direct contact with each other (or at least apps do not have to touch the kernel). Such indirection has historically been in the form of **software interrupts**.

Although alternatives exist, software interrupt implementations are typically supported by hardware and are the de-facto standard. Architectures typically have an instruction that will trigger an interrupt, hence the name *software* interrupt. Different architectures give different names to it, e.g., *int* in x86 and *syscall* in MIPS. In Cortex-M4 jargon they are called **supervisor calls** and the instruction that triggers the interrupts is ***svc***. (More details in Section 6—Interrupts)

In addition, typically there is a library/API sitting between user programs and the actual software interrupts (e.g. the GNU C Library and the Windows API). User programs can then call the library/API (in user mode), which in turn makes the corresponding call to the system. Not all subroutine calls to the library/API result in a call to the system, though. While some functions may be able to handle everything from ***userland***, some others will translate to a kernel request.

As for MiniOS, we would like to comply with the standard[9]. So we implement a system call interface and a small library *minilib*, as shown in Figure 10.3.

[9] How else could it be done anyway?

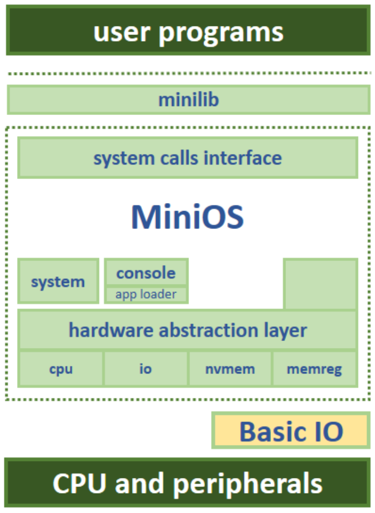


Figure 10.3: Architecture goal for this chapter

Note: Because there is no file system or scheduler yet, for the most part, the system call interface will be a wrapper for the HAL. Still, not every functionality of the HAL needs to be offered by *minilib*; and not every functionality offered by *minilib* must end up as a call to the system.

In the following section we consider a concrete example.

10.5 Example system call

As described earlier, the machine is set up in user mode before app execution. Once in *userland* applications can make subroutine calls to *minilib*, as shown in Listing 10.10.

```
#include 'minilib/led.h'
...
MINIOSAPP main(void){
    led_write( Led1, LedOn );
    return 0;
}
```

Listing 10.10: App calling minilib

For simplicity, *minilib* is as an assembly file (*minilib.s*), not an actual pre-compiled library, at least for the time being. See Listing 10.11 and 10.12. (Notice that, although *minilib* is one single library, we make header files for different *groups* of functionality, such as LEDs, display, buttons, and so forth).

```
.equ   SVCLedSet, 0
...
.thumb_func
.global  led_write
led_write:
    svc  SVCLedSet
    bx  lr
```

Listing 10.11: minilib.s

```
...
enum tLedState{ LedOff=false, LedOn };
enum tLedNum{ Led0 = 0, Led1, Led2, Led3 };

void led_write( tLedNum , tLedState );
```

Listing 10.12: led.h

Wait! That code is exactly like the one on the system side. So is *minilib* just a wrapper? For some library calls, yes it is a wrapper. For some others it is not. The important part here is the use of software interrupts.

When calling *led_set*, minilib translates it into a system call. Upon execution of the corresponding SVC instruction, control flow is transferred to the *SVC Handler*, where the system call is *attended*. To comply with the system's architecture (where interaction with the machine is via the HAL), we add SVC functionality to the HAL (see Listing 10.13 and 10.14).

```
void (*svc_callback)(void);
...
/**
*  CPU SVC Start
*
*  Starts SVC calls and registers a callback function.
*
*  (at)param callback the function that gets called on supervisor calls
*/
void hal_cpu_svc_start( void(*callback)(void) ){
   svc_callback = callback;
}
```

Listing 10.13: New SVC functionality (in hal_cpu.c)

```
//SVC Handler
//(no prologue and no epilogue to avoid
// changing the offsets of the stacked frame on SVC interrupt)
.thumb_func
.global SVC_Handler
SVC_Handler:
   ldr   r3, =svc_callback
   ldr r3, [r3]
   bx r3
```

Listing 10.14: The actual SVC Handler (in hal_asm.s)

Then within the system calls interface we use the new SVC functionality to create a **system calls entry point**, as shown in Listing 10.15

```
...
void syscalls_init(void){
   hal_svc_start( syscalls_entry_point );
}

void syscalls_entry_point(void){

   //attend syscalls here
}
```

Listing 10.15: syscalls.c

By way of these changes, the callback registered in *hal_cpu_svc_start* becomes the entry point of the system. So on execution of an SVC instruction, control flow is transferred to *syscalls_entry_point* wherein we are officially in ***kernel land***. As this process can be cumbersome let us see a demo of the process so far: System Calls—Entering Kernel Mode.

10.6 Example system call continued (attending a system call)

Once in the kernel, the system must find (a) what request is being requested; and (b) the parameters of the request. In concrete terms, (a) the immediate value used in the SVC instruction, as different immediate values correspond to different system calls; and (b) any arguments necessary to attend the request (*led_num* and *state* in this particular case).

The problem then becomes—how to pass arguments to the kernel when making a supervisor call? The trick lies in knowing that on exception entry registers are pushed onto the stack in a process known as *stacking* (Chapter 6—Interrupts). Since they were pushed onto the stack (PSP) on execution of the SVC instruction, they can be read by the system[10]. Also, since we know R0-R3 are used to pass arguments in a function call (see Chapter 4—Stack), we expect to find *led_num* and *state* in the stacked *r0* and *r1*, respectively. Based on this we derive the code in Listing 10.16. (The exact details as to how the SVC immediate value is being read is left as exercise.)

[10] Although we want to prevent apps from accessing the system, the system is free to access app's address space as it wishes

```
void syscalls_entry_point(void){

   uint32_t sp = hal_cpu_get_psp();

   //Get syscall number from SVC instruction in program memory
   //-------------------------------------------------------
   //To find its address we look at the return address stacked on
   //exception entry (offset 6 from SP).
   //That stacked return address is the address of the instruction to
   //be executed on exception return (typically bx lr).
   //We know bx lr is a 16-bit instruction. So to get the SVC intruction
   //we just read 2 bytes before the specified address.
   uint32_t svc_number = ((uint16_t*) ((uint32_t*)sp)[6])[-1];

   //extract the number from the read instruction
   svc_number &= 0x00FF;

   //get arguments
   void* arg0 = ((uint32_t*)sp)[0];
   void* arg1 = ((uint32_t*)sp)[1];
   void* arg2 = ((uint32_t*)sp)[2];
   void* arg3 = ((uint32_t*)sp)[3];

   //attend syscall
   switch(svc_number){
      case SVCLedWrite:
         hal_led_write( (tLedNum)arg0, (tLedState)arg1 );
         break;

      default: /* Error */
   }
}
```

Listing 10.16: Attending a syscall

10.7 Running the console in kernel mode

Everything is set up now. Still, there is one detail we have purposefully overlooked: the console is currently executing in user mode! In fact, on application termination the console crashes because it's using the wrong stack and lacks privileges. However what we have seen so far is more than enough for one lab. The solution is discussed in the next labs. **Until then, the reader must remember that, after an app returns, the console will crash**.

10.8 Final considerations

Why did we bother in doing all this? To protect the system from evil and sloppy applications. So... is the system secure now? Not even a little. Kernels, and software in general, are plagued with bugs and implementations that create vulnerabilities. One just needs to take a quick look at the National Vulnerability Database to see the vast amount of vulnerabilities in today's software (this is the list for Linux). As it turns out, writing secure software is difficult.

That being said, now it is more difficult to mess with the kernel (or will be once memory protection is in place). One has to hand craft attacks for them to work. See, for instance, Buffer Overflow Exploit Demo (SAM4S Xplained Pro + MiniOS + Xbee)

Resources on the web

- System Calls make the world run.
- (Video) Linux Tutorial: How a Linux System Call Works
- (Video) Glibc Helps Hackers Pop Linux - Daily Security Byte EP. 217
- (Video) Collaboration Summit 2013 - GNU C Library
- (Video + Paper) FlexSC: Flexible System Call Scheduling with Exception-Less System Calls.

Exercises

This is MiniOS base code[11] for this chapter, and this is the App base code.

1. Implement any four system calls[12]. that do not return values. Write an app to demonstrate their functionality.

[11] We encourage the reader to use her own solutions instead.

[12] Example of apps using syscalls: LedBlink (demo here), ClockPrint (demo here), Buttons (demo here), Serial (demo here), and LettersDisplay (demo here).

2. Implement any four system calls that return values[13]. Write an app to demonstrate their functionality.

3. Write a system call for *display_printf* (analogous to *printf*) that works for any number of arguments[14].

4. Write an *strace* system call analogous to Linux strace. Demonstrate with an app.

5. From the list of existing Linux system calls, pick two and implement for MiniOS.

6. Implement a simpler version of the tracealyzer for freeRTOS.

7. Make performance study of the overhead of implementing system calls with synchronous interrupts. For ideas see, for instance, The Performance of µ-Kernel-Based Systems; and FlexSC: Flexible System Call Scheduling with Exception-Less System Calls

[13] Hint here.

[14] Hint here

Chapter 11

Error Handling

We left all that stuff out [error recovery code]. If there's an error, we have this routine called panic, and when it is called, the machine crashes, and you holler down the hall "Hey, reboot it"

Dennis Ritchie

Errar es de humanos. We as humans are bound to make mistakes, and that includes those among us who write OSes, those who write apps, and those who are end users. For an OS this means errors during execution are expected and must be accounted for. Errors include bugs in OS code, bugs in app code, misuse of the computer software/device, and even hardware malfunction—that last one is not necessarily human mistake.

While in real life it can be difficult (even philosophical) to call a mistake, in computer programming it is fairly simple. That is detect them, not avoid them, let alone recover from them. As far as an OS is concerned, there are two type of errors to be detected: fatal system errors and fatal exception errors[1]. Let us begin by looking how to detect errors in the system itself.

[1] What about non-fatal ones? The writer has no idea. Perhaps they are not worth the effort.

11.1 Fatal System Errors

Fatal system errors occur when the kernel detects certain **internal condition**, and thus determines **execution cannot continue**. This type of errors are caused by bugs in the kernel, errors in configuration, missing hardware, corrupted memories, etc.

Different systems act differently upon encountering fatal internal errors. Linux, for instance, raises a **kernel panic**; whereas windows raises a **bug check**. What all (or most) of these approaches have in common is that a) they display some sort of error information[2], and b) they halt execution. As for MiniOS, once we detect an internal error, we will raise a kernel panic with similar behaviour.

How do we detect internal errors? Simple—in code we look for conditions that should not be. As usual, let us look at an example.

11.2 Fatal System Error example

Let us say we are initializing IO and the FAT file system cannot be mounted on the SD Card; either because there is no SD card inserted, or the SD card is not FAT-formatted. Because we cannot access user apps without the SD card, we agree execution cannot continue under such circumstances; and thus raise a kernel panic. The HAL function in question is *hal_nvmem_start*. It returns false when it cannot initialize the specified non-volatile memory, as shown in Listing 11.1

```
/**
*  Non-volatile Memory start
*  ...
*  (at)return true if the memory was initialized correctly, false
    otherwise.
*/
bool hal_nvmem_start( tNVMemId memid ){
        ...
```

Listing 11.1: hal_nvmem_start

[2] In the case of a bug check the blue screen of death is displayed. See here a short history.

So raising a kernel panic is as simple as what is shown in Listing 11.2.

```
void io_init(void){

   //Starts All IO devices
   ...
   if( !hal_nvmem_start( NVMemSDCardFAT ) )
      system_panic( 'FS failed to mount' );
```

Listing 11.2: Kernel panic example

After the panic function is called, the machine has officially *crashed*. The exact details on what happens upon crashing are in in Listing 11.3. (It simply prints the panic message and halts showing an LED panic sequence.)

```
void system_panic( const uint8_t* panic_msg ){

   //report msg
   hal_io_display_cls();
   println( ':( Something went wrong...' );
   println( '' );
   println( panic_msg );

   //Show panic LED sequence.
   bool led_state = false;
   while( true ){
      //toggle all LEDs
      for( uint32_t i=0; i<5; i++ )
         hal_io_led_write( i, led_state = !led_state );

      //pause
      hal_cpu_delay(150);
   }
}
```

Listing 11.3: Kernel panic definition

Crashing is demonstrated in Error Handling—Kernel Panic Demo.

Is that it? Will that catch all runtime errors? No, sometimes things do not go wrong in such a controlled manner. Sometimes crashing is the result of a CPU fault. Before looking at faults, first let us take a look at errors in applications.

11.3 Fatal Exception Errors

Handling errors in applications is different from system errors. First, the kernel cannot catch application errors in the same way it catches those that are internal. Second, when something goes wrong in userland there is no need to crash the machine. We can simply terminate the offending application, and continue *como si nada*[3].

How do we catch errors in user apps? Errors in apps are caught by hardware in the form of **faults**, and they are known as **Fatal Exception Errors**. Next we look at the details.

11.4 CPU faults (or traps)

CPU faults are a type of CPU exception intended to ***trap*** illegal behaviours. The Cortex-M4 implements four faults: **Bus Fault**, **Memory Management Fault**, **Usage Fault**, and **Hard Fault**. Divide-by-zero and unaligned accesses, for instance, trigger a usage fault. Depending on the fault it may be possible to continue execution or not.

How do we know when a fault has occurred? Faults are exceptions, so we know because execution is transferred to an exception handler. Namely, *BusFault_Handler*, *MemManage_Handler*, *UsageFault_Handler*, and *HardFault_Handler*. We already know access to the machine can only occur via the HAL. So, as first step, we enhance the HAL to support faults.

```
//Faults-related definitions
static void (*hardfault_callback)(void);
static void (*busfault_callback)(void);
static void (*usagefault_callback)(void);
static void (*memmanagerfault_callback)(void);
...

void hal_fault_register_callback( tFaultType fault_type, void(*cb)(void)
    ){
   switch(fault_type){
      case FaultHard:         hardfault_callback       = cb; break;
      case FaultBus:          busfault_callback        = cb; break;
      case FaultUsage:        usagefault_callback      = cb; break;
      case FaultMemManager:   memmanagerfault_callback = cb; break;
      default:          /* Error */                    break;
```

[3] As if nothing had happened.

```
    }
}

void BusFault_Handler(void){ (*busfault_callback)(); }
void UsageFault_Handler(void){ (*usagefault_callback)(); }
void MemManage_Handler(void){ (*memmanagerfault_callback)(); }
void HardFault_Handler(void){ (*hardfault_callback)(); }
```

Listing 11.4: HAL support for Faults

In this manner when a fault occurs, the registered callback is executed.

Now, so far no faults have been enabled, so all CPU faults fall back into *hard faults*. Specific faults are enabled via the *System Handler Control and State Register* (*SHCSR*), which is part of the *System Control Block* (*SCB*). Documentation states bits 18, 17, and 16 enable the usage fault, bus fault, and the memory manager fault, respectively. The corresponding code is in Listing 11.5 as part of *faults_init*.

```
//Register offsets SCB SHCR
#define SCB_SHCSR_USGFAULTENA_OFFSET   18 /* pg. 223 ATSAM4SD32C
    Datasheet */
#define SCB_SHCSR_BUSFAULTENA_OFFSET   17
#define SCB_SHCSR_MEMFAULTENA_OFFSET   16
...
void faults_init(void){

   //Enables Usage, Bus, and Mem Manager Fault Exception
   //by setting USGFAULTENA, BUSFAULTENA, and MEMFAULTENA in
   //the SHCSR register part of the system control block (SCB) register
   //(sse pg. 223 ATSAM4SD32C)

   uint32_t shcsr =  hal_cpu_get_scb_shcsr();

   shcsr |= ((1 << SCB_SHCSR_USGFAULTENA_OFFSET)
            | (1 << SCB_SHCSR_BUSFAULTENA_OFFSET)
            | (1 << SCB_SHCSR_MEMFAULTENA_OFFSET) );

   hal_cpu_set_scb_shcsr( shcsr );
}
```

Listing 11.5: Enabling Faults

In addition, the *div-by-zero* trap and *unaligned access* trap must be enabled. Enabling the former lets the CPU trigger a fault when an integer division by zero is attempted (*SDIV* and *UDIV* instructions). Similarly, enabling the latter lets the CPU trigger a fault on unaligned access to half-words. To do this we set bits 4 and 3 in the *Configuration and Control*

Register (*CCR*), also within the *SCB*. Both steps are shown in Listing 11.6.

```
//Register offsets SCB CCR
#define SCB_CCR_DIV_0_TRP_OFFSET 4  /* pg. 217 ATSAM4SD32C Datasheet */
#define SCB_CCR_UNALIGN_TRP__OFFSET 3
...
void faults_init(void){
    ...
    //enables division by zero and unaligned memory access traps
    uint32_t ccr = hal_cpu_get_scb_ccr();

    ccr |= ((1 << SCB_CCR_DIV_0_TRP_OFFSET) | (1 <<
        SCB_CCR_UNALIGN_TRP__OFFSET));

    hal_cpu_set_scb_ccr( ccr );
    ...
}
```

Listing 11.6: Enabling Faults

Although it is not evident from the code, *faults_init* is defined within a new module: faults. This module will be responsible for handling everything related with CPU faults (see Figure 11.1). Additional HAL functions, such as *hal_cpu_get_scb_ccr*, have also been defined. Lastly, we added fault reporting to the console, and extra error reporting code (complete code in the base code for this chapter). For a list of Cortex-M4 faults see pages 3 and 4 of the document Using Cortex-M3/M4/M7 Fault Exceptions.

Catching of app errors is demonstrated with the app DivByZero in Error Handling—Div-by-zero Demo

Next we look at terminating an app after an error has been detected.

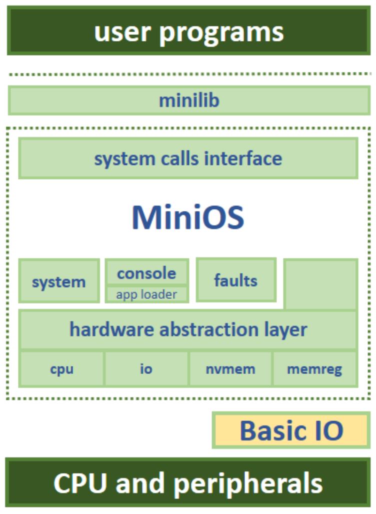


Figure 11.1: Architecture goal for this chapter

11.5 Fatal Exception Errors (continued)

At the moment app errors are handled by *crashing* the machine. Yet it makes more sense to terminate the app that triggered the fault and continue execution (something like this). Because kernel code can also trigger faults, the first step to terminate *misbehaved apps* is to determine where the exception occurred.

A simple way of doing this is looking at bit 3 of *EXC_RETURN*. This "value" is stored on LR on exception entry and has information about the state of the CPU at the moment of the exception. Bit 3, in particular, has a record of which stack was in use. So we use this information. The corresponding code appears in Listing 11.7 to 11.9. Notice we made other changes: a) Fault handlers are now in assembly; b) we removed the *tFaultType*, since we can read the *IPSR* register to see the currently triggered fault; and c) we introduced instead, a *tFaultOrigin* which would server better our purposes[4].

```
//BusFault Handler
.thumb_func
.global BusFault_Handler
BusFault_Handler:
    b faults_goto_right_callback

//UsageFault Handler
.thumb_func
.global UsageFault_Handler
UsageFault_Handler:
    b faults_goto_right_callback
...

faults_goto_right_callback:
    tst lr, 0b100
    beq faults_else                 /* if ( exc_return & 0b0100 ) */
    ldr  r0, =fault_app_callback    /*    temp = app_callback;    */
    b faults_end                    /*                            */
faults_else:                        /* else                       */
    ldr  r0, =fault_system_callback/*    temp = system_callback; */
faults_end:
    ldr r0, [r0]
    bx r0                           /* temp();                    */
```

Listing 11.7: Redirecting faults to the right callback (*hal_cpu_asm.s*)

[4] He who writes is clearly intentionally guiding the reader through his own thinking process, mistakes included. This raises the question of whether this has pedagogic purposes or it is mere laziness (he who writes suspects it is the latter.)

```
//Faults-related definitions
void (*fault_app_callback)(void);
void (*fault_system_callback)(void);
...

void hal_cpu_fault_register_callback( tFaultOrigin faultOrigin ,
    void(*callback)(void)  ){
      switch(faultOrigin){
         case FaultApp:    fault_app_callback = callback;      break;
         case FaultSystem: fault_system_callback = callback;   break;
         default:       /* Error */                         break;
      }
}
```

Listing 11.8: Replaced *tFaultType* for *tFaultOrigin* (*hal.c*)

```
void faults_init(void){
     ...
    //sets callbacks
    hal_cpu_fault_register_callback( FaultApp, faults_app_entry_point );
    hal_cpu_fault_register_callback( FaultSystem,
        faults_system_entry_point );
}
/*
*  App Faults entry point
*/
static void faults_app_entry_point(void){
    //KILL THE APP!!
}

/*
*  Kernel Faults entry point
*/
static void faults_system_entry_point(void){
    //We are the cause of the fault O.o, panic!
    system_panic( make_error_msg( get_error_code() ) );
}
```

Listing 11.9: Different handlers for app and system faults (*faults.c*)

Okay. Now we have different entry points for system and app faults. Any system fault will be considered a *Fatal System Errors*, and will thus end in a kernel panic. As for app faults, all is left is to kill the running app. Unfortunately that will not get resolved until there is a scheduler in place (at least in MiniOS' case).

Demo of the two entry points in Error handling—Kernel and App traps (MISSING).

Resources on the web

- (Video) What happens when divide by zero on mechanical calculator Facit ESA-01
- (Video) Problems with Zero - Numberphile
- (Video) The Design of a Reliable and Secure Operating System by Andrew Tanenbaum
- (Video) Breaking the x86 Instruction Set
- (Video) The Page-Fault Weird Machine: Lessons in Instruction-less Computation
- (Video) Capturing 0Day Exploits With Perfectly Placed Hardware Traps
- (Video) Dennis Ritchie's video interview June 2011 by DennisRitchie.in

Problems

Base code is in here and here. Notice that we have included a start from console or start from app option (see demo). A welcome message screen (see here), and we have removed the witing-for-character in the console. Also there seems to be a bug in display_printf

- Double and triple faults
- Do some reasearch, find a few other faults in the Cortex-M4 and write apps that demonstrate them....
- CHnage panic to also include a register dump
- CHnage panic to also include mem dump, at least the useful part of memory
- CHnage panic to log panic info to a file and automatically restart after 10 secs... show a counter....

- How would you handle dumping/logging memory and registers when the state of the system is not okay. See Kdump
- Raise a kernel panic when a board or sensor or display is not plugged. Add preprocessor defines that tell MIniOS which IO peripherals to expect.

Chapter 12

Multitasking

Simultaneity is merely a subjective impression.

Richard Feynman.

Multitasking is [pending definition]. Introduction is [missing].

12.1 CPU sharing

Computer resources are limited, and so are CPU cores. While some platforms have dozens of cores, others like the SAM4S have a single one. Whatever the number is, if an arbitrary number of apps are to execute simultaneously, they ought to share CPUs.

There are two approaches to the sharing of CPUs: cooperative and pre-emptive multitasking. **Cooperative multitasking** describes the approach where executing apps "voluntarily give up" control of the CPU, which is then assigned to an app waiting for CPU. On the contrary, **pre-emptive multitasking**, describes the approaches where the OS "forcefully takes" control of the CPU from an app in execution, which is then assigned to an app waiting for CPU.

So far we have been using the term *app in execution* to mean exactly that—an app that is running at some given moment. Hereafter we will use the term **process** instead. Although an app in execution is not exactly a process, for the time being it shall suffice.

In pre-emptive multitasking, the part of the OS that coordinates the "taking" and "giving" of CPU time to processes is known as **scheduler** (since it does it on a schedule basis). In general, a good scheduler will fairly distribute CPU time over processes. The exact details of how this happens is know as scheduling policy.

Before getting into scheduling policies, there is one problem of more technical nature that must be solved first. Namely, context switching.

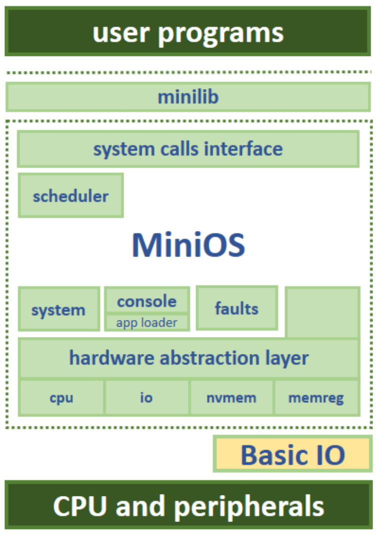


Figure 12.1:

Context switching is more or less the process composed of the following four steps. a) taking of the CPU from the process currently holding it (let us call it the *active process*); b) saving the active process' context; c) restoring the context of the new process selected as active process (let us call it the *new active process*); and finally d) giving control of the CPU to the new active process.

In the remaining of this chapter, we will use these four steps as a basis to write a *pre-emptive process scheduler* for MiniOS (Figure 12.1). Let us begin with the first step.

12.2 Context Switching (the "taking" part)

To support context switching the Cortex-M4 has a built-in *system timer*, or *SysTick* timer in Cortex-M4 jargon. **The system timer enables the kernel to take the CPU from the active process**. How? Well, anytime the system timer *ticks* (i.e. fires), execution is taken to the system timer's interrupt handler, wherein the kernel once again has control of the machine. That is all the magic!

How often should the CPU be "taken" from processes? Typically **the exact amount of CPU time each process gets is defined in terms of *ticks***

and *quanta*[1]. A tick is a system timer's tick. A quantum is the number of ticks a process gets to use the CPU before getting pre-empted; and it is often a small number (2, 5, 10 etc). In our case we use a 1ms tick, and for simplicity a quanta of 1 (so context switch happens on every tick). To define this behaviour we use the HAL's function *hal_cpu_systimer_start*, which takes a tick frequency, and a callback[2]. Code is shown in Listings 12.1 to 12.3.

```c
...
#define TICK_FREQ 3

// Tick Callback
// (No prologue and no epilogue)
__attribute__((naked)) static void tick_callback(void){
}

void scheduler_init(void){
   //starts system timer
   hal_cpu_systimer_start( TICK_FREQ, tick_callback );
}
```

Listing 12.1: *scheduler.c*

```c
//systick-related definitions
inline static uint32_t ms_to_ticks(uint32_t time_in_ms){
   return (sysclk_get_cpu_hz()/1000)*(time_in_ms);
}
void (*systick_callback)(void);

...
void hal_cpu_systimer_start(uint32_t tick_freq_in_ms, void(*cb)(void)){
   systick_callback = cb;
   SysTick_Config( ms_to_ticks(tick_freq_in_ms) );
}
```

Listing 12.2: SysTick in HAL (C part)

```asm
...
.thumb_func
.global SysTick_Handler
SysTick_Handler:
   ldr   r3, =systick_callback
   ldr r3, [r3]
   bx r3
```

Listing 12.3: Systick in HAL (assembly part) (*hal_cpu_asm.s*)

[1] Plural of quantum

[2] As it is usual with exception callbacks, *tick_callback* has no epilogue nor prologue (hence the naked attribute).

Next we continue with steps b and c of the context switching process.

12.3 Context Switching (the “saving” and “restoring” part)

A process’ context is the state of the machine at the moment of a context switch—well, not the entire machine. Only the part relevant to the temporary “pausing” of processes, i.e., CPU registers. There is only one set of registers for all processes to share, including the kernel itself. So they must be saved and restored on context switch. That, of course, only includes CPU registers that are available to processes. **So a process’ context includes those registers accessible in user mode** (see Figure 10.2).

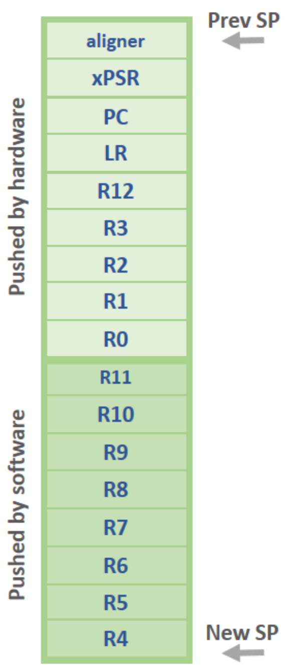


Figure 12.2:

Wait, what about a process’ allocated memory? Shouldn’t that too be stored and restored? Allocated memory includes static variables, automatic variables, and dynamic memory. The interesting part is that, because a process’ memory is *allocated* (See Chapter 3 and 5), it will remain intact during the time a process is “paused”. Unless, of course, there is some sort of bug in the kernel or another process decides to mess with its fellow process. Even in the latter case we can stop processes from accessing each others address space (as we shall see in Chapter 13—Memory Protection).

So, again, to save a process’ context we simply store, on pre-emption, all registers accessible in user mode. If the saving and restoring is done right, a process will never get to know it was temporarily put “out of service”.

The question now is—where do we store them? While we could store them anywhere (say by creating a *struct* ProcessContext), for convenience we store them in the process stack (the stack for apps). To see how this happens, let us consider the case of a context switch on a single running app.

Let us say an app is executing, and a tick occurs. Once execution reaches *SysTick_Handler* (Listing 12.3), we know registers R0-R3, R12, LR, PC, and APSR have been stacked (in *hardware*) as part of the stacking process that occurs on exception entry (Section 6—Interrupts). As it turns out, those stacked registers are half of the pre-empted process' context. The other half (R4-R11) we push them manually onto the stack, *in software*. The process stack after both hardware and software stack frames have been pushed is depicted in Figure 12.2. The code for it is on Listing 12.4.

```
...
.thumb_func
.global hal_cpu_save_context
hal_cpu_save_context:
     mrs r0, psp
     stmfd r0!, {r4-r11}
     msr psp, r0
     bx lr

.thumb_func
.global hal_cpu_restore_context
hal_cpu_restore_context:
     mrs r0, psp
     ldmfd r0!, {r4-r11}
     msr psp, r0
     bx lr
```

Listing 12.4: Saving/Restoring a process' context (*hal_cpu_asm.s*)

That is it! Next we discuss how the machine is given back to the new active process.

12.4 Context Switching (the "giving back" part)

Okay, the active process was running, a tick occurred, CPU was taken from it, and a new active process has been selected. To give back (or for the first time) the CPU to the new active process, we simply have to return from *SysTick_Handler*. Returning from an exception is a matter of loading PC with the EXC_RETURN for the exception being handled. EXC_RETURN has information about the processor mode and the active stack before exception entry; and it is available on LR[3].

Now, because we know processes always run in user mode using the process stack, there is no need to save the EXC_RETURN value in LR on pre-emption, and then restore it on a process next execution time. We can simply write the value *0xFFFFFFFD* every time, as shown in Listing 12.5 and 12.6.

```
.equ USER_MODE_EXEC_VALUE, 0xFFFFFFFD
...
hal_cpu_return_exception_user_mode:
      ldr pc, =USER_MODE_EXEC_VALUE
```

Listing 12.5: Returning from a context switch (*hal_cpu_asm.s*)

```
...
__attribute__((naked)) static void tick_callback(void){
      hal_cpu_save_context();

      //switch active thread
      // ...

      hal_cpu_restore_context();

      hal_cpu_return_exception_user_mode();
}
```

Listing 12.6: Returning from a context switch *scheduler.c*

Wait, wait! Why can't we just return from the exception handler, without calling *return_exception_user_mode* (a process will always run in user mode and using the process stack, so on every tick the value 0xFFFFFFFD will be in LR anyway)? Not entirely sure, I think it's because on the first tick the machine will be executing kernel code.

Finally, to start the scheduler we call (at the very last) *scheduler_init()* from within *system_init*. Despite not having any means to create processes, we can still test context switching. If everything is working correctly, context switching should pass unnoticed.

For a demo[4] of everything so far see Multitasking—Context Switch Demo

[3] Note the EXC_RETURN value in LR is different from the *stacked* LR value. The stacked LR value contains whatever was on LR before the interrupt.

[4] This demo shows an older version of MiniOS, where apps used to be compiled to-

12.5 Processes

The concept of process is abstract and can mean slightly different things on different contexts. As far as we are concerned, a process is an instance of MiniProcess (Listing 12.7).

```c
...
typedef struct{
     uint8_t* name;
     uint32_t* sp;
     tProcessState state;
}MiniProcess;
```

Listing 12.7: Processes in MiniOS (*scheduler.c*)

Processes have a name, a stack pointer, and, at all times, they are in one of more possible states. At the moment a process' state can be either ready or running, as shown in Figure 12.3.

To create processes we will have a system call, *process_create*, defined as part of the scheduler. This system call will take the path to a binary, a name, and the process' stack size in words (see Listing 12.8).

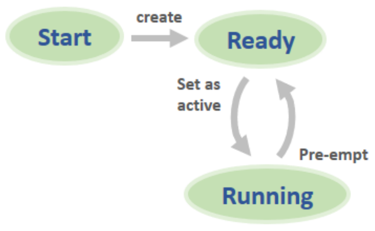


Figure 12.3: Transition diagram for a process state.

```c
...
int main(void){
     process_create( 'Joanna process', 'App.bin', 512  );
}
```

Listing 12.8: A system call for process creation

In the following we expand what we have as scheduler to support one process; and we approach this by attempting to implement *process_create*.

gether with MiniOS; which makes it more instructive, since the reader can see execution going back to the app.

12.6 A one-process scheduler

Starting with the obvious, we write the incomplete definition in Listing 12.9

```
...
MiniProcess proc;

void scheduler_process_create( uint8_t* binary_name,
                               uint8_t* name,
                               uint32_t stack_sz ){
    proc.name = name;
    proc.state = ProcessStateReady;
    proc.sp = ...
}
```

Listing 12.9: First definition of *process_create* syscall (*scheduler.c*)

This raises a question—where is the stack for some given process? So far nowhere. We have not allocated any stack space for any process. When and how do we allocate space for a process' stack? The *when* is at creation time. The *how* is trickier, and there are several ways to approach this. Yet we will do it in the simplest way the author was able to imagine: by placing all stacks contiguously, one after another. The details are discussed next.

12.6.1 Process stack allocation

Let the end of the whole process stack[5] be *epstack*, the end of the stack for a process *i* be *epstack[i]*, and the stack size in words for a process *i* be *stack_size[i]*. Initially, before the first process (process 1) is created, the process stack is empty. So the SP for process 1 is *epstack=epstack[1]*. Following a one-after-another layout, the SP for process 2 must be *epstack[1] - stack_size[1]*; and, in general, *epstack[i-1] - stack_size[i-1]*, up to

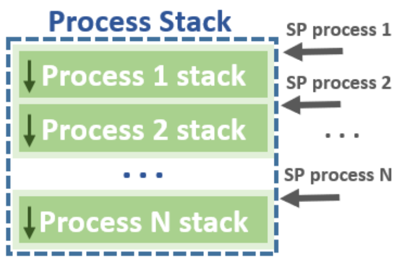


Figure 12.4: Processes' stacks

[5] Recall the *process stack* is the stack we have been using to run apps in user mode. It had nothing to do with actual processes, until now.

the point where no more space is available. This layout is depicted in Figure 12.4.

Listing 12.10 shows how space is allocated in code. Note it happens in two steps: 1) we call *stack_top* to set the SP for process *i* to the top of the stack allocated so far (again, *epstack[i-1] - stack_size[i-1]*); and 2) we call *stack_alloc*, which moves down the same pointer by *stack_size[i]* words, i.e. *epstack[i] - stack_size[i]*.

```
MiniProcess proc[PROCESSES_MAX_NUM];
...
void scheduler_process_create( uint8_t* binary_name,
                               uint8_t* name,
                               uint32_t stack_sz ){
   ...
    //Allocate space
    proc[i].sp = stack_top(); //set SP
    stack_alloc( stack_sz );  //make space
}
```

Listing 12.10: Allocating stack space for a process (*scheduler.c*)

Once SP has been set correctly, when a process executes for the first time it will do so using its own stack.

Everything seems to be in place now. Stack is allocated for a process and its SP is set accordingly on creation. Then, on context switch, the active process' context is saved and the new active process' context is restored. The next step, and perhaps the *trickiest* part, is to execute the new active process.

12.6.2 Executing the active process

Consider how processes get to run. Processes run because a *tick* occured and it was determined they were next. So

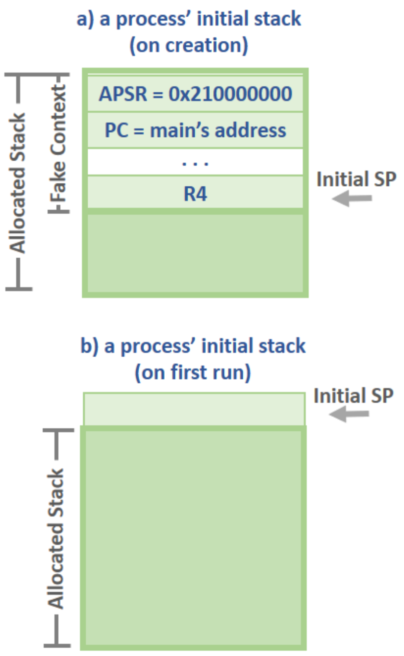


Figure 12.5:

processes run on return of the *SysTick_Handler*, or on return of *tick_callback* (Listing 12.6), however the reader prefers to see it.

Let us consider the base case. How does a process get to run the first time? The first time a process runs is because *process_create* was invoked. So, should a new process run right after *process_create* returns? Maybe not. Maybe we will simply place the new process in a waiting list so that it runs on its next turn. That seems reasonable.

Now, when looking at it closer, one can see the base case is similar to the general case, except for one detail. **The first time a process runs, there is no hardware nor software context to restore**. Hardware and software contexts are stored on pre-emption, and a newly created process has not been pre-empted yet. Garbage will be restored as context instead. The solution is simple—**manually insert a hardware and software context for newly created processes**.

What register values shall we place in this "fake" context? For most registers it will not matter, the same way it did not matter when we were running single apps. It will only matter for PC and bit 24 in APSR. PC tells where execution should continue after a context switch, and APSR[24] is the *thumb state bit*, which indicates in what *thumb mode*[6] the processor is.

In other words, initially, when a process is created, a fake context will be placed as part of the allocated stack, thereby simulating a pre-emption (Figure 12.5a). Then on the first tick, on *tick_callback* return (after destacking), the same process will start with an empty stack (Figure 12.5b).

Finally, there is one detail left: SP must also be stored/restored on context switch. The code with everything discussed so far appears in Listing 12.11. (Note there is extra code for loading the binary from permanent storage, among other shenanigans.)

```

#define TICK_FREQ       3
#define CONTEXT_SIZE    16
#define INITIAL_APSR    (1 << 24) //Bit 24 is the Thumb bit
#define OFFSET_PC       14
#define OFFSET_APSR     15

```

[6] More details in Section 2.3.1, and in pg. 4 of Cortex -M4 Devices, Generic User Guide.

```
...
static MiniProcess proc[1];          //processes
static MiniProcess waiting_list[1]; //dummy waiting list
static MiniProcess active_proc;      //The active process
static uint32_t process_count = 0;

//A null process (used to mark the lack of an active processs)
static MiniProcess null_proc = {
         .name = 'null',
         .state = ProcessStateNull
   };

/*
*  Scheduler Init
*
*   Initializes the scheduler. The system timer is not started here.
*/
void scheduler_init(void){
   //Init process stack
   static tMemRegion stack_memreg;
   hal_memreg_read( MemRegUserStack, &stack_memreg );
   stack_init( stack_memreg.base ); //stack_init( epstack )

   //Active process is the null process
   active_proc = null_proc;
}

__attribute__((naked)) static void tick_callback(void){

   //save software context
   hal_cpu_save_context();

   //Is there an active process?
   if( active_proc.state != ProcessStateNull ){
      //save SP
      active_proc.sp = hal_cpu_get_psp();

      //place active process in dummy waiting list
      waiting_list[0] = active_proc;
   }

   //get next active process from dummy waiting list
   active_proc = waiting_list[0];

   //restore SP
   hal_cpu_set_psp( active_proc.sp );

   //restore software context
   hal_cpu_restore_context();

   //give CPU to active process
   hal_cpu_return_exception_user_mode();
}

/*
*  Scheduler Process Create
*
*  Creates a process from a binary in nvmem. Ticking begins here!
*/
```

```
void scheduler_process_create( uint8_t* binary_file_name,
                               uint8_t* name ){
   tMemRegion proc_memregion;
    uint32_t stack_sz;

   //Load app binary
   if( loader_load_app( binary_file_name, &proc_memregion, &stack_sz )
       != LOADER_LOAD_SUCCESS )
      return SCHEDULER_PROCESS_CREATE_FAILED;

   //Set process info
   proc[0].name = name;
   proc[0].state = ProcessStateReady;

   //Allocate space for 'fake' context
   stack_alloc( CONTEXT_SIZE );

   //Allocate space for remaining stack
   proc[0].sp = stack_top();                //set SP
   stack_alloc( stack_sz - CONTEXT_SIZE );  //make space

   //Insert 'fake' context
   proc[0].sp[OFFSET_PC] =     ((uint32_t) (proc_memregion.base +1));
   proc[0].sp[OFFSET_APSR] =   ((uint32_t) INITIAL_APSR);

   //Add new process to dummy wait list
   waiting_list[0] = proc[0];
   process_count++;

   //Start ticking on first process
   if( process_count == 1 ){
      hal_cpu_set_psp( proc[0].sp );   //or else the first tick fails
      hal_cpu_systimer_start( TICK_FREQ, tick_callback );
   }

   return SCHEDULER_PROCESS_CREATE_SUCCESS;
}
```

Listing 12.11: One-process scheduler (*scheduler.c*)

That is it. If everything went right, we should be able to create a process, and see how execution goes as if there was no scheduler, as demonstrated in Scheduler—One-Process Scheduler Demo (Time is the demo app).

12.7 Comments on the one-process scheduler

Okay this is not over yet. There are a few things to comment here. First, the *system timer* is not started util after a process is created (within *sched-*

uler_process_create itself). This simplifies *tick_callback* as it ensures no ticking can occur when no processes exist, or while one is being created.

Second, a process' initial SP is no longer being passed from the App to the OS in the binary (as we were doing before). Why? Well, after some thought, the author realized the stack can be placed anywhere. Unlike references to static variables, references to stack variables are always relative. **As long as enough space is left empty (i.e. allocated), the stack can be placed anywhere**. In particular, in the *UserStack MemReg* (Figure 12.6). How much space should it be allocated for a process? We will let the app decide that. So instead of the app passing the stack location to the OS, the app will pass the stack size, as shown in Listing 12.12.

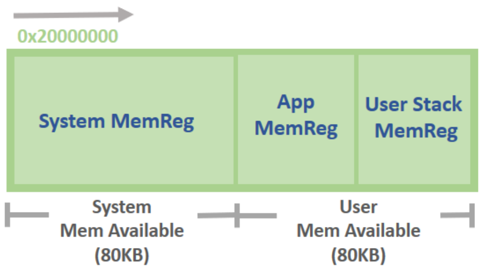


Figure 12.6:

```
...
#define MINIOSAPP __attribute__ ((section('.entry_point'))) uint32_t
#define STACKINFO __attribute__ ((section('.stack_info'))) uint32_t

STACKINFO stack_sz = 512;

MINIOSAPP main(void){
...
}
```

Listing 12.12: Passing App's stack size to OS

Lastly, memory regions (*hal_memregion.c*) were modified to reflect the memory layout in Figure 12.6.

12.8 Running Apps as processes

To run apps as processes, the loader was modified[7], *loader_exec_app* was replaced by *loader_load_app*. MiniOS' *main* (Listing 12.13). Note an extra parameter *loader_rval* was also added to *scheduler_process_create*. This

[7] Recall the loader loads a binary in *nvmem* to the App's MemRegion (Figure 12.6).

param is populated with the return value from the loader when the loader fails.

```c
int main(void)
{
    //Initializes the system
    system_init();

#ifdef SYS_INIT_FROM_CONSOLE
    //Execute the console
    console_begin();
#else
    //Create process with default app
    uint32_t loader_rval;
    uint32_t sched_rval;

    sched_rval = scheduler_process_create( 'App.bin',
                                            'Johanna Process',
                                            &loader_rval );

    if(  sched_rval == SCHEDULER_PROCESS_CREATE_FAILED ){
        ...
    }
#endif

    //Wait here until the timer's first tick
    //(It also prevent from executing garbage, if the console returns)
    while(1);
}
```

Listing 12.13: *main.c*

Likewise the console had to be changed. Complete code available in base code for this chapter. For a demo see Multitasking—Running apps as processes

12.9 Final considerations

Support for multiprocessing in MCUs is rare. As far as the author knows, there are no other MCU operating systems with support for multiprocessing. MiniOS processes are not like other OSes' processes.

Resources on the web

- (Paper) When Poll is Better than Interrupt

Problems

Base code is here and here.

1. Implement a *sleep* syscall that enables processes to go to sleep for a specified time in ms. Because now it can be possible that no process is ready for execution, we need to put the CPU to sleep when that happens. Typically to achieve this schedulers have an idle process, which code is something on the lines of *while(true) cpu_sleep();* [8]. Demonstrate with an app.
2. Write a round-robin n-process scheduler. Demonstrate.
3. Enable processes to receive command-line arguments and to return values. Demonstrate with an app.
4. Enable processes to return (i.e. die). Also enable the console to continue executing after an app invoked from the console finishes running. Demonstrate. (Note for me: see Old Scheduler's thread_delete) (Note for me: I think we want to run the console as a PROCESS THAT RUNS IN USER MODE (EXCEPT THAT ITS CODE IS IN FLASH)!!!)
5. Extend problem 2 and introduce quanta into the scheduler to enable priority-based scheduling. See here

Advanced Exercises:

1. Using the stack-assign strategy discussed in this chapter, it is easy to see how stack space from dead processes cannot be reused. Fix this problem by extending problem 2 to support reusing stack space that was left free by dying processes.
2. Currently the HAL busy-waits on input that is not ready. For instance, calling *serial_getc* in userland terminates in the CPU busy-waiting in kernel mode inside *hal_io_serial_getc()*. Modify the HAL and IO system calls so that reading input that is not available makes a process go to sleep. Let the kernel wake up that process once the requested input is available.

[8] Feel free to use *hal_cpu_sleep* in *hal_cpu_asm.s* to put the CPU in sleep mode

Chapter 13

Memory Protection

> A C program is like a fast dance on a newly waxed dance floor by people carrying razors
>
> Waldi Ravens.

(See example segfault here).

[1]

Anatomy of a program, there's kernel code in sstems with virtual memory XKCD segfault image

Hackers Remotely Kill a Jeep on the Highway—With Me in It (Video)

Add this one later... when ther's more stuff...

MINIX 3: a Modular, Self-Healing POSIX-compatible Operating System.

[1] This is also the case in systems with virtual memory, see Anatomy of a Program in Memory

Chapter 14

Wireless Network Stack

It requires a much higher degree of imagination to understand the electromagnetic field than to understand invisible angels. ... I speak of the E and B fields and wave my arms and you may imagine that I can see them ... [but] I cannot really make a picture that is even nearly like the true waves.

Richard Feynman

[INTRO MISSING]

14.1 Choosing a network device

For computers to "form networks", they must have some sort of IO device capable of communicating with other similar IO devices, so that, by means of these devices, they can "link" with each other and create networks. This IO device may have many forms. In Desktop computers, for instance, it is called the WNIC (wireless network interface card), since it is, well, the "interface" to some computer network. In most cases, a home LAN, which in turn is linked to the Internet.

Okay, so then, all we need for MiniOS is a WNIC. In some sense—yes. We do need a communication peripheral by which MiniOS devices can link with each other. On the other hand, in MCUs there is no concept of "one single device" to connect to the "one network" (the Internet). Instead, there is a large variety of hardware devices based on a variety of protocols (IEEE 802.15.4, Zigbee, LoRA, Bluetooth, WiFi, propietary, among others) at different frequencies (433Mhz, 900Mhz, 2.4Ghz). One can even choose the difficulty of the interface. Some devices are intended more for industry and their use requires expertise, whereas some others are more for enthusiasts and require less. Some devices do "more" as to free MCUs from network processing, and some do less. Like that, there are other variables to consider. In either case, as Schwartz has argued[1], having too much choice is not necessarily a good thing.

So, back to the original question, which device? Well, what type of "networks" we want to create or join. We would like to eventually use MiniOS to create IoT devices. So we need access to the internet. The simplest way to access the internet is via WiFi—so WiFi it is. We would also like to have the ability to form *ad hoc* networks. For this the author has two preferences, IEEE 802.15.4 and LoRA. They both have relatively large ranges, are relatively easy to use, and are low power. So maybe we support both? No, too much work. LoRA is also intended for IoT, has the largest range of the two, and let us face it, there is a hype for Lora at the moment. Not to mention, that there is a vendor that offers both WiFi and LoRa devices with a very similar interface. Again, let us save some life—LoRa it is[2]. In particular, the WiFi and LoRa device from Figure 14.1.

Wait, wait, what about the price per device? Let see. Both the WiFi device and the LoRa device have a cost of $35. For WiFi that is a good price. For a low-power low-rate transceiver, $35 is fine for one, but not so much for, say 7, or 10. Well, as it turns out the vendor offers a $24 900Mhz radio (not LoRa, though) with the same interface. In principle

[1] The paradox of choice.

[2] The reader would be surprised of how much of design in Engineering is "choosing" (be it device, software, tool, platform, version, method, protocol, frequency, library, environment, language, so on), and convincing other engineers that their *better choice that will solve all world problems* is as random and inadequate as that of oneself. **At the end Engineering design is all about tradeoff, and the personal evaluation of that trade-off (we value different thing sdifferently)**

that means we can have three different radios WiFi, LoRa and a cheaper 900Mhz without having to rewrite or relearn stuff.

If the purpose is low price, there exist much cheaper radios, why not use them? Well yes, but three radios one interface beats a cheaper transceiver (at least the author's biased appreciation).

Okay fine. But is it guaranteed that they have the same interface, no surprises? No it is not guaranteed. We presume, based on some experience and some "internet research". Yet there is hope, because this is where another engineering principle (Agile principle actually) comes in handy: If you are going to fail, fail fast. So our next step is to write a common interface for the three devices discusses in this section

14.2 A common interface for network devices

14.2.1 LoRa radios and 900Mhz

14.2.2 WiFi Device

Adafruit Feather M0 WiFi w/ATWINC1500

https://www.adafruit.com/product/3010

14.3 A Common Network Abstraction

Since we'll have more than one radios to support and to support future use of different radios, we will use a common network abstraction (same as we did with the HAL).

14.4 Trickle as network protocol

14.5 distance-aware network

14.6 Zero-copy network

14.7 The 1s and 0s

Sniffing bytes with a wireless logic analyzer.

Cool demo.

14.8 Context—ad-hoc embedded wireless networks

Somehow connect it to IoT and swarm

As exercise, show a wireless buffer overflow exploit

Appendix A

ASF as Basic IO

This appendix contains general information on how to integrate device drivers and other low-level software part of the Atmel Software Framework (ASF) into an Atmel Studio solution. This includes background theory on IO peripherals and working examples. Before continuing the reader is encourage to get acquainted with Chapter 4 (IO).

The Atmel Software Framework is a compilation of bare-metal software for Atmel MCUs. Next it is explained how to set up a new ASF template project.

A.1 Working with the ASF

To start working with the ASF it is required a new ASF template project, as shown in Basic IO - Creating an empty ASF empty project, and Basic IO - Linker script.

ASF code is organized in modules that are composed of a set of source files that add functionality to the project. Sometimes some modules depend on others, so one has to be careful to fulfil dependencies. Adding or removing modules is done through the ASF wizard. See Basic IO - Adding Removing ASF modules.

Before getting to the actual examples there are a few considerations you need to be aware of.

A.2 Final considerations

While the use of different IO peripherals require calls to different routines, there are two routines that are common to all of them: *sysclk_init* and *board_init*. The former does basic low-level initialization of the board, where the latter initializes the system clock i.e. the one running at 120 MHz. They will be the first two lines of code in every example, and because *sysclk_init* initializes the clock, it must be called first.

Finally, recall peripherals can be (a) built in the MCU itself, (b) built onto the board, or (c) external. For instance, Parallel IO (GPIO) is built into the MCU, LED0 is part of the board, and anything in an expansion board (e.g. OLED display) is external. **Also recall that on-board, and external devices, interact with those built-in, as the built-in ones are the ones software can access**. (If this is not clear review chapter 3—Memory).

A.3 On-chip examples

This section includes on-chip peripheral. Those peripherals part of the ATSAM4SD32C MCU.

A.3.1 Parallel IO

Allows control and configuration of Parallel IO. Most of the MCU pins are mapped to one of the many IO Lines: PA0, PB3, PC7, so forth. Depending on the context Parallel IO may be called General Purpose IO (GPIO), Parallel Port, et cetera.

Demo

SAM4S Xplained Pro GPIO Demo

A.3.2 Real-Time Clock (RTC)

The RTC peripheral integrates a clock and a Gregorian calendar with programmable interrupts (alarms). The RTC uses an external 32.768 kHz crystal oscillator, so it operates independent of the system's main clock. In your board it is the little crystal next to the MCU chip:

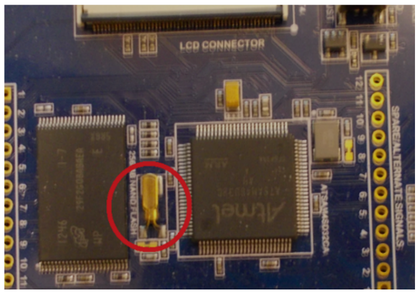


Figure A.1: *On-board 32 Khz Oscillator*

Setting up the RTC requires switching the system's slow clock to take the 32.768kHz external crystal as source, as it is initially connected to a not-very-precise internal oscillator. Then wait until the slow clock is ready and stable.

Demo

SAM4S Xplained Pro RTC Demo

A.3.3 Real-Time Timer (RTT)

The RTT start counts in steps of near 30us ($1/2^{15}$). So for instance, a steps parameter of 2 will make the timer fire near every 60us. Similarly, a steps parameter of 32767 ($1/2^{15} - 1$) will make the timer fire every second.

Demo

SAM4S Xplained Pro RTT Demo

A.3.4 System Timer (SysTick)

The SysTick is a timer intended to be used by an OS, hence its name. Unlike other timers, the SysTick is part of the Cortex-M4 CPU itself.

Demo

SAM4S Xplained Pro SysTick Demo

A.3.5 PWM

[Explanation in progress] In the meantime... Pulse-Width modulation (PWM) is a technique for controlling power given to ”something”. You can use it to dim a LED or a lamp, control the temperature of a coffee heater, or control the speed of a motor. See this tutorial.

It is also used for controlling Hobbie servomotors (motors that can stay in a steady position). Though in this case, power is not being controlled. Hobbie servomotor vendors have made their interface to be PWM, and it has become the standard.

Demo

SAM4S Xplained Pro PWM Example

A.3.6 Universal Asynchronous Receiver Transmitter (UART)

The UART is a serial transceiver. Simple to use in comparison to other serial interfaces. [In progress...] In the mean time... You would want to use this for machine-to-machine communication, either MCU to MCU or MCU to PC. Say you want to make a Java app talk to the MCU (via Virtual COM ports), or make your MCU talk to a sensor that uses UART as interface.

The SAM4S has one UART go to the Debug USB port, so communications going through Debug USB is not USB (i.e. USB protocol), but UART. This means that from the host side (the PC side) it is possible to off-the-shelf talk to the MCU with a serial terminal (you can download one from the Atmel Gallery). Another option is to use a serial library for some programming language, say Python, and write something that talks to the MCU, though that normally requires more trial and error.

Demo

SAM4S Xplained Pro UART and USART (examples with Interrupts)

A.3.7 Universal Synchronous Asynchronous Receiver Transmitter (USART)

[In progress]... Meanwhile... USART is similar to a UART, except it supports synchronous communication (i.e. with a clock line), so it is slightly more complex than UART. Other than that I do not know how they differ, as every time I've used USART I've used it as UART.

Demo

SAM4S Xplained Pro UART and USART (examples with Interrupts)

A.3.8 Analog-to-Digital Converter (ADC)

An Analog-to-digital converter (ADC) converts an analog voltage to a digital representation of it. Computers only understand discrete values such as 0001, 0010, 0011 (think of MOV, LDR, STR, ADD). An analog voltage on the contrary exists in the physical world and it is not discrete; depending on what device we use to measure it, we can get measurements such as 3.7v and 1.00000071v. As such when we do an ADC conversion, we talk of a range values (min and max), and a resolution (the number of discrete steps). For example, let us say we want to do a digital conversion of a value between 0v and 5v with 2-bit resolution ADC. With four possible steps we would get the following relationship: 00 = 0 to 1.25v, 01 = 1.25 to 2.5v, 10 = 2.5v to 3.75v, and 11 = 3.75v to 5v.

Analog to digital conversion is useful because the output interface of many sensors is an analog value. These sensors output different voltage levels in function of the amount of physical ”something” in the transducer (the piece of semiconductor converting the physical something, say air pressure, into electricity). This voltage level is then transformed into a digital representation by the ADC in the MCU, where it is read from the ADC interface registers by the ADC drivers.

The ATSAM4SD32C integrates a 12-bit ADC peripheral, which is multiplexed into 16 ADC lines. Each line has a conversion rate of 1 MHz (i.e. a conversion takes microsecond to complete). 12-bit ADC means the digital value converted can be between 0 and 4096.

Demo

SAM4S Xplained Pro ADC Demo

A.4 On-board and expansion boards examples

This section includes peripheral part of the SAM4S board, and its expansion boards.

A.4.1 Buttons

Unlike other GPIO-based devices, mechanical buttons are peculiar: when pressed, they bounce. We like to think that when a button is pressed it will change the IO line's state and when the button is released its state will go back. Something like this:

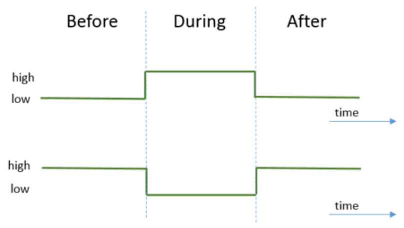


Figure A.2: Button Ideal

The physical world is never that ideal, however. When a mechanical button is pressed it bounces, therefore generating a train of pulses instead of just one.

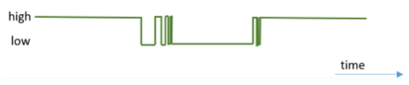


Figure A.3: Button real

This is then interpreted as the button being pushed several times. The ATSAM4SDC32 has hardware support for de-bouncing, which allows to filter pulses which duration is less than a specified threshold. This will not eliminate all the glitches, but it will make it much better; so expect a few of them when you press buttons. Another way is to do it by software, but this requires intervention of the CPU. The idea is the same, whenever a change in state is detected in an IO line, check the IO line again a few milliseconds later; if the state is the same then the button was pressed, else it was a glitch. Maybe even check the IO line in several occasions after the first pulse and determine that the button was pressed only when the state of the button was the same in all the occasions (an example of this is shown in one of the Parallel IO entries... the one with the movement sensor).

Additional Material: This and this.

Demo

SAM4S Xplained Pro Buttons Demo

A.4.2 SSD1306 Controller (OLED Screen)

The display in the OLED1 extension board is a 128x32 pixel white monochrome OLED Display. It is driven by a SSD1306 display controller from Solomon Systech. It interfaces with the MCU via SPI (serial peripheral interface).

Demo

SAM4S Xplained Pro OLED Demo

A.4.3 LEDs

See GPIO

A.4.4 Light Sensor

See ADC

A.4.5 AT30TSE75X (Temperature Sensor)

No description for now.

Demo

SAM4S Xplained Pro Temperature Sensor Demo

A.4.6 BNO055

A 9-axis orientation sensor. It is a neat sensor that applies filtering and some other fancy techniques to raw acceleration, gyroscope, and compass data, so as to get an absolute orientation value in euler angles.

Demo

SAM4S Xplained Pro BNO055 Demo

A.5 External third-party examples

These are examples from non-Atmel external peripherals.

A.5.1 Xbee Series 1

[In progress]A wireless transceiver that implements the IEEE 802.15.4 protocol (think of it as a smaller Wifi). The protocol is layer 2 (up to the MAC layer). That means that with an Xbee, provided there are available drivers, it is possible to do off-the-shelf point-to-point communication. However, more proper MAC layer communication (with software MAC frames) requires additional work.

Demo

Xbee MAC Interface

A.5.2 Piezo Buzzer

See PWM advanced Demo.

A.5.3 Motion Sensor

See Parallel IO advanced demo.

A.6 Other Low-level Software

Delays

Busy-waiting delays work by making the CPU do something until the desired delay time has elapsed. Note this approach is not efficient as it wastes CPU time that could be going to something else, but if your application does not care about power consumption, or CPU utilization, a busy-waiting delays will do the work.

Demo

SAM4S Xplained Pro Delays Demo

FAT System

EDIT THIS ENTRY... add an actual example, and mention there's a bug in a newer version of the ASF that requires a SD CArd to be present in the back

FatFS is a third-party generic FAT file system implementation, which has been made part of the Atmel ASF. This is its official website containing its API and additional examples.

If what you want is to write your own file system, and not use FatFS, look at the SD card and Mini SD Card entries.

Demo

SAM4S Xplained Pro FatFS Demo

Bibliography