Programming a toy computer from scratch

A practical introduction to computer systems

Eric Bruneton

Programming a toy computer from scratch

A practical introduction to computer systems

Eric Bruneton

Version
This book was built from commit 1fc6f0aca63fba948c923e9c8a35ccc3452296da on August 19, 2024 in the source code repository. The latest version can be downloaded at https://ebruneton.github.io/toypc/.

Source code
The source code of this book, in LaTeX and Rust, is available at https://github.com/ebruneton/toypc.
The programs it describes are also available separately, at https://ebruneton.github.io/toypc/.

Feedback
Please report errors or potential improvements at https://github.com/ebruneton/toypc/issues.

Contents

Introduction

Billions of people are using computers or smartphones, which are computers before being phones. One doesn't need to understand how computers work to use them, but if you want to know, this book might help you.

Popular science books about this topic intentionally leave out many details. On the other hand, textbooks emphasize theoretical aspects and focus on a narrow topic. This book is different. Its goal is to introduce how computer hardware and common programming languages and operating systems work, via a practical example which can be understood down to the smallest detail.

For this it proposes you to assemble and program your own toy computer. And to make sure not to omit any details, it explains how you can do this from scratch, without using any existing programming tool. It is organized in four parts:

- the first part briefly presents the main basic ideas used to design microprocessors, which are the core component of a computer. This is necessary to understand the main concepts used in the next parts. It ends with the presentation of a virtual, toy microprocessor, which can be simulated online, and of a few programs using it.
- the second part explains how the components of your toy computer work, how they can be programmed, and how to assemble them. Based on this, it then describes how to build a basic system allowing programs to use the computer's keyboard and screen. Finally, it presents how an initial program can read and execute other programs, based on this input and output system.
- the third part explains how a computer can be programmed in a language which can be "easily" understood by humans, unlike the 0s and 1s used by its microprocessor. For this it describes how to progressively build a toy language, and a program which can translate it into 0s and 1s that the microprocessor can execute. In order to give you an idea of what common programming languages look like, this toy language is an extremely simplified version of real and popular ones.
- the fourth part explains how users can easily store files and launch applications on their computer, thanks to a (set of) program(s) called an operating system. For this it describes how to progressively build a toy operating system for your toy computer. For the same reason as above, this system is an extremely simplified version of real, frequently used ones.

Target audience

This book is designed for people looking for a practical and fully detailed example introducing how microprocessors, programming languages and operating systems work. It does not explain the theories and principles behind this. If you want to

learn them, you should read computer science textbooks instead (some references are provided at the end of each part). Conversely, if you only want to understand the general ideas, it is better to read popular science books instead.

How to read this book

You can read this book without actually assembling or programming a toy computer, just to understand how this could be done. In this case you can skip the tutorial-like sections, which describe concrete steps to follow (plug this wire here, type this on the keyboard, press this button, etc). This is the case, in particular, of the "Experiments" and "Compilation and tests" sections.

Alternatively, you can read this book while following the instructions on an emulator. In this case you do not need to assemble a toy computer, nor to buy the necessary components for this (described in Appendix A). Instead, simply use the emulator provided at https://ebruneton.github.io/toypc/emulator.html. For this you need a desktop or laptop computer (tablets and smartphones are not really usable for this task).

Finally, you can buy the components, assemble them, and follow the instructions for real. This is more costly but probably more fun than the previous methods. This method also requires a desktop or laptop computer, with a USB port and capable of running python3. You can also use all three methods: start with the first one, then re-read the book with the second method and optionally with the third, if you feel that you need to (typing a program is slower than reading it – this can trigger some questions, and finding the answers yourself can give you a better understanding).

Note also that, if you are stuck or simply want to skip some steps, you can follow the instructions of any chapter without doing those of the previous ones (once the computer is physically assembled, if you choose this method). Hence, for instance, you can skip the instructions of part 2, do those of part 3 on the emulator, and those of part 4 for real. You can also already have a look at the final programs and operating system obtained at the end of this book, on the emulator, by opening the following link: https://ebruneton.github.io/toypc/emulator.html?script=backups/final.txt. See the companion website of this book for more details (https://ebruneton.github.io/toypc/).

PART

1 A Toy Microprocessor

Introduction

Before programming a toy computer from scratch it is useful to have some basic ideas about how computers work. For this, one method is to build a toy computer from scratch. It is possible to build one for real from individual electromagnetic relays [5] or transistors [18]. But this requires a lot of time and space, and a significant budget. Moreover, the resulting computer would be too small to run or even store the toy programs of this book. For this reason, this part presents how a toy computer *could* be built, but does not give all the details necessary to physically build one. It is organized as follows:

- Chapter 1 briefly presents binary numbers, which are the basis of how computers work, and how to compute with them.
- Chapter 2 explains how an electric circuit can compute additions and subtractions of binary numbers.
- Chapter 3 shows how loops in circuits can be used to memorize numbers, and in particular the (intermediate) results obtained with the above arithmetic circuits.
- Chapter 4 shows how a circuit can control another, in order to make it perform a series of computations, specified by a *program*.
- Chapter 5 puts everything together to obtain a toy microprocessor, and shows how it can be programmed with a few examples.

Note Most of the circuits presented in this part are also available on CircuitVerse (https://circuitverse.org/), an online digital circuit simulator. Thanks to it you can interact with the circuits presented in this part, which helps getting a better and more practical understanding of how they work. See the companion website of this book for more details (https://ebruneton.github.io/toypc/).

CHAPTER

1 Binary Numbers

As its name implies, a computer performs computations, on numbers. A number is an abstract concept which can be represented in many different concrete ways. For example, the number of days in a week can be represented with "seven", "7", "VII", etc. Some representation methods, also called numeral systems, are more practical than others to perform computations. For instance, doing additions and multiplications is easier in the arabic numeral system than in the roman one. In fact they are even easier to do in the so called *binary numeral system*. Computers use it for this reason. In order to understand how they work it is thus necessary to know first what binary numbers are, and how to compute with them. This is the goal of this chapter.

1.1 Binary numbers

An arabic number such as 237 represents 2 times 100, plus 3 times 10, plus 7 times 1. In mathematical notation this gives

$$237 = 2 * 100 + 3 * 10 + 7 * 1 = 2 * 10^2 + 3 * 10^1 + 7 * 10^0$$

where x^n denotes 1 if $n = 0$, or $x * x * \ldots\ x$ (n times) otherwise. In other words, an arabic number is a sequence of digits between 0 and 9, where the i^{th} digit from the right (counting from 0) represents a quantity of 10^i.

A binary number is similar but uses two digits instead of ten, namely 0 and 1, called *bits*. It is thus a sequence of bits, where the i^{th} bit from the right (counting from 0) represents a quantity of 2^i. For example

$$101_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 1 * 4 + 0 * 2 + 1 * 1 = 5$$

where the subscript 2 indicates a binary number (to avoid confusions with arabic numbers; $101_2 = 5 \neq 101$ = "one hundred one"). Another example is

$$\begin{aligned} 11101101_2 &= 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\ &= 1 * 128 + 1 * 64 + 1 * 32 + 0 * 16 + 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1 \\ &= 237 \end{aligned}$$

The leftmost bit of a binary number is called its *most significant* bit. Conversely, the rightmost bit is called the *least significant*. The i^{th} bit from the right (counting from 0), is called bit number i, or simply bit i.

n	2^n	2^n	$2^n - 1$	$2^n - 1$
0	1	1_2	0	0
1	2	10_2	1	1
2	4	100_2	3	11_2
3	8	1000_2	7	111_2
4	16	10000_2	15	1111_2
5	32	100000_2	31	11111_2
6	64	1000000_2	63	111111_2
7	128	10000000_2	127	1111111_2
8	256	100000000_2	255	11111111_2
16	65536	10000000000000000_2	65535	1111111111111111_2

TABLE 1.1 Some frequently used powers of 2, in arabic and binary notation.

Some numbers have a very simple binary representation and are frequently used. For instance, 2^n is a one followed by n zeros in binary, like 10^n in arabic notation. Another example is $2^n - 1$, which is simply n ones (like $10^n - 1$ is n nines in arabic). Table 1.1 gives some examples of these numbers.

1.2 Arithmetic operations

1.2.1 Addition

Adding two binary numbers can be done as with arabic numbers. Namely one column at a time, from right to left. For instance, adding 1101010_2 and 101110_2 can be done as follows:

$$\begin{array}{r} {\scriptstyle 1\,1\quad 1\,1\,1\quad} \\ 1\,1\,0\,1\,0\,1\,0 \\ +\quad 1\,0\,1\,1\,1\,0 \\ \hline 1\,0\,0\,1\,1\,0\,0\,0 \end{array} \qquad \begin{array}{r} {\scriptstyle 1\quad} \\ 1\,0\,6 \\ +\ 4\,6 \\ \hline 1\,5\,2 \end{array}$$

Starting from the right, we add 0 and 0, which gives 0. We then add 1 and 1, which gives $2 = 10_2$. Since this is more than one bit, we put the least significant one, here 0, in the current column, and we *carry* the most significant one, here 1, in the column on the left (shown in red). This is similar to the addition of the equivalent arabic numbers, shown on the right, where $6 + 6$ gives 12, leading to a carry of 1.

We continue by adding 0 and 1, plus the carry from the previous column, which gives 2 again. In the next step we add 1 and 1, plus the carry from the previous column, which gives $3 = 11_2$. We thus put 1 at the bottom of this column, and carry 1 in the next one. And so on for the remaining columns.

Although the overall process is the same for binary and arabic numbers, adding binary numbers is much easier, as stated above. Indeed, there are only $2 * 2 = 4$

result bit:

a	b	a+b
0	0	0
0	1	1
1	0	1
1	1	0

carry bit:

a	b	a+b
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 1.2 The binary addition tables.

result bit:

a	b	a-b
0	0	0
0	1	1
1	0	1
1	1	0

carry bit:

a	b	a-b
0	0	0
0	1	1
1	0	0
1	1	0

TABLE 1.3 The binary subtraction tables.

possible cases when adding two bits, but $10 * 10 = 100$ cases when adding two decimal digits. These four cases are summarized in Table 1.2.

1.2.2 Subtraction

Similarly, subtracting two binary numbers can be done as with arabic numbers. For instance, subtracting 101110_2 from 1101010_2 can be done as follows:

$$\begin{array}{r} 1\,1\,0\,1\,0\,1\,0 \\ -\;\;1\,0\,1\,1\,1\,0 \\ \hline 0\,1\,1\,1\,1\,0\,0 \end{array} \qquad \begin{array}{r} 1\,0\,6 \\ -\;\;4\,6 \\ \hline 0\,6\,0 \end{array}$$

Starting from the right, we subtract 0 from 0, and then 1 from 1, which gives 0 in both cases. In the next step, since we cannot subtract 1 from 0, we subtract it from $10_2 = 2$ instead, which gives 1. We thus put a 1 in the current column, and a carry of 1 in the subtrahend on the left column (shown in red). This is similar to the subtraction of the equivalent arabic numbers, shown on the right, where $0 - 4$ is replaced with $10 - 4$, yielding the result 6 and the carry 1.

We continue by subtracting 1, plus the carry from the previous column (*i.e.*, a total of 2), from 1. Since this is not possible we subtract them from $11_2 = 3$ instead, which gives the result 1 and the carry 1. And so on for the remaining columns.

As with additions, there are only four possible cases when subtracting two bits, which is much simpler than the hundred possible cases for decimal digits. These four cases are summarized in Table 1.3.

1.2.3 Multiplication

Multiplying two binary numbers can also be done as with arabic numbers. Namely by multiplying the first by each bit / digit of the second. And by adding the results, each shifted by one bit / digit to the left from the previous one. For instance, multiplying 1101010_2 by 101110_2 can be done as follows:

$$
\begin{array}{rr}
 & 1101010 \\
* & 101110 \\
\hline
 & 0000000 \\
 & 1101010 \\
 & 1101010 \\
 & 1101010 \\
 & 0000000 \\
 & 1101010 \\
\hline
 & 1001100001100
\end{array}
\qquad
\begin{array}{r}
106 \\
* \ 46 \\
\hline
636 \\
424 \\
\hline
4876
\end{array}
$$

Here again, although the process is the same, multiplying binary numbers is much easier than arabic numbers. Indeed, multiplying the first number by each bit of the second boils down to multiplications by 0 or 1, which are trivial. By contrast, multiplying an arabic number by a decimal digit requires using a multiplication table with 100 possible cases. It also involves carries.

Some multiplications are even easier to do than with the general method described above. In particular, multiplying x by 2^n can be done by simply shifting x by n bits to the left, *i.e.*, by adding n zeros on the right. For instance, $1101010_2 = 106$ multiplied by $2^3 = 8$ is simply $1101010\mathbf{000}_2 = 848$. This is similar to multiplications by 10^n in arabic notation (for example, 46 times $10^3 = 1000$ is $46\mathbf{000}$). Shifting a binary number x by n bits to the left is noted $x \ll n$.

The opposite operation, shifting x by n bits to the right, *i.e.*, dropping the n least significant bits, is noted $x \gg n$. It corresponds to dividing x by 2^n. For instance, shifting $1101010_2 = 106$ by 3 bits to the right gives $1101_2 = 13 = \lfloor 106/2^3 \rfloor$[1]. This is similar to dropping the n least significant digits of an arabic number, which divide it by 10^n (for example, 4876 shifted to the right by 2 digits is $48 = \lfloor 4876/100 \rfloor$). Dividing arbitrary binary numbers can be done as with arabic numbers, but is not presented here.

1.2.4 Conversions

Computers do all their computations with binary numbers because, as shown above, this is much easier to do than with arabic numbers. However, humans prefer to specify inputs with arabic numbers, and to get results in arabic too. This requires converting arabic numbers to binary ones, and vice versa.

[1] The $\lfloor x \rfloor$ notation designates the integer part of x. For instance, $106/8 = 13.25$ and $\lfloor 13.25 \rfloor = 13$.

One method to convert an arabic number to binary is to convert each digit from left to right, and to multiply the result by 10 before adding the next digit. For instance, to convert 46 to binary, we start by converting 4, which gives 100_2. We multiply this by $10 = 8 + 2$, which can be done by shifting 100_2 by 3 bits and by 1 bit to the left, and by adding the results: $100\mathbf{000}_2 + 100\mathbf{0}_2 = 101000_2$. Finally we convert 6, which gives 110_2 and we add this to the previous result, yielding 101110_2. This method is well suited for computers since it only involves computations on binary numbers (plus a small conversion table for each digit from 0 to 9).

Another method consists in dividing the arabic number by 2 repeatedly. The remainders give the bits of the equivalent binary number, from right to left. For instance, dividing 46 by 2 repeatedly gives 23 (remainder 0), 11 (remainder 1), 5 (remainder 1), 2 (remainder 1), 1 (remainder 0), and 0 (remainder 1). The corresponding binary number is thus 101110_2. Since this method involves divisions on arabic numbers, it is more adapted for humans than for computers.

Similarly, one method to convert a binary number to arabic is to "convert" each bit from left to right, and to multiply the result by 2 before adding the next bit. For instance, converting 101110_2 gives successively 1, $1 * 2 + 0 = 2$, $2 * 2 + 1 = 5$, $5 * 2 + 1 = 11$, $11 * 2 + 1 = 23$, and $23 * 2 + 0 = 46$. Since this method involves multiplications of arabic numbers, it is more adapted for humans. But it can also be used on computers, if necessary.

Another method to convert a binary number is to divide it by 10 repeatedly. The remainders, converted to arabic, give the digits of the equivalent arabic number, from right to left. It is well suited for computers since it only involves computations on binary numbers (plus a small conversion table for each binary number from 0 to $1001_2 = 9$).

1.3 Logical operations

Binary numbers can also be used to perform *logical operations*, unlike arabic numbers. A logical operation computes whether some *proposition* is true of false, depending on the status of one or more other propositions. A proposition is a statement which is either true or false.

Consider for example a keyboard. A proposition might be "the E key is currently pressed", "the left Shift key is currently released", or "the e letter is currently pressed". They are either true or false, depending on the current state of the keyboard. These propositions, noted $\mathrm{KeyPressed}(k)$, $\mathrm{KeyReleased}(k)$, and $\mathrm{LetterPressed}(l)$, are not completely independent. Some can be computed from the others. For example, we can compute $\mathrm{KeyReleased}(k)$ as the opposite of $\mathrm{KeyPressed}(k)$. This logical operation is the *negation*, also called *not*, and is noted $\neg$:

$$\mathrm{KeyReleased}(k) = \neg\mathrm{KeyPressed}(k)$$

We can also compute whether the proposition "a Shift key is pressed" is true from the above propositions. Indeed, this is the case if at least one of the two Shift keys is

p	$\neg p$
0	1
1	0

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

p	q	$p \oplus q$
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 1.4 The truth tables of not ($\neg$), and ($\wedge$), or ($\vee$), and exclusive or ($\oplus$).

pressed. This logical operation is the *disjunction*, also called *or*, and is noted $\vee$:

$$\mathrm{ShiftPressed} = \mathrm{KeyPressed}(\mathrm{LeftShift}) \vee \mathrm{KeyPressed}(\mathrm{RightShift})$$

The keyboard is in "uppercase mode" if a Shift key is currently pressed, or if caps are locked, but not both (a Shift key reverses the effect of CapsLock). This logical operation is the *exclusive disjunction*, also called *exclusive or*, and is noted $\oplus$:

$$\mathrm{UppercaseMode} = \mathrm{ShiftPressed} \oplus \mathrm{CapsLocked}$$

As a last example, we can also compute whether $\mathrm{LetterPressed}(\mathrm{E})$ is true from $\mathrm{KeyPressed}(\mathrm{E})$ and $\mathrm{UppercaseMode}$. Indeed, this is the case if both are true. This logical operation is the *conjunction*, also called *and*, and is noted $\wedge$:

$$\begin{aligned}\mathrm{LetterPressed}(\mathrm{E}) &= \mathrm{KeyPressed}(\mathrm{E}) \wedge \mathrm{UppercaseMode}\\ \mathrm{LetterPressed}(\mathrm{e}) &= \mathrm{KeyPressed}(\mathrm{E}) \wedge \neg\mathrm{UppercaseMode}\end{aligned}$$

The above logical operations do not depend on the meaning of the propositions, but only on whether they are true or false. And their result is either true or false. For instance, $\neg$ true is false, $p \wedge q$ is true if and only if both p and q are true, $p \vee q$ is true if at least one of p and q is true, etc. By representing true with 1 and false with 0, they can be seen as operations on individual bits. This gives, for example, $\neg 1 = 0$, $1 \wedge 0 = 0$, $1 \wedge 1 = 1$, etc. By doing this for all possible cases we get the *truth table* of each operation, represented in Table 1.4. Note that the truth tables of $\oplus$ and $\wedge$ are identical to those giving the result and carry bit of $a + b$, respectively (see Table 1.2). The result bit of $a - b$ is also equal to $a \oplus b$, and the carry bit is $b \wedge \neg a$ (see Table 1.3). Hence, it suffice to know how to implement these logical operations with electric circuits, or other technologies, in order to be able to implement arithmetic circuits.

We can then generalize these logical operations from individual bits to whole binary numbers. By definition, a *bitwise* logical operation on two binary numbers is done by applying it on each bit separately, column by column. Thus, for instance:

$$\begin{array}{r} 1100 \\ \wedge\, 1010 \\ \hline 1000 \end{array} \qquad \begin{array}{r} 1100 \\ \vee\, 1010 \\ \hline 1110 \end{array} \qquad \begin{array}{r} 1100 \\ \oplus\, 1010 \\ \hline 0110 \end{array}$$

This can be used to perform several logical operations in parallel (since there is no carry each column can be computed independently of the others, possibly at the same time). For instance, we can represent the current state of a 100 keys keyboard with a 100 bits binary number S, using one bit per key. We can then do the following operations, which are commonly used in many similar contexts:

- to check whether at least one letter key is pressed, we can compute $S \wedge L$, where L is the binary number whose i^{th} bit is 1 if and only if the i^{th} key is a letter. If the result is 0 no letter key is pressed, otherwise at least one is pressed.
- if a new set of keys if pressed, we can compute the representation of the new keyboard state with $S' = S \vee P$, where P represents the newly pressed keys. For instance, if the 0^{th} and 3^{rd} keys are currently pressed, and if the user presses the 0^{th} and 2^{nd} keys[2], we get $S = 1001_2$, $P = 101_2$ and $S' = 1101_2$. This correctly represents the fact that the 0^{th}, 2^{nd}, and 3^{rd} keys are now pressed.
- if a new set of keys if released, we can compute the representation of the new keyboard state with $S' = S \wedge \neg R$, where R represents the newly released keys. For instance, if the 0^{th} and 3^{rd} keys are currently pressed, and if the user releases the third, we get $S = 1001_2$, $R = 1000_2$ and $S' = 1$. This correctly represents the fact that only the 0^{th} key remains pressed.

1.4 Hexadecimal numbers

Binary numbers are very practical to perform computations, but are not very compact. Arabic numbers are much more compact (a given number has about 3.3 less digits than bits on average), but converting between binary and arabic is not so easy. To solve these issues *hexadecimal* numbers are commonly used.

Hexadecimal numbers are like arabic numbers, but use 16 digits instead of 10. They are called hex digits and are noted 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (= 10), B (= 11), C (= 12), D (= 13), E (= 14), and F (= 15). An hexadecimal number is thus a sequence of hex digits, where the i^{th} hex digit from the right (counting from 0) represents a quantity of 16^i. For instance

$$ED_{16} = E_{16} * 16^1 + D_{16} * 16^0 = 14 * 16 + 13 = 237$$

where the subscript 16 indicates an hexadecimal number (to avoid confusions with words or arabic numbers; $10_{16} = 16 \neq 10$ = "ten").

Each hex digit can be represented with up to 4 bits, and each group of 4 bits can be represented with an hex digit, as shown in Table 1.5. It is thus very easy to convert a binary number to hexadecimal: simply convert each group of 4 bits independently, with Table 1.5. For instance, to convert 11101101_2, we convert 1110_2 (E_{16}), 1101_2 (D_{16}), and concatenate the results, yielding ED_{16}. Conversely, to convert ED_{16} to binary we simply concatenate the conversions of E_{16} (1110_2) and D_{16} (1101_2), yielding 11101101_2.

[2] A pressed key can be "pressed" again due to autorepeat.

binary	hex	binary	hex	binary	hex	binary	hex
0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F

TABLE 1.5 Conversion between binary and hexadecimal.

Hexadecimal numbers are thus compact (a given number has about 4 times less hex digits and bits) and easy to convert to and from binary, which solves the above issues. On the other hand, doing arithmetic computations with them is harder than with arabic numbers (this involves tables with $16 * 16 = 256$ entries). But this is not necessary since we can convert them to binary, do computations in binary, and convert the result back to hexadecimal.

CHAPTER

2 Logic Gates and Arithmetic Circuits

As explained in the previous chapter, computing arithmetic operations on binary numbers boils down to the computation of simple logical operations such as conjunctions and exclusive disjunctions. This chapter explains how these operations can be implemented with electric circuits, and then how these circuits can be combined to perform arithmetic operations.

2.1 Transistors

In order to implement a logical operation with an electric circuit we first need a way to represent 0 and 1 with some electric states. One possibility is to view a wire connected to the ground as 0, and a wire connected to the power source (*e.g.*, +5V) as 1. To implement a circuit for $\neg p$, for instance, we can use an input wire for p, and an output wire for the result $\neg p$. The circuit in the middle must then connect the output wire to the ground (resp. power source) if the input wire is connected to the power source (resp. ground). A simple switch can do this, provided it is controlled by the input wire, instead of manually. In fact, as illustrated in the next sections, such switches are sufficient to implement any logical operation.

An electric switch itself controlled by electricity connects or disconnects two terminals, hereafter noted a and b, depending on the voltage or current in a third one, noted c. One method to do this is to use a *transistor*. Another method is to use a *relay*. Transistors are much more efficient than relays, and are used virtually everywhere. But relays are simpler to understand and, for this reason, we use them in this chapter to explain how logic gates work.

A relay can be built with an electromagnet controlling a metallic connector. There are two types of relays connecting or disconnecting two terminals (see Figure 2.1):

- in a *normally open* relay, the a and b terminals are disconnected when no current is flowing through the electromagnet. They are connected when the relay is *active*, *i.e.*, when there is a current in the electromagnet.
- in a *normally closed* relay, the a and b terminals are connected when the relay is *inactive*, *i.e.*, when there is no current in the electromagnet. They are disconnected when it is active.

In the following we represent relays with the symbols illustrated in Figure 2.2. We

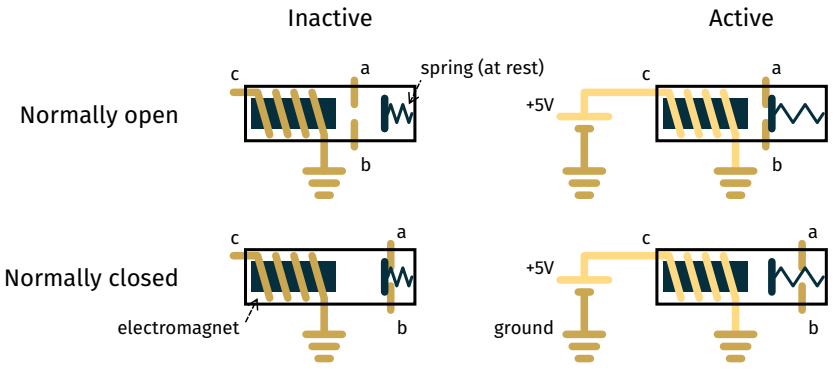


FIGURE 2.1 The two types of relays used in this chapter. The electromagnet, when active (right), attracts a metallic piece. This connects the a and b terminals of a normally open relay (top), and disconnects those of a normally closed one (bottom). When the electromagnet inactive, a spring moves the metallic piece away from it.

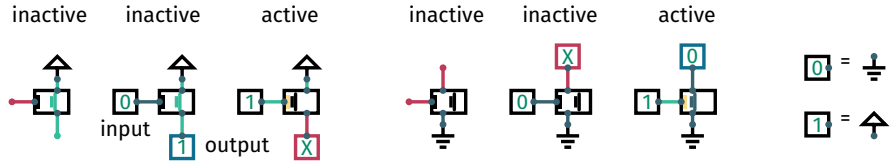


FIGURE 2.2 The symbols and colors used for normally closed (left) and normally open (right) relays, as well as for wires and input (black) and output (blue) terminals connected to the ground, to the power source (up triangle), or to nothing (in red).

draw input terminals connected to the ground (resp. power source) with a 0 (resp. 1) inside a black square. Similarly, we use a 0 (resp. 1) inside a blue square for output terminals connected to the ground (resp. power source). We represent those which are not connected to anything with an X inside a red square. Finally, we draw wires connected to the ground, to the power source, or to nothing in blue, green, and red, respectively (see Figure 2.2).

2.2 Logic gates

2.2.1 NOT

A *NOT gate* is a circuit implementing the logical not operation. This gate can be built with two relays controlled by the same input[1]. The first, normally closed, connects

[1] In practice, with electromagnet relays, 0 can be represented with a terminal connected to the ground or to nothing. Then a single normally closed relay is sufficient to build a NOT gate [5]. In this chapter we do as if it was not the case. This leads to circuits which are much closer to those built with the most common technology, namely Complementary Metal Oxide Semi-conductors (CMOS).

FIGURE 2.3 The two possible states of the NOT gate.

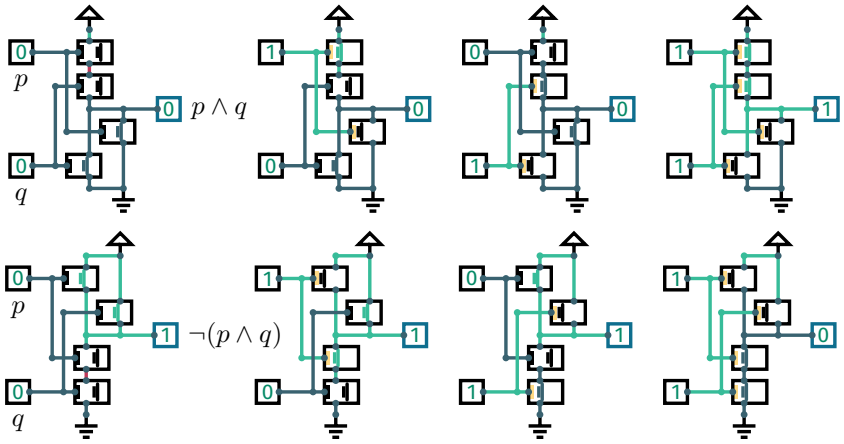


FIGURE 2.4 The four possible states of the AND (top) and NAND (bottom) gates.

the output to the power source by default. The second, normally open, connects the output to the ground when active (see Figure 2.3). Hence, when the input is 0 the first relay connects the output to the power source, *i.e.*, sets it to 1 (while the second does nothing). Conversely, when the input is 1, the first relay has no effect but the second connects the output to the ground, *i.e.*, sets it to 0 (see Figure 2.3).

2.2.2 AND and NAND

The *AND gate* is a circuit implementing the logical and operation. This circuit must connect the output to the power source when both inputs are 1. This can be done with two normally open relays connected in series. Conversely, this gate must connect the output to the ground when at least one input is 0. This can be done with two normally closed relays connected in parallel (see Figure 2.4).

The *NAND gate* implements the negation of the logical and, *i.e.*, it computes $\neg(p \wedge q)$. It can be obtained by connecting a NOT gate to the output of an AND gate. But a simpler method is to switch the power source and the ground of the AND gate or, equivalently, the upper and lower halves of this circuit[2] (see Figure 2.4).

[2]With the CMOS technology "normally closed" (resp. "open") transistors are only used in the upper (resp. lower) half of a gate. Hence a CMOS AND gate is built with a NAND gate followed by a NOT.

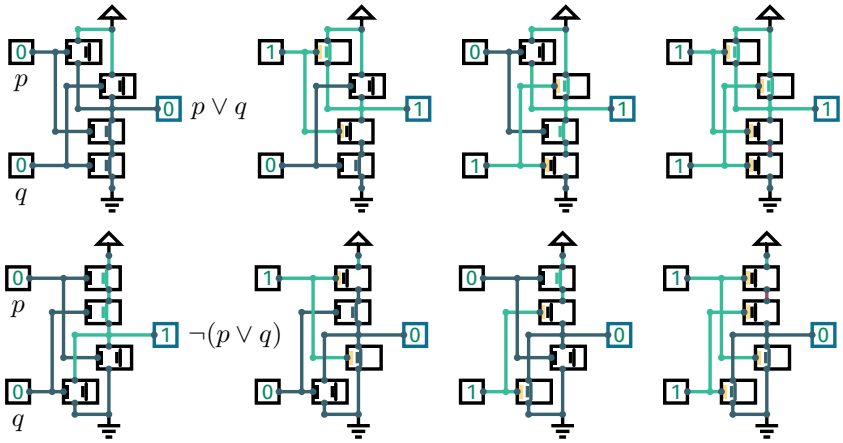


FIGURE 2.5 The four possible states of the OR (top) and NOR (bottom) gates.

2.2.3 OR and NOR

The *OR gate* is a circuit implementing the logical or operation. This gate must connect the output to the power source when at least one input is 1. This can be done with two normally open relays connected in parallel. Conversely, this gate must connect the output to the ground when both inputs are 0. This can be done with two normally closed relays connected in series (see Figure 2.5).

The *NOR gate* implements the negation of the logical or, *i.e.*, it computes $\neg(p \vee q)$. As the NAND gate, it can be obtained by switching the upper and lower halves of the OR gate circuit (see Figure 2.5).

2.2.4 XOR

The *XOR gate* implements the exclusive or operation. The result of $p \oplus q$ is 1 when p is 1 and q is 0, or when p is 0 and q is 1. This gate must thus connect the output to the power source if at least one these two cases happens. This can be done with two sub circuits, one for each case, connected in parallel. Each sub circuit must connect its output to the power source when both inputs have a specific value. This can be done with two relays connected in series: a normally open for p or q, and a normally closed for $\neg p$ or $\neg q$.

Conversely, the result of $p \oplus q$ is 0 when both "p is 0 or q is 1" and "p is 1 or q is 0" are true. The same reasoning as above leads to two sub circuits connected in series, where each sub circuit uses two relays connected in parallel. This lead to the final circuit shown in Figure 2.6.

In the following, to simplify figures and to make it easier to distinguish each logic gate, we represent them with their American National Standards Institute (ANSI) symbols, shown in Figure 2.7.

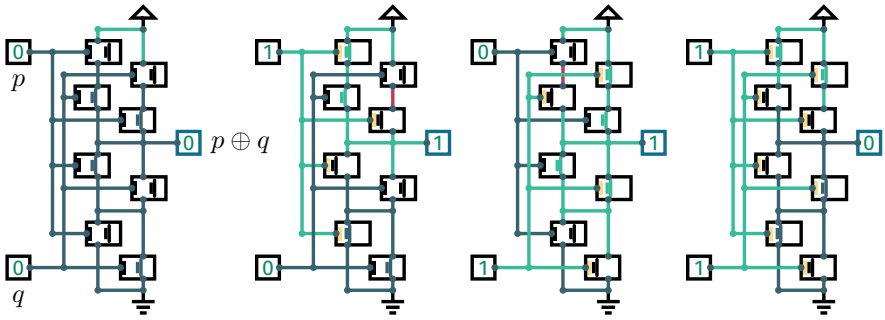


FIGURE 2.6 The four possible states of the XOR gate.

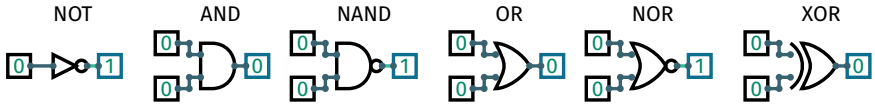
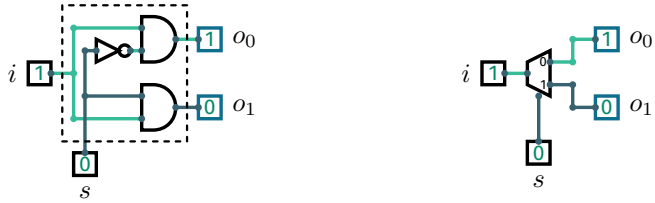


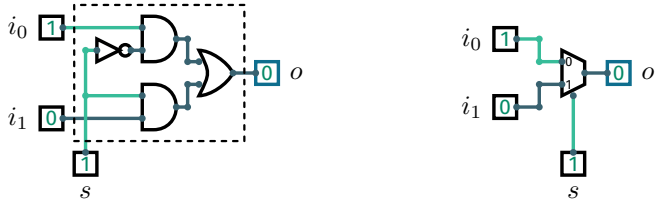
FIGURE 2.7 The ANSI symbols of the NOT, AND, NAND, OR, NOR and XOR logic gates.

2.3 Multiplexers and demultiplexers

Logic gates can be assembled to create more and more complex circuits. A simple example is the *demultiplexer*, shown below, and represented with the symbol on the right:



This circuit copies its input i to the o_s output, *i.e.*, to o_0 if $s = 0$ or to o_1 if $s = 1$. It sets the other to 0. It can be viewed as a "railroad switch" for signals. The *multiplexer*, shown below and represented with the symbol on the right, does the opposite:



This circuit sets its output o to the i_s input, *i.e.*, to i_0 if $s = 0$, or to i_1 if $s = 1$.

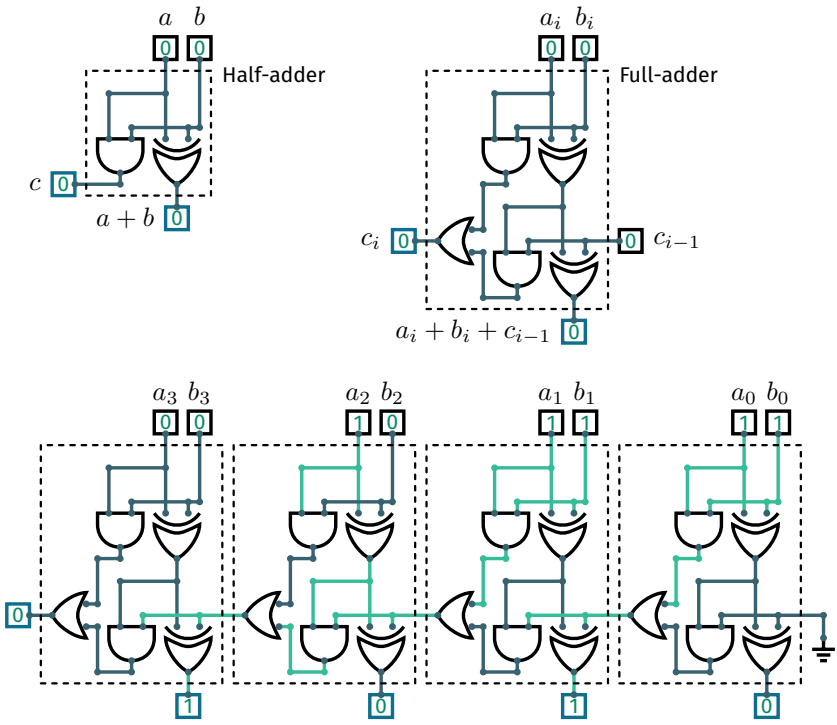


FIGURE 2.8 A circuit to add two 4-bit numbers (bottom) can be built with 4 full-adder circuits (top right), each made of two half-adders (top left) and an OR gate. Here this circuit computes $0111_2 + 0011_2 = 1010_2$ ($7 + 3 = 10$).

2.4 Arithmetic circuits

2.4.1 Adder

As shown in the previous chapter, the addition of two bits is simply their exclusive disjunction, with a carry equal to their conjunction. In other words, we can add two bits with an XOR gate, plus an AND gate for the carry. The resulting circuit, called a *half-adder*, is illustrated in Figure 2.8.

As explained in Section 1.2.1, adding two binary numbers a and b must be done step by step, from right to left. At each step, one bit a_i from a must be added to one bit b_i from b, and to the carry c_{i-1} from the previous step. In other words, three bits must be added at each step, but the above circuit can only add two. The solution is to connect two copies of it: a first copy adds a_i and b_i, and a second adds c_{i-1} to the result of the first. Each copy produces a new carry, but at most one of these can be 1. Indeed, if $a_i + b_i$ gives a carry then the second stage necessarily adds c_{i-1} to 0, which cannot give a carry. Hence the new carry c_i resulting from $a_i + b_i + c_{i-1}$ can

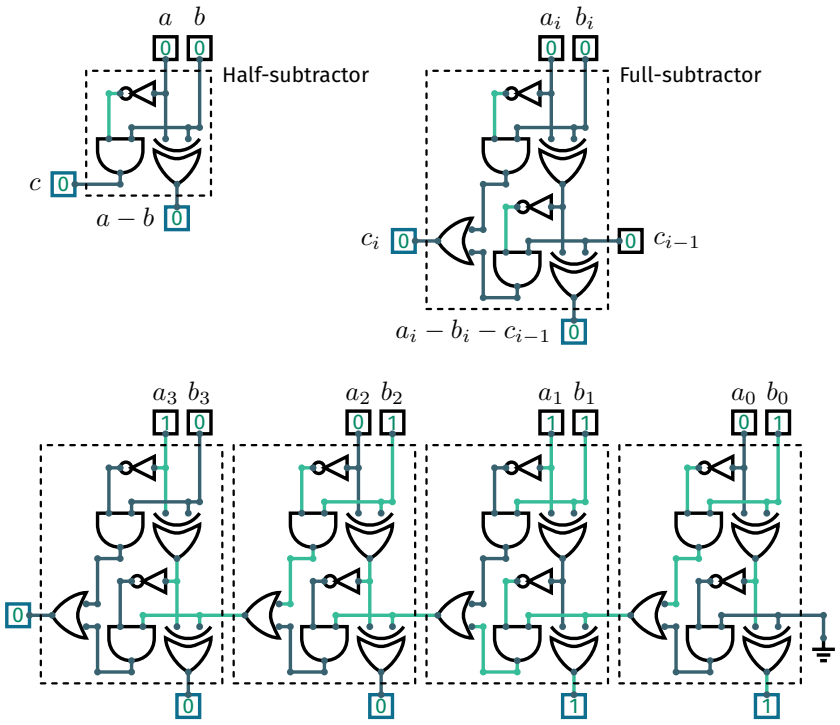


FIGURE 2.9 A circuit to subtract two 4-bit numbers (bottom) can be built with 4 full-subtractor circuits (top right), each made of two half-subtractors (top left) and an OR gate. Here this circuit computes $1010_2 - 0111_2 = 0011_2$ ($10 - 7 = 3$).

be computed with a disjunction of the carries from the two half-adders. This leads to the *full adder* circuit shown in Figure 2.8.

Finally, to add two binary numbers with n bits each, we simply need to connect n full-adder circuits, with the output carry c_i of step i connected to the input carry c_{i-1} of step $i + 1$ (see Figure 2.8).

2.4.2 Subtractor

Subtracting two binary numbers can be done with a very similar circuit. As shown in the previous chapter, subtracting a bit b from a gives their exclusive disjunction (as their addition), plus a carry equal to the conjunction of $\neg a$ and b (versus of a and b for an addition). In other words, a circuit to subtract b from a can be obtained by adding a NOT gate in a half-adder circuit. The result, called a *half-subtractor*, is illustrated in Figure 2.9.

Subtracting two binary numbers a and b must be done step by step, from right to left. At each step, one bit b_i from b, and the carry c_{i-1} from the previous step, must

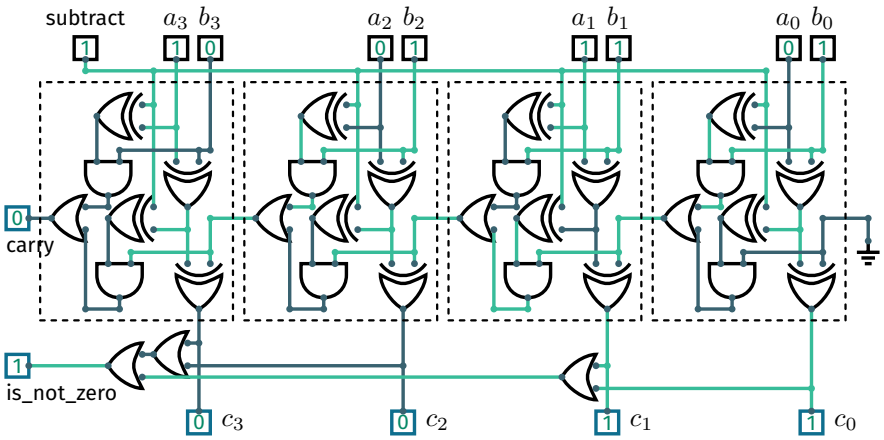


FIGURE 2.10 A simple Arithmetic Unit which can perform additions, subtractions, and comparisons of 4-bit numbers.

be subtracted from a bit a_i from a. In other words, three bits must be subtracted at each step, but the above circuit can only subtract two. The solution is to connect two copies of it: a first copy subtracts b_i from a_i, and a second subtracts c_{i-1} from the result of the first. Each copy produces a new carry, but at most one of these can be 1. Indeed, if $a_i - b_i$ gives a carry then the second stage necessarily subtracts c_{i-1} from 1, which cannot give a carry. Hence the new carry c_i resulting from $a_i - b_i - c_{i-1}$ can be computed with a disjunction of the carries from the two half-subtractors. This leads to the *full subtractor* circuit shown in Figure 2.9.

Finally, to subtract two binary numbers with n bits each, we simply need to connect n full subtractor circuits, with the output carry c_i of step i connected to the input carry c_{i-1} of step $i+1$ (see Figure 2.9).

2.4.3 Arithmetic and Logic Unit

As shown in Section 1.2.1, multiplying two binary numbers a and b boils down to additions of left shifted copies of a, each multiplied by a bit of b. Furthermore, $a_i * b_j$ gives the same result as $a_i \wedge b_j$. Hence a circuit to multiply two n-bit binary numbers (yielding $2n$ bits) can be obtained with n copies of an n-bit adder, plus n^2 AND gates to compute the $a_i \wedge b_j$ terms.

Comparing two n-bit binary numbers is also easy to do. Indeed:

- $a = b$ if and only if the n least significant bits of $a - b$ are equal to 0.
- $a > b$ if and only if at least one of the n least significant bits of $a - b$ is different from 0, and if there is no carry in the n^{th} column (counting from 0).
- $a < b$ if and only if at least one of the n least significant bits of $a - b$ is different from 0, and if there *is* a carry in the n^{th} column (counting from 0).

Hence a subtractor circuit, plus a another computing whether its output (excluding the carry) is different from 0, is sufficient to compare two numbers.

Finally, circuits computing bitwise logical operations on n-bit numbers are trivial to implement. Indeed, we just need n copies of the corresponding logic gate, each computing one bit of the result, independently of the others (*i.e.*, in parallel).

All these circuits can be put together into a larger circuit called an *Arithmetic and Logic Unit*. Such a circuit accepts two binary numbers as input, plus a third one specifying an operation to perform on them. It produces as output the result of this operation, on the given numbers.

For instance, a very simple Arithmetic "and Logic" Unit which can only perform additions, subtractions, and comparisons is shown in Figure 2.10. If its `subtract` input is 1 it subtracts its two 4-bit inputs. Otherwise it adds them. For this it uses a subtractor circuit where the NOT gates are replaced with XOR gates, connected to the `subtract` input. When this input is 0, the XOR gates behave as a simple wire ($p \oplus 0 = p$), which gives an adder circuit. When `subtract` is 1 these gates behave as NOT gates ($p \oplus 1 = \neg p$), yielding a subtractor. Finally, three OR gates compute whether at least one bit of the output is 1. Together with the carry bit, this can be used to compare the inputs, as explained above.

To conclude this chapter, it should be noted that a relay takes some time to switch between its active and inactive states (because its moving metallic piece cannot move instantly). This is the case for transistors too. Consequently, the output of a logic gate does not change instantly when its inputs change. And this is the same for all circuits. The more logic gates there is between an input and an output, the longer it takes for an input change to *propagate* to the output. These propagation delays must be taken into account in some circuits, including some presented in the next chapters.

CHAPTER

3 Register and Memory Circuits

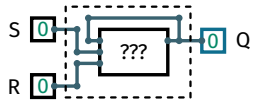
The very basic Arithmetic Unit presented in the previous chapter can perform several operations, but only one at a time. To do a sequence of operations one has to memorize intermediate results, or to note them somewhere. For instance, to add a, b, and c, one has to set the inputs to a and b, memorize the output $a + b$, and replace the inputs with $a + b$ and c (changing the inputs immediately changes the output, hence one cannot directly "copy" it to an input). To avoid this mental or manual work, a solution is to use additional circuits to memorize intermediate results. This chapter explains how this can be done with logic gates.

3.1 Memory cells

3.1.1 SR latch

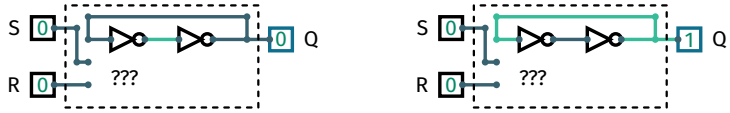
A circuit which can memorize a single bit must have some inputs to set the value to memorize, and an output equal to the last memorized value, noted Q. One possibility is to use one input to *set* the memorized value to 1, noted S, and another to *reset* it to 0, noted R. Connecting S to the power source should change Q to 1, but connecting it to the ground should *not* change Q to 0 (otherwise this circuit would have no "memory"). Likewise, setting R to 1 should reset Q to 0, but setting it to 0 should not change Q. In particular, when both S and R are 0, Q should keep its memorized value, which can be 0 or 1.

The above requirements lead to a circuit whose output is not completely determined by its current inputs, unlike all the circuits presented so far (since Q can be 0 or 1 when $S = R = 0$). To achieve this a solution is to use a "hidden" input equal to the last value of Q, noted Q_{last}. Then Q can be defined as a function of its inputs again ($Q = Q_{\text{last}}$ if $S = R = 0$). By definition Q_{last} is the last output of the circuit, which leads to a loop:

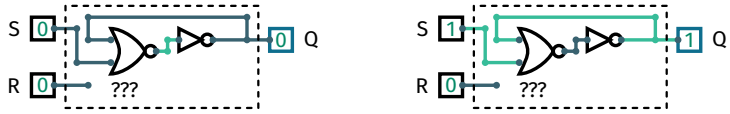


When $S = R = 0$, Q should be equal to the Q_{last} input. For this the circuit in the above box cannot simply connect Q to Q_{last}, since an electric current cannot flow in a

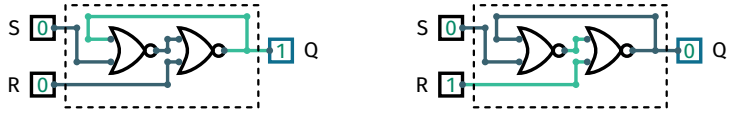
loop. A solution is to use two NOT gates in series instead. Indeed, this leads to a loop which has two stable states, $Q = 0$ and $Q = 1$:



To set Q to 1 we need to force the output of the right NOT gate to 1 or, equivalently, to force the output of the left NOT gate to 0. The latter can be done by replacing the left NOT gate with a NOR gate connected to S:



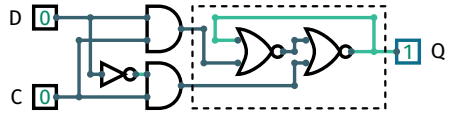
Indeed, this NOR gate behaves like a NOT gate when $S = 0$ (since $\neg(x \vee 0) = \neg x$), but forces its output to 0 when $S = 1$ (because $\neg(x \vee 1) = 0$). Similarly, we can set Q to 0 when $R = 1$ by replacing the remaining NOT gate with a NOR gate connected to R. This yields the following circuit, called an *SR latch*:



If $S = R = 1$ the two NOR gates force their output to 0. Switching from this state to $S = R = 0$ make them behave as NOT gates again, but starting with their input and output equal to 0. This state is unstable: depending on which gate updates its output to 1 first, the end result can be $Q = 0$ or $Q = 1$ (or, in theory, an infinite oscillation between 0 and 1). For this reason S and R must not be set to 1 at the same time. Note that the above unstable state also occurs when the circuit is powered on. In the following we assume that the R input of each SR latch is briefly set to 1 when circuits are powered on, so that their initial state is always 0.

3.1.2 D latch

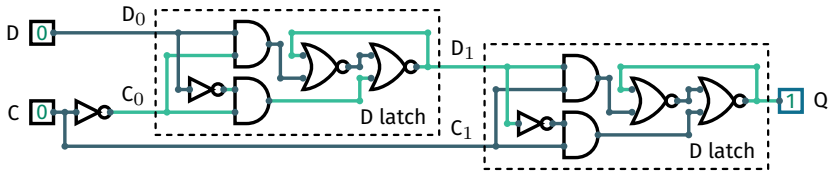
Using set and reset inputs is only one possibility to change the memorized value Q. Another is to set Q to the current value of some "data" input, noted D, when a "copy" input, noted C, is 1 (and to keep it unchanged when $C = 0$). In other words, Q should be set to 1 when $D = 1$ and $C = 1$, should be reset to 0 when $D = 0$ and $C = 1$, and should keep its value when $C = 0$. This is easy to do with an SR latch and 3 more gates to convert D and C to appropriate values of S and R:



This circuit is called a *D latch*. One advantage, compared to the SR latch, is that it does not have forbidden inputs such as $S = R = 1$. Indeed, thanks to the gates in front of the SR latch, this case can never happen.

3.1.3 D flip-flop

As long as $C = 1$, the output Q of a D latch changes each time D changes. This is not practical to memorize the value of D at a precise moment, unless C remains equal to 1 only for a very brief time (but long enough to allow the SR latch to stabilize to a potentially new state). To solve this issue, a possibility is to make sure that D does not change while $C = 1$. This can be done with two D latches in series, using opposite values for C, as shown below:



Indeed:

- when $C = 1$ the output D_1 of the first D latch does not change when D changes, because $C_0 = 0$. Hence the output of the second D latch does not change either, although its C_1 input is 1. In other words, the first D latch makes sure that D_1 does not change while $C_1 = 1$, as required.
- when $C = 0$ the output D_1 of the first D latch changes each time D changes, because $C_0 = 1$. But then $C_1 = 0$ and thus the output Q of the second D latch does not change.
- when C changes from 1 to 0, the first D latch memorizes the value of D and its output D_1 changes to this potentially new value. But this takes some time, whereas C_1 changes immediately when C changes. Hence, when D_1 changes, C_1 is already 0, and thus Q does not change.
- when C changes from 0 to 1, the first D latch keeps its state, *i.e.*, D_1 remains equal to the current value of D (this was the case since C was last set to 0). But C_1 also changes from 0 to 1. The second D latch thus memorizes D, and Q changes to D.

In summary, this circuit[1], called a *D flip-flop*, memorizes the value of D at the precise moment when C changes from 0 to 1, and keeps its state otherwise. In the following, to simplify figures and to make it easier to distinguish memory cells, we represent SR latches and D flip-flops as shown in Figure 3.1.

[1] Other circuits can achieve the same effect, with less gates and transistors (especially with relays [5]).

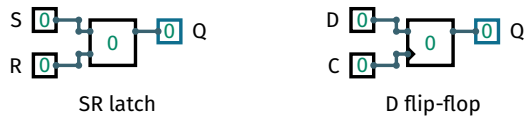
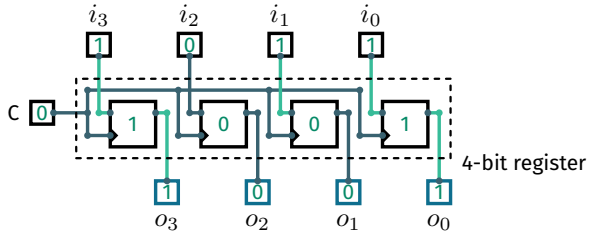


FIGURE 3.1 SR latches and D flip-flops are represented with their currently memorized value inside a large black square. The D flip-flop symbol differs from the SR latch symbol with a small black triangle on its C input.

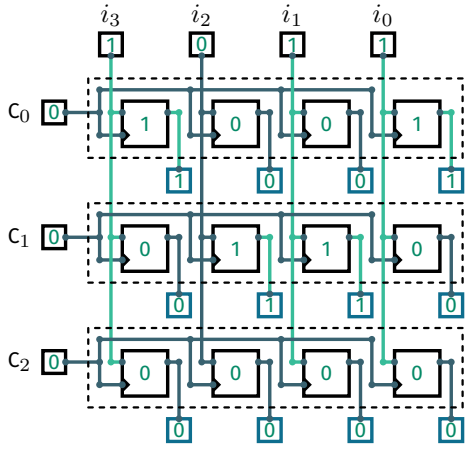
3.2 Memory circuits

To memorize an intermediate result of the Arithmetic Unit we need to memorize n bits simultaneously. This is easy to do with n flip-flops connected to the same C input:



This circuit is called a *register*. It memorizes its input when C changes from 0 to 1. Its output is the last memorized value.

To memorize several intermediate results we can connect the n outputs of the Arithmetic Unit to several n bit registers. We can then choose in which register to store this output by activating the C input of only one of these registers. For instance, the following circuit can store a 4-bit number in one of 3 registers:



However, getting the value from one these registers is not very easy because this circuit has too many output wires. To make it easier to use we can add one more input

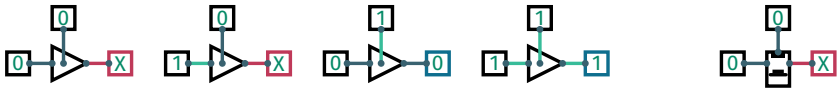
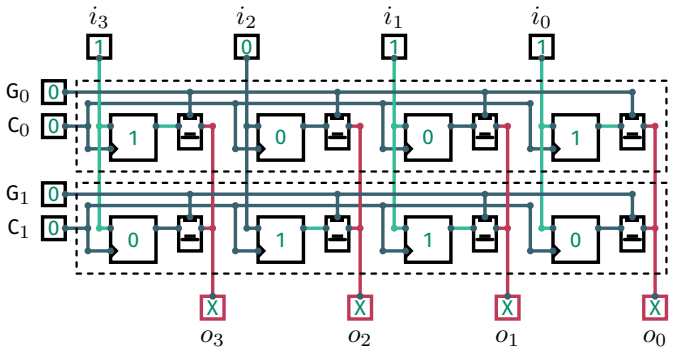


FIGURE 3.2 The four possible input combinations of a tristate buffer and the corresponding outputs (left). A tristate buffer is like a normally open relay (right).

per register, to optionally connect its outputs to n shared output wires. For this we can connect the i^{th} flip flop of each register to a shared output wire o_i via a normally open relay:



Together with additional inputs G_j, connected to the control inputs of all the relays of the j^{th} register, this allows connecting or disconnecting a whole register to the shared output wires. Normally open relays used in this way are called *tristate buffers* and are represented as shown in Figure 3.2. Their name comes from the fact that their output can have 3 states: 0, 1, or "disconnected".

In the above circuit G_0 and G_1 must not be simultaneously set to 1. Indeed, doing so could connect together the outputs of two flip flops with different states, resulting in a short circuit. More generally, with more than 2 registers, at most one G_j input must be set to 1 at a time. This can be done with a *binary decoder*. A binary decoder with k inputs $a_0, a_1, \ldots a_{k-1}$ has 2^k outputs $o_0, o_1, \ldots o_{2^k-1}$. It sets its output o_j to its i input, and all the others to 0, where j is the binary number $a_{k-1} \ldots a_1 a_0$. This circuit can be implemented with several demultiplexers, as illustrated in Figure 3.3.

We can connect it to 2^k registers as shown in Figure 3.4. This new circuit connects the outputs of the binary decoder to the G_j inputs of the registers and, via AND gates connected to a new w input, to their C_j inputs. This forms a *Random Access Memory* (RAM), called this way because it allows *reading* and *writing* (*i.e.*, to get and set) values in any order. For instance, with the circuit in Figure 3.4:

- reading the value of the j^{th} register can be done by setting the $a_2 a_1 a_0$ inputs to the bits of j in binary. The value is then obtained on the $o_3 o_2 o_1 o_0$ outputs. $j = a_2 a_1 a_0$ is called the *address* of this register.

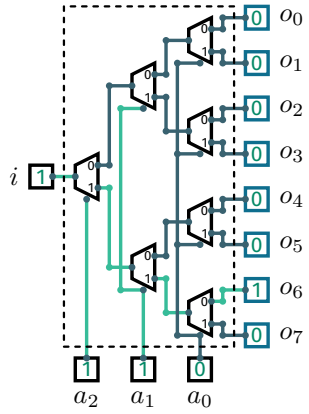


FIGURE 3.3 A binary decoder with 3 inputs. Here $a_2a_1a_0 = 110_2 = 6$, hence $o_6 = i$.

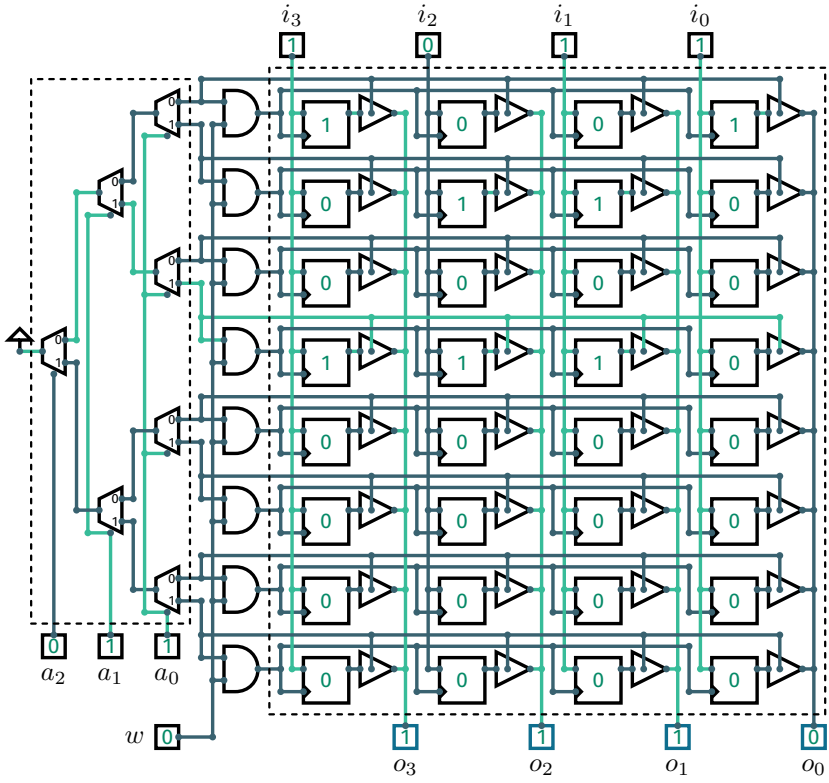


FIGURE 3.4 A Random Access Memory (RAM) storing eight 4-bit values. This circuit currently reads the value at address $a_2a_1a_0 = 011_2 = 3$, namely 1110_2. Setting w to 1 would write the input value $i_3i_2i_1i_0 = 1011_2$ at address 3.

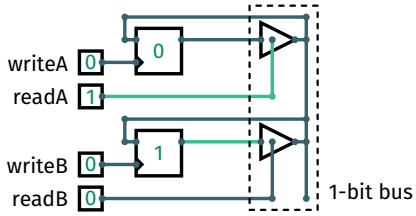
- writing a value v in the j^{th} register can be done by setting $i_3i_2i_1i_0$ to the bits of v in binary, by setting the $a_2a_1a_0$ inputs to the bits of j in binary, and finally by changing w from 0 to 1. The last step changes the C inputs of all the flip flops of the j^{th} register, making it memorize the shared inputs $i_3i_2i_1i_0$.

This basic circuit uses one address per group of 4 bits. In practice, most computers use one address per group of 8 bits, called a *byte*. They also use much more than 3 bits per address. A 10-bit address can refer to $2^{10} = 1024$ bytes, called a *kilobyte* (KB). A 20-bit address can refer to 1024 kilobytes, called a *megabyte* (MB). And a 30-bit address can refer to one *gigabyte* (GB – 1024 megabytes).

3.3 Bus

The above RAM circuit can store several intermediate results, but it has only one input address and one output value. Hence it is not sufficient, for instance, to directly add or subtract two intermediate results with the Arithmetic Unit. To solve this we can use two separate registers as input of the Arithmetic Unit, provided we have a way to copy values from the RAM to one or the other of these registers.

A circuit which can copy values from one register to another, or from a register to RAM or vice-versa, is called a *bus*. A 1-bit bus connecting n flip-flops is easy to build. We just need to connect the D input of each flip flop to a common wire, and to connect their Q output to this same wire via a tristate buffer, as in the RAM circuit:



The above circuit can copy the value from A to B by setting readA to 1 and then by changing writeB from 0 to 1. The first step connects A's output to the bus and thus to the D input of B. The second step memorizes this value in B. Conversely, this circuit can copy the value from B to A by setting readB to 1 and then by changing writeA from 0 to 1. It easy to generalize to 3 or more flip-flops. It can also be generalized to an n-bit bus, to copy values between 2 or more n-bit registers, or the RAM. For instance, the circuit in Figure 3.5 can copy values between two 3-bit registers. It is made of three copies of the 1-bit bus, with shared "read" and "write" inputs. Copying 3-bit values from A to B or vice versa can be done as with the 1-bit bus.

To maintain a register connected in "read mode" to the bus we can memorize the "read" inputs in SR latches. And, to make it easier to read another register, we can connect the S input of each latch to the R input of all the others (so that setting one resets the others – as in the RAM, at most one register must be connected to the bus at a time). For instance, the circuit in Figure 3.6 sets readA to 1, readB to 0, and

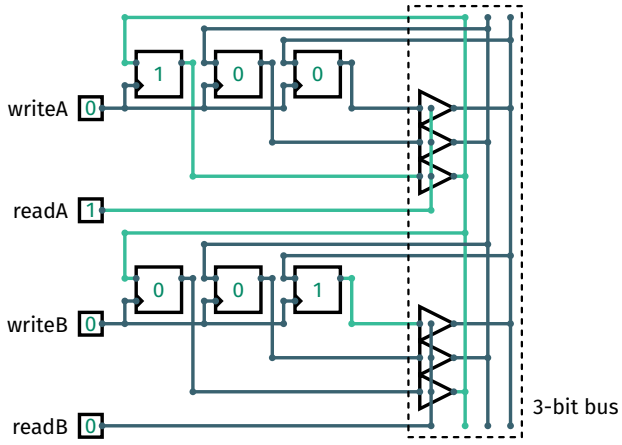


FIGURE 3.5 A 3-bit bus (right) connected to two 3-bit registers (left).

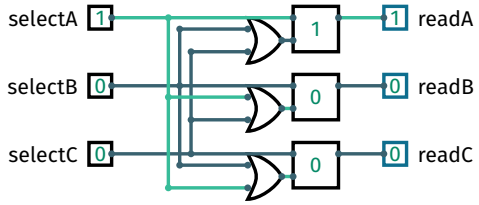


FIGURE 3.6 A possible "controller" for a 3-bit bus, to maintain one of the 3 registers connected to the bus.

readC to 0 when selectA is 1, and keeps them in this state even if selectA is reset to 0. Similarly, it sets readB (resp. readC) to 1, and resets the others to 0, when selectB (resp. selectC) is 1. At most one "select" input must be set to 1 at a time.

3.4 Example

We can now use the circuits presented in this chapter to memorize the intermediate results of an Arithmetic and Logic Unit (ALU). The circuit in Figure 3.7 connects the ALU from Figure 2.10 (with 3 bits only to simplify) to 3 input bits, a RAM, and two registers named R0 and R1, via a bus, as schematized in Figure 3.8.

Thanks to the bus, this circuit can copy values from any source (Input, RAM, R0, or the ALU's output) to any sink (RAM, R0, or R1), which gives $4 * 3 = 12$ possibilities. For example, computing $a + b - c$ can be done as follows:

- set the input to a and copy it in R0. For this, first send a *pulse* on selectINPUT (*i.e.*, set it to 1 for a short time and then reset it to 0). Then send a pulse on writeR0 (this memorizes a when writeR0 changes from 0 to 1).

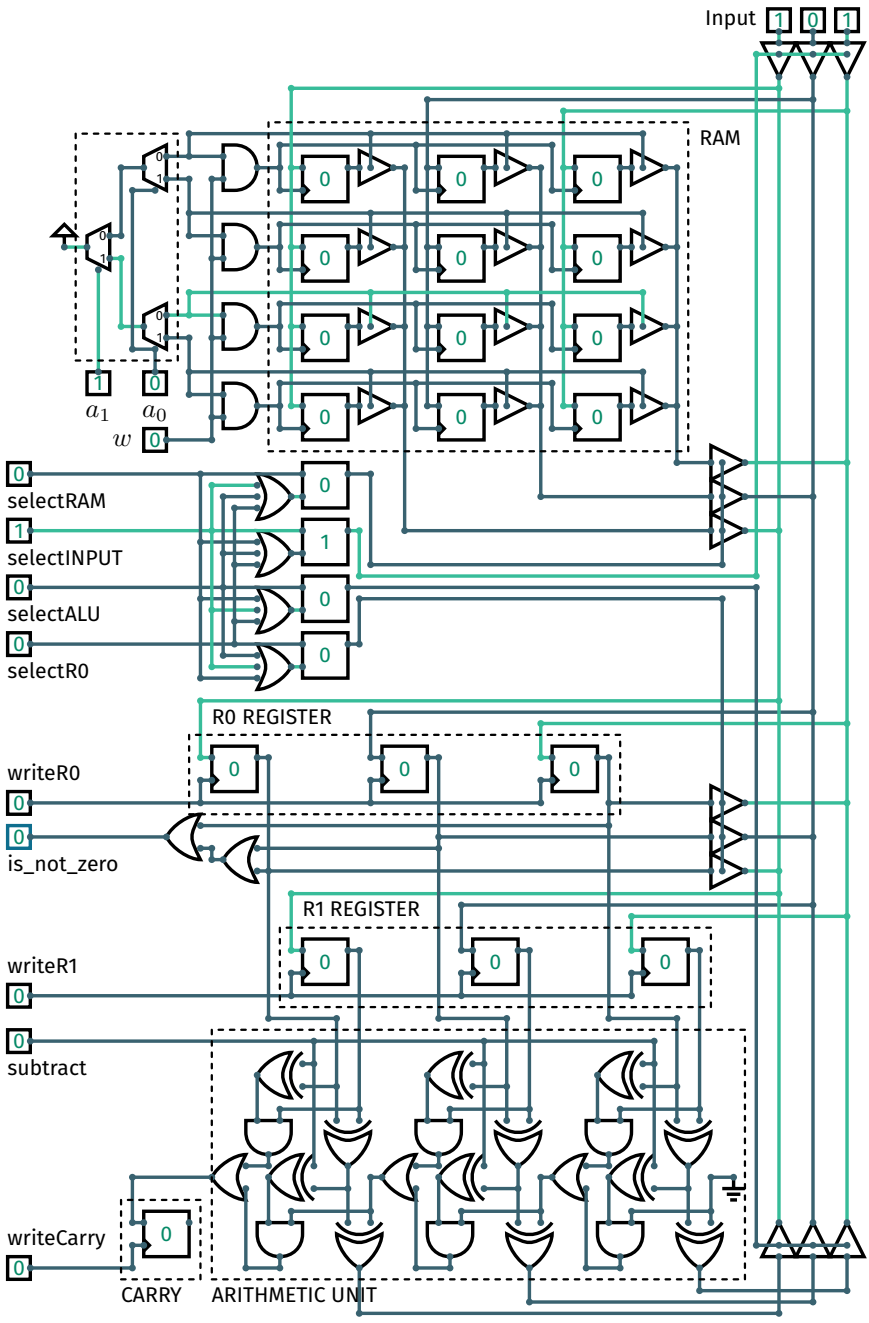


FIGURE 3.7 A 3-bit Arithmetic Unit (bottom) connected to 3 input bits (top right), a 4 values RAM (top left), and 2 registers R0 and R1 (middle), via a bus (right).

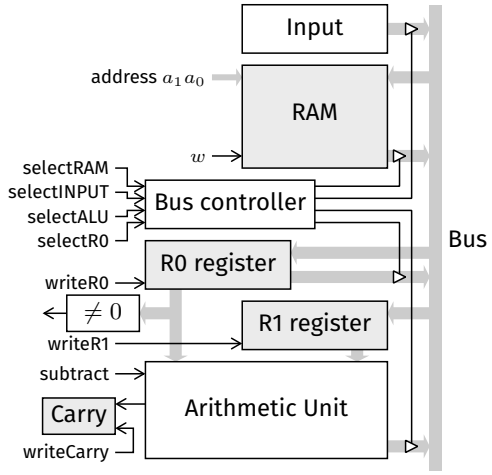


FIGURE 3.8 The block diagram corresponding to Figure 3.7.

- set the input to b and copy it in R1 by sending a pulse on writeR1 (there is no need to send a pulse on selectINPUT first since the bus controller keeps the last selected source connected).
- set and maintain "subtract" to 0, send a pulse on selectALU, and wait a short time until the ALU has computed $a + b$. Then store the result in R0 by sending a pulse on writeR0.
- set the input to c and copy it in R1 by sending a pulse on selectINPUT, followed by a pulse on writeR1.
- set and maintain "subtract" to 1, send a pulse on selectALU, and wait a short time until the ALU has computed $a + b$ (in R0) minus c (in R1). Then store the result in R0 by sending a pulse on writeR0.

At this stage we can use the is_not_zero and carry outputs, for instance, to test if $a + b - c$ is equal to 0, or to compare $a + b$ and c. We can also store $a + b - c$ in RAM for later use. For this it suffice to send a pulse on selectR0, followed by a pulse on w, after having set $a_1 a_0$ to the desired destination address.

CHAPTER

4 Control Circuits

Thanks to registers and memory circuits we can use an Arithmetic and Logic Unit to perform computations without having to mentally memorize intermediate results. Instead, as shown in the previous chapter, we can simply send a series of pulse signals on the correct inputs, and in the correct order. But this requires to memorize this procedure. And executing it manually is very slow and error-prone, even if the circuit does each operation very quickly. To solve the first issue a solution is to store some description of the desired procedure in Random Access Memory. To solve the second one we can use new circuits to execute this procedure for us, by sending the appropriate pulses. This chapter explains how this can be done.

4.1 Instructions

A procedure such as the one presented in Section 3.4 could be described in an abstract way as "read 3 numbers a, b, and c in input, compute $a + b - c$, and write the result in RAM at address x". However, representing such descriptions with one or more numbers which can be stored in RAM is not easy. And figuring out which pulses to send to execute them would also be quite complicated.

This procedure can also be described as a sequence of elementary actions: "wait an input value", "send a pulse on selectInput", "send a pulse on writeR0", "wait an input value", "send a pulse on writeR1", etc. Each action can easily be represented with a small number (*e.g.*, 0 for "wait an input value", 1 for "send a pulse on selectInput", etc). And each action is easy to execute. However, such a description is hard to design and to understand for humans (because its high level meaning is lost in the details).

A trade-off is to describe this procedure with more abstract actions, but not too abstract either, called *instructions*. For instance, an instruction could be "wait an input value and store it in R0", "add the values in R0 and R1 and store the result in R0", or "copy the value in R0 in RAM, at address 3". As shown below, such instructions are not too complex to represent with a number, called their *encoding* (to store them in memory). And they are still quite simple to execute (each instruction only requires sending 2 or 3 pulses at most). Finally, a sequence of instructions is less hard to design and to understand than the corresponding sequence of pulses (but still quite hard; we address this problem in Part 3).

Simple procedures, also called *programs*, can be described with a sequence of

instructions, to be executed one after the other. For this we can store their encoding one after the other in memory, *i.e.*, at consecutive addresses. Then, after the instruction at address a is executed, the one at address $a + 1$ should execute[1].

4.1.1 Jump instructions

Some programs need to repeat the same sequence of instructions two or more times. For instance, a "calculator" program needs to repeat forever the same sequence (read two numbers in input, compute and output their sum, repeat). In other words, after the last instruction of the sequence is executed, the instruction at the next address should *not* be executed. Instead, execution should restart at the first instruction of the sequence. This can be described with a so called *jump instruction*. A "jump to a" instruction specifies that the next instruction to execute is the one at address a.

In many cases a sequence of instructions must be repeated a precise number of times. For instance, to compute $a * b$ with the circuit of Figure 3.7, we can repeat b times a sequence adding a to R0 (initially set to 0). Then, after a has been added to R0, there are two cases: either we need to repeat the sequence again, or we need to continue with the rest of the program (*e.g.*, output the result $a * b$). This can be described with a *conditional jump* instruction. Such an instruction either jumps to a given address, or continues to the instruction at the next address, depending on some condition (for instance, whether R0 is equal to 0 or not).

4.2 A toy instruction set

To illustrate the above discussion we define in this section a concrete set of instructions for a circuit such as the one in Figure 3.7 (*i.e.*, with a RAM and two registers R0 and R1 as input of a very basic Arithmetic Unit). These instructions are the following:

- Memory:
  - the Load instruction copies the value at a given address a into the R0 register.
  - the Store instruction copies the value in the R0 register at a given address a.
- Arithmetic:
  - the Add instruction adds the value at address a to the value in the R0 register, and stores the result in R0.
  - the Subtract instruction subtracts the value at address a from the value in the R0 register, and stores the result in R0.
- Jumps:
  - the Jump instruction specifies that the next instruction to execute is the one at address a.

[1] Assuming that each encoded instruction can fit in the n bits between two consecutive addresses.

  - the Jump If Zero instruction specifies that the next instruction to execute is the one at address a if the value in R0 is equal to 0. Otherwise execution continues with the instruction at the next address.
  - the Jump If Carry instruction specifies that the next instruction to execute is the one at address a if the last Add or Subtract instruction produced a non-zero carry bit. Otherwise execution continues with the instruction at the next address.

- Input and output:

  - the Input instruction waits for the user to press a button, and then copies the value on the input wires into the R0 register.
  - the Output instruction displays the value in the R0 register, and then waits until the user presses a button.

4.2.1 Encoding

The above *instruction set* contains 9 instructions. We can thus give them numbers from 0 to 8, called *operation codes*, or *opcodes*. This requires at least 4 bits to encode each instruction. But all instructions except the last two have an associated address a, called an *operand*. This operand must also be encoded as part of the instruction, which requires more bits.

In the following we assume that the memory contains $2^5 = 32$ bytes, each with their own address, and that R0, R1, and the Arithmetic Unit work on 8-bit values. We then use 5 bits per address, and we encode each instruction in one byte, as follows:

LDR	$\mathrm{R0} \leftarrow \mathrm{mem8}[a]$	0 0 1 a
STR	$\mathrm{R0} \rightarrow \mathrm{mem8}[a]$	0 0 0 a
ADD	$\mathrm{R0} \leftarrow \mathrm{R0} + \mathrm{mem8}[a]$	0 1 0 a
SUB	$\mathrm{R0} \leftarrow \mathrm{R0} - \mathrm{mem8}[a]$	0 1 1 a
JMP	jump to a	1 0 0 a
IFZ	if $\mathrm{R0} = 0$ then jump to a	1 0 1 a
IFC	if carry $\neq 0$ then jump to a	1 1 0 a
IN	$\mathrm{R0} \leftarrow$ input	1 1 1 0
OUT	$\mathrm{R0} \rightarrow$ output	1 1 1 1

The left column is the *instruction mnemonic*, an abbreviation of the instruction name. The middle column is a symbolic description of the effect each instruction. Here $dst \leftarrow src$ or $src \rightarrow dst$ means a copy of the value in src into dst, and $\mathrm{mem8}[a]$ means the 8-bit value at address a. Finally, the right column is the binary number corresponding to this instruction, *i.e.*, its encoding. For instance, the encoding of the LDR 7 instruction, which copies the byte at address $7 = 111_2$ into R0, is 001_2 followed by 7 encoded in 5 bits, 00111_2, which gives $00100111_2 = 39$.

4.2.2 Example program

With the above instruction set a "calculator" program adding numbers in an endless loop can be implemented as follows:

IN	R0 ← input	1 1 1 0 *0 0 0 0*	0
STR	R0 → mem8[*6*]	0 0 0 *0 0 1 1 0*	1
IN	R0 ← input	1 1 1 0 *0 0 0 0*	2
ADD	R0 ← R0 + mem8[*6*]	0 1 0 *0 0 1 1 0*	3
OUT	R0 → output	1 1 1 1 *0 0 0 0*	4
JMP	jump to *0*	1 0 0 *0 0 0 0 0*	5
(data)	the a number	0 0 0 0 0 0 0 0	6

where the left part gives the symbolic description of each instruction, and the right part their encoding and their address (in gray).

The first two instructions read a number a as input and store it at address 6. The next two instructions read a second number b, and add a to it. The last two instructions output the value in R0, which at this stage contains $a + b$, and jump back to the first instruction to add two new numbers. The next byte after these five instructions is the one used to store a.

4.2.3 Notes

Adding two 8-bit numbers can give a 9-bit number. For instance, $11111111_2 = 255$ plus 1 gives the 9-bit number $100000000_2 = 256$. However, the registers and the memory can only store 8-bit numbers, and the Arithmetic Unit can only use 8-bit numbers as input. Hence, in practice, and unless a program does something special with the carry bit (with the IFC instruction), all additions are *modulo* $2^8 = 256$. This means that adding a and b does not give $a+b$ but the remainder of the division of $a+b$ by 256. It is noted $(a+b) \mod 256$, where $x \mod m$ is defined as $x - \lfloor x/m \rfloor * m$. For instance, adding 255 and 1 gives 0[2].

Similarly, subtracting two numbers can give a negative result, but the registers and the memory can only store nonnegative numbers. Hence, in practice, and unless a program does something special with the carry bit, all subtractions are modulo 256 too. For instance, subtracting 1 from 0 gives 255 because $-1 \mod 256 = -1 - \lfloor -1/256 \rfloor * 256 = -1 - (-1) * 256 = 255$ (recall that $\lfloor y \rfloor$ means the integer part of y).

When $a+b$ differs from $(a+b) \mod 2^n$ we say that there is an (integer) *overflow* (where n is the Arithmetic Unit's "bit width"). We say the same when $a - b \neq (a - b) \mod 2^n$, $a * b \neq (a * b) \mod 2^n$, etc. With an Arithmetic Unit such as the one in Figure 2.10, there is an overflow if and only if the carry output is 1.

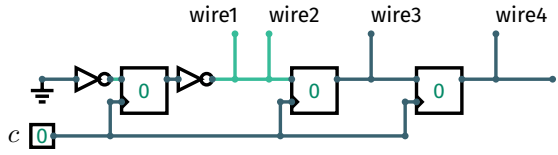
[2]This *modular arithmetic* is used in everyday life with hours. For example, 10 a.m plus 5 hours is 3 p.m because $(10 + 5) \mod 12 = 3$.

4.3 Control circuits

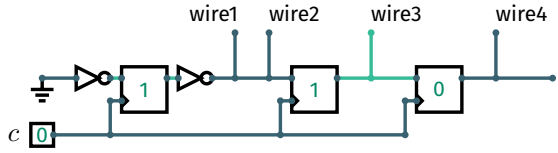
We now have a way to describe a sequence of instructions with some numbers stored in memory. The next step, as described in the introduction of this chapter, is to build a circuit to automatically *execute* these instructions. Which means sending a corresponding sequence of pulse signals on the registers, memory, and bus circuits. For instance, to execute an IN instruction with the circuit in Figure 3.7, one needs to:

- connect the input wires to the bus by sending a pulse on "selectInput".
- wait for the user to press a button.
- store the input value in R0 by sending a pulse on "writeR0".

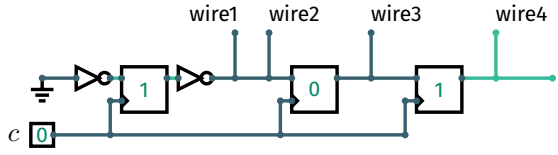
More generally, all instructions can be executed by sending appropriate pulses 1) on the correct wires, 2) in the correct order, and 3) at appropriate times (signals must have time to propagate throughout the circuit between two pulses). The first two items can be ensured with circuits of the following form:



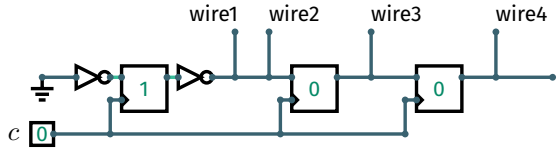
When this circuit is powered on "wire1" and "wire2" change from 0 to 1 due to the NOT gate. Changing c from 0 to 1 (and back to 0) makes the first and second D flip-flops memorize 1. This resets "wire1" and "wire2" to 0, and sets "wire3" to 1:



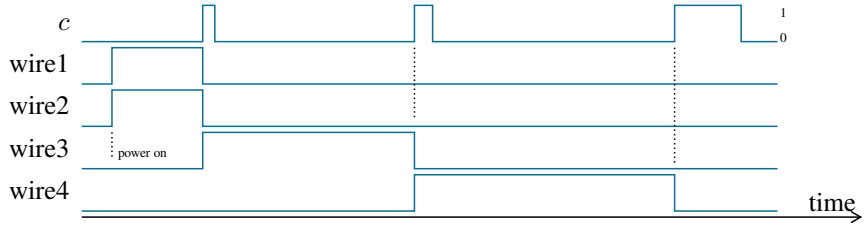
Changing c from 0 to 1 (and back to 0) again makes the second and third flip-flops memorize 0 and 1, respectively. This resets "wire3" to 0, and sets "wire4" to 1:



Finally, changing c from 0 to 1 (and back to 0) one more time resets "wire4" to 0:



In other words, with a series of pulses on the c input, one gets two simultaneous pulses on "wire1" and "wire2", followed by a pulse on "wire3" and then on "wire4":



Each wire pulse starts and ends at the precise moment when c switches from 0 to 1. This shows that, with circuits like the one above, it is possible to send pulses on specific wires, in a specific order. The only requirement is the ability to send a series of pulses on a shared input c, which can be done with a *clock*.

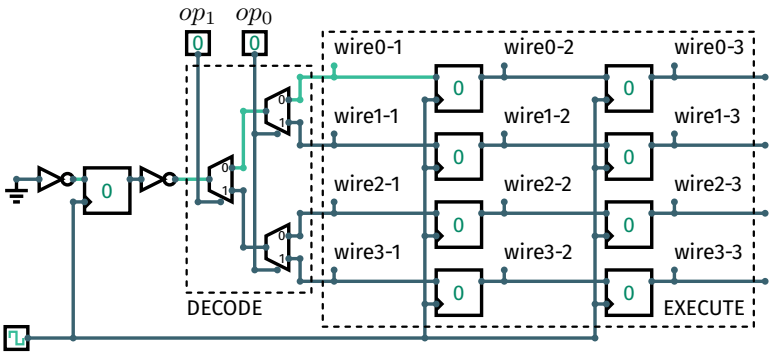
4.3.1 Clock

A clock is a circuit which generates a signal switching between 0 and 1 at a constant frequency. A clock can be implemented in many ways. For instance, one could use a pendulum acting on a switch. But this would not be very practical, and can not produce high frequencies. Instead, a frequently used method is to use the oscillations of a crystal. Crystals can oscillate one million times per second or more (*i.e.*, at 1 MHz or more). In the following we represent a clock with the symbol on the left:

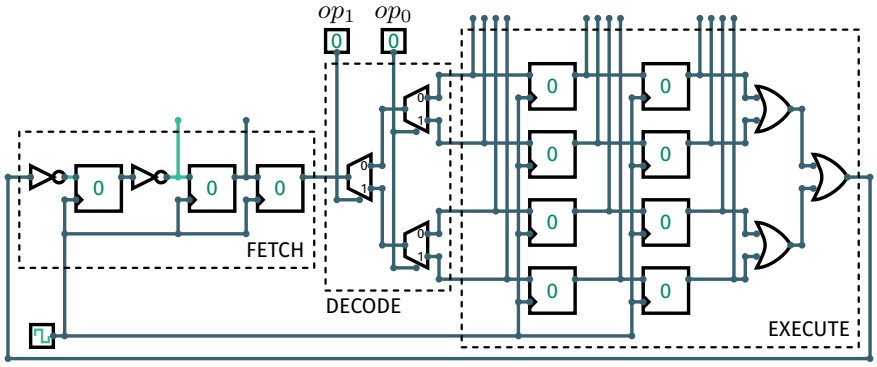
A clock generates the signal shown above (right). A *period*, also called a *clock cycle*, is the time between successive pulses. The clock frequency is the number of pulses per second, *i.e.*, the inverse of its period. Increasing the frequency increases the number of instructions which are executed per second. However, the frequency cannot be increased without limit. Indeed, there must be enough time between two pulses for signals to propagate throughout the circuit. For instance, computing an addition in an Arithmetic Unit takes some time, because the input values have to propagate through all its logic gates, up to the carry output. If a pulse is sent to write the sum in a register before this delay, a wrong result will be stored.

4.3.2 Control loop

A circuit like the one above can generate a sequence of pulses to execute one instruction. But each type of instruction needs a different sequence of pulses to be executed. The solution is to use several circuits like this, one per type of instruction. And to connect them to a binary decoder, so that the correct subcircuit is used depending on the instruction opcode. For instance, if there are only 4 different opcodes, we can use a circuit similar to the following:



Depending on the two bits of the instruction opcode, $op_1 op_0$, the above circuit sends pulses on "wire0-1" to "wire0-3", or on "wire1-1" to "wire1-3", etc. Before this the instruction must be read in memory, so that $op_1 op_0$ contain the correct values. This can be done, as shown later, with a so called FETCH circuit sending an appropriate sequence of pulses. Finally, after the instruction has been executed, the next one must be fetched, decoded, and executed. For this it suffice to connect the outputs of the EXECUTE subcircuit back to the input of the FETCH circuit:



In this way we get a pulse which loops forever in the FETCH, DECODE and EXECUTE circuits, each time going through a specific EXECUTE subcircuit.

4.4 A toy control unit

To illustrate the above discussions we design in this section a very basic *control unit* for the circuit of Figure 3.8 (with an 8-bit *architecture*, *i.e.*, an 8-bit Arithmetic Unit, 8-bit registers, etc). As its name implies, a control unit controls the rest of the circuit, called the *processing unit* (*i.e.*, the Arithmetic and Logic Unit, the registers, the bus, etc). It does so by executing instructions stored in memory. We assume here that these instructions are those defined in Section 4.2.

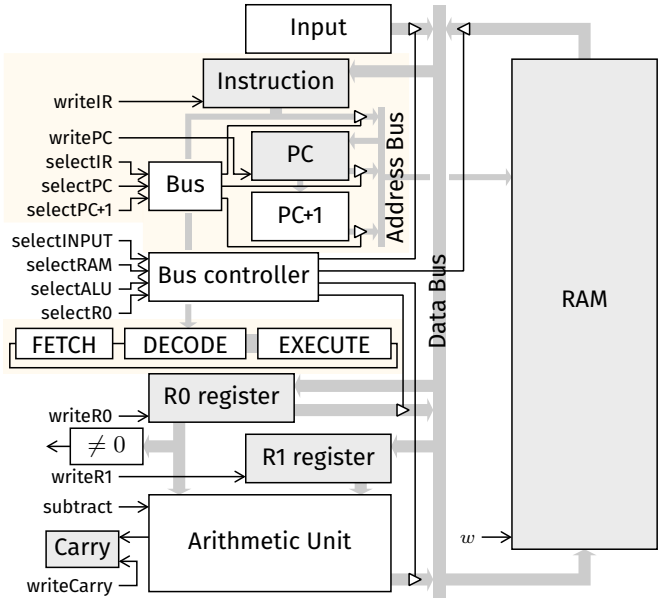


FIGURE 4.1 A basic control unit (yellow background) for the circuit in Figure 3.8 (white background), with the instruction set of Section 4.2.

The core part of our example control unit is a control loop circuit with FETCH, DECODE, and EXECUTE subcircuits, as presented above. To implement it we need two new registers, in addition to R0 and R1 (see Figure 4.1):

- the *Program Counter* (PC) register stores the address of the instruction being currently executed or, once it has been executed, the address of the next instruction to execute. Since addresses use only 5 bits, this register is a 5-bit register.
- the *Instruction Register* (IR) stores the encoding of the instruction being currently executed. This 8-bit register stores a copy of the original instruction in memory. This is necessary to have access to its value during its execution, which might require reading or writing values at other addresses in memory.

Once an instruction has been executed, the Program Counter value must be incremented by one to execute the next instruction. Unless the last instruction was a jump. In this case the Program Counter value must be replaced with the operand of this jump instruction. To do this we include two more circuits in our control unit (see Figure 4.1):

- a 5-bit incrementer, which computes "PC+1", *i.e.*, the value in the Program Counter register plus 1. This is an adder circuit similar to the one in Section 2.4.1, simplified for the case where one input is always 1.
- a 5-bit *address bus*, to which we connect the Program Counter, the output of the above incrementer, and the 5 least significant bits of the Instruction Register (*i.e.*,

the address operand). This bus is also connected to the address decoder of the RAM and thus selects which address to read or write to.

Thanks to these components, we can increment the Program Counter by connecting the output of the incrementer to the address bus, and by sending a pulse on "writePC" to store this value (see Figure 4.1). Likewise, we can replace the Program Counter with the operand of a jump instruction by connecting the Instruction Register to the address bus, and by sending a pulse on "writePC".

4.4.1 FETCH circuit

With the above architecture, fetching an instruction can be done as follows:

- send simultaneous pulses on "selectPC" and "selectRAM" to read the value in memory at the address stored in the Program Counter, and to get it on the data bus.
- send a pulse on "writeIR" to write this value in the Instruction Register.
- send a pulse on "selectIR" to prepare reading or writing a value at the address operand of the new instruction. Technically this step is part of the instruction's execution, but we include it in the FETCH circuit to avoid duplications (it is common to all instructions except IN and OUT).

4.4.2 DECODE circuit

Decoding an instruction can be done with a binary decoder with 3 inputs, namely the 3 most significant bits of the Instruction Register. Plus a single demultiplexer, controlled by the 4^{th} most significant bit, in order to distinguish the IN and OUT instructions (the 3 most significant bits are 111_2 for both instructions).

4.4.3 EXECUTE circuit

The EXECUTE circuit has 9 subcircuits, one per type of instruction:

LDR This subcircuit sends a pulse on "writeR0" to store the value read from memory at the instruction's address operand (selected by the last step of the FETCH circuit). It then increments the PC value with a pulse on "selectPC+1", followed by one on "writePC".

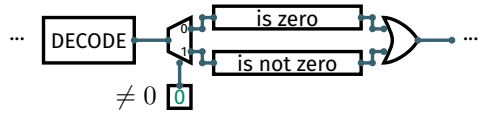
STR This subcircuit sends a pulse on "selectR0", followed by a pulse on w to store R0's value in memory, at the instruction's address operand (selected by the last step of the FETCH circuit). It then increments the PC value, as above.

ADD This subcircuit sends a pulse on "writeR1" to store the value at the instruction's address operand in R1. It then sends a pulse on "selectALU" to get the sum of the values in R0 and R1 on the bus, followed by simultaneous pulses on "writeR0" and "writeCarry" to write it in R0 and Carry. It then increments the PC value as above.

SUB This subcircuit is almost the same as the ADD subcircuit. It just sends an additional pulse on "subtract", at the same time as the "selectALU" pulse. These pulses last until the one on "writeR0" starts. This ensures that the correct result, the difference of R0 and R1 values, is written in R0.

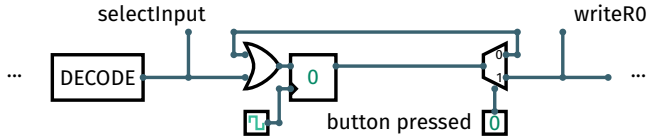
JMP This subcircuit just sends a pulse on "writePC" to replace the Program Counter with the instruction's address operand (selected by the last step of the FETCH circuit).

IFZ This subcircuit has two branches. The first, executed if the value in R0 is 0, is the same as the JMP subcircuit. The second, executed if R0's value is not 0, increments the PC value as for non-jump instructions. The two branches are connected to a demultiplexer controlled by the "$\neq 0$" signal (see Figure 4.1):



IFC This subcircuit is almost the same as the IFZ one, except that its demultiplexer is controlled by the value of the Carry register.

IN This subcircuit sends a pulse on "selectInput", and then waits until a button is pressed. This can be done with a loop similar to the control loop, with a demultiplexer to either wait, or to continue with the next instruction:



In the latter case, this subcircuit sends a pulse on "writeR0", and then increments the Program Counter as above.

OUT This subcircuit sends a pulse on "selectR0" and then waits until a button is pressed, with the same method as above. It then increments the Program Counter's value.

CHAPTER

5 A Toy Microprocessor

Thanks to control circuits we can now run a program stored in memory in an automated way, which is much faster and safer than using a processing unit manually. Together, a control unit and the processing unit it controls form a *Central Processing Unit* (CPU), also called a *microprocessor*. This chapter presents a very simple one, based on the design introduced in the previous chapter. It also illustrates its capabilities with two example programs.

5.1 Implementation

The circuit in Figure 5.1 is a microprocessor for the instruction set of Section 4.2. It is based on the circuit of Figure 3.7 (extended to an 8-bit architecture), augmented with the control unit designed in Section 4.4. Its physical layout matches more or less the block diagram in Figure 4.1. This section only presents the parts of this circuit which have not been explained in the previous chapters.

5.1.1 Input and output

The input is made of 8 pins which are connected to the bus via tristate buffers, plus a *Light Emitting Diode* (LED). When on, the LED indicates that the microprocessor is waiting a value on the input pins. The user must set them to 0 or 1 as desired, and press a button when done. Similarly, the output is made of 8 pins connected to the bus, plus a LED. When on, the LED indicates that a value is available on the output pins, and that the user must press a button to resume the execution of the program.

The two LEDs are connected to the two loops inside the EXECUTE subcircuit (for the IN and OUT instructions), which wait for the user to press a button. In this way they are turned on when an IN or OUT instruction starts, and turned off when the button is pressed.

The button is a push button, used for both IN and OUT instructions. Due to its speed, the microprocessor might execute several instructions during the time this button stays pressed. In particular, it might execute several IN or OUT instructions. In this case the user would not have the time to enter a second value, or to read the first output value. To avoid this, the push button is connected to a circuit which generates a short pulse (lasting one clock cycle) when it transitions from the “open” to the

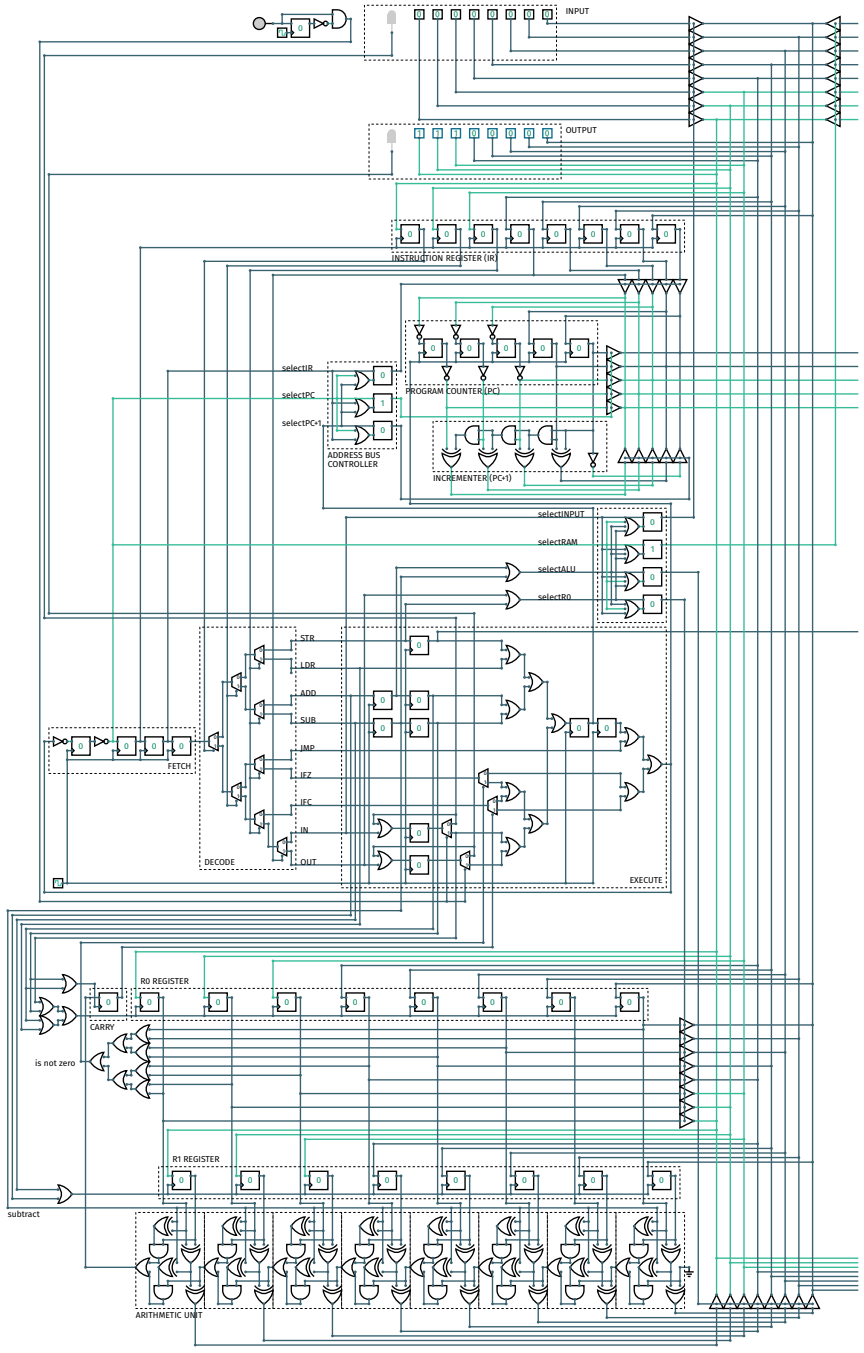
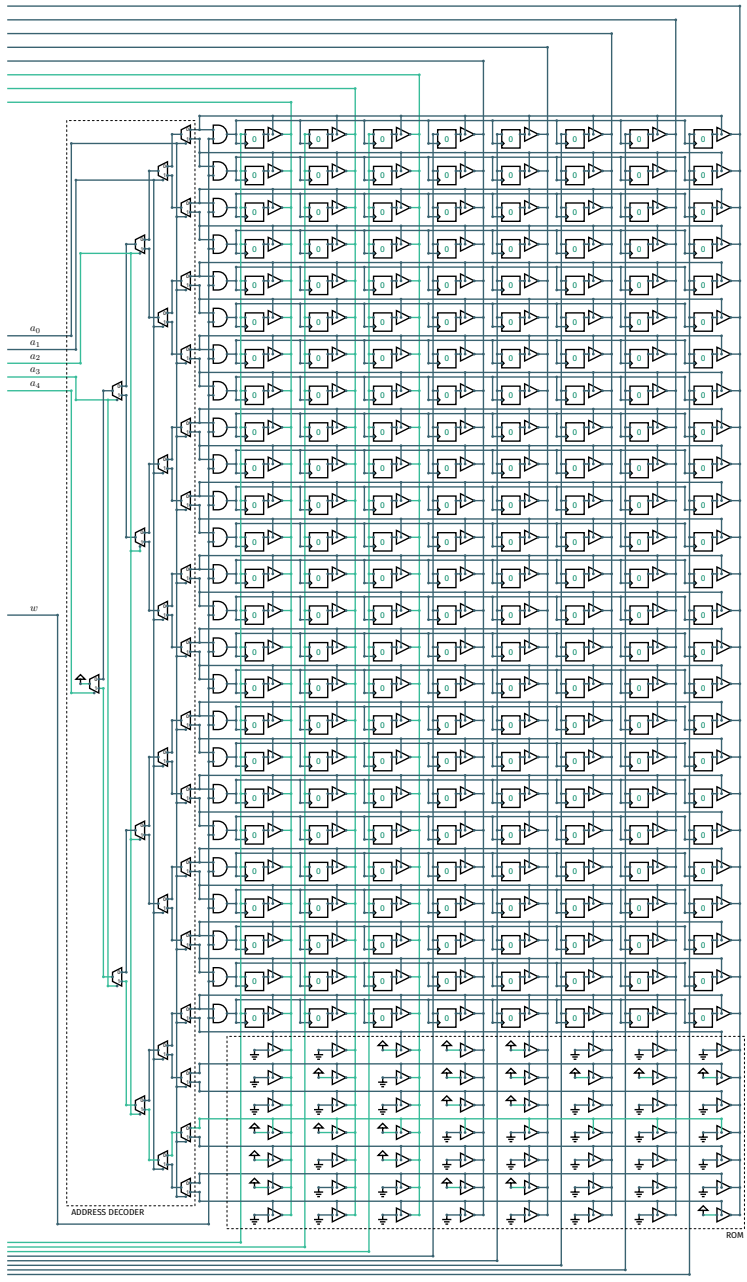
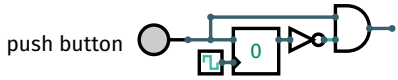


FIGURE 5.1 A toy microprocessor, implementing the instruction set of Section 4.2,



and its 25 bytes Random Access Memory and 7 bytes Read-Only Memory.

"closed" state. It is then necessary to release and press again this button to generate a new pulse. The pulse generator circuit is the following:



The output of the D flip-flop is the push button's state at the previous clock cycle. Hence the output of the AND gate is 1 if and only if the button is currently pressed, and was not pressed at the previous clock cycle. In other words, it is 1 only when the button transitions from the "open" to the "closed" state, as desired.

5.1.2 Boot program

To run a program with our microprocessor we first need to store it in memory (as we assumed earlier, all flip-flops are initially 0, and thus all the memory too). But we no longer have any manual control over the memory, since we transferred it to the control unit. Hence the only way to store anything in memory is to use a program reading values in input and writing them in memory. But then we need to store this program in memory first! To solve this chicken and egg problem, a solution is to store it in a Read-Only Memory (ROM), *i.e.*, a memory containing immutable values.

This is what the memory circuit in Figure 5.1 does: values at addresses 0 to 25 excluded (which is noted $[0, 25[= [0, 24]$) can be read and written, but values at addresses $[25, 32[$ cannot be modified. On the other hand, the latter do not need to be initialized first. They can thus contain a program which is ready to be executed when the microprocessor *boots*, *i.e.*, when it is powered on. The ROM in Figure 5.1 contains the following *boot program*:

LDR	R0 $\leftarrow$ mem8[*24*]	0 0 1 | *1 1 0 0 0*	25
ADD	R0 $\leftarrow$ R0 + mem8[*31*]	0 1 0 | *1 1 1 1 1*	26
STR	R0 $\rightarrow$ mem8[*24*]	0 0 0 | *1 1 0 0 0*	27
IN	R0 $\leftarrow$ input	1 1 1 0 | *0 0 0 0*	28
IFZ	if R0 = 0 then jump to *0*	1 0 1 | *0 0 0 0 0*	29
JMP	jump to *24*	1 0 0 | *1 1 0 0 0*	30
(data)	the value 1	0 0 0 0 0 0 0 1	31

Moreover, the Program Counter register uses NOT gates around its three most significant bits, which give it the initial value $11100_2 = 28$. Hence, when it boots, our microprocessor starts by executing the IN instruction at address 28. It thus waits a value v_0 in input. If this value is not 0, the instruction at address 30 jumps to address 24. Initially, the value at this address, in RAM, is 0. This corresponds to a STR 0 instruction, which thus stores v_0 at address 0. The next 3 instructions, in $[25, 27[$, add 1 to the value at address 24. This value thus becomes 1, which corresponds to STR 1. Hence, after a second value v_1 is read in input, and if it is not 0, the new instruction at address 24 stores v_1 at address 1. And so on with the next input values: v_2 stored at

address 2, v_3 at address 3, etc. This loop ends when the input value is 0. In this case the IFZ instruction at address 29 jumps to address 0. The effect is to run the program $v_0, v_1, \ldots v_n$ stored in memory by the previous steps, starting at address 0.

5.2 Example programs

The above microprocessor can run several "useful" programs. A first example is the adder program of Section 4.2.2. To run it, one first need to enter its 6 instructions with the boot program, followed by a 0 value. After that the programs reads two values in input, outputs their sum, and repeats these two steps forever.

5.2.1 Multiplier

Another example is a program to compute products. Since the Arithmetic Unit cannot perform multiplications, nor shifts (see Section 1.2.3), we need a program computing them with repeated additions. The general *algorithm*, *i.e.*, the specification of the main steps that this program must follow, is the following:

1. read two numbers a and b in input, and initialize c to 0.
2. if $b = 0$ go to step 4.
3. otherwise, subtract 1 from b, add a to c, and go back to step 2.
4. output c and go back to step 1 to compute another product.

This algorithm executes step 3 b times, and thus adds a to c b times. Hence, when $b = 0$, c contains $a * b$ and can be output. Assuming that a, b, and c are stored at addresses A, B, and C, respectively, and that the values at addresses $ZERO$ and ONE are 0 and 1, this gives the following (abstract) instructions:

- step 1: IN, STR A (store an input value in a), IN, STR B (store an input value in b), LDR ZERO, STR C (store 0 in c).
- step 2: LDR B, IFZ "step4".
- step 3: SUB ONE, STR B (subtract 1 from B, still in R0), LDR C, ADD A, STR C (add a to c), JMP "step2" (go back to step 2).
- step 4: LDR C, OUT (output c), JMP 0 (go back to step 1).

Step 1 has 6 instructions, and thus step 2 starts at address 6. Step 2 and 3 have a total of 8 instructions, and thus step 4 starts at address $6 + 8 = 14$. We can thus replace IFZ "step4" with IFZ 14, and JMP "step2" with JMP 6. We can also store a, b, c, 0, and 1 after the last instruction, *i.e.*, starting at address 17 (since there are 17 instructions). In the following we use $ONE = 17$, $ZERO = 18$, $A = 19$, $B = 20$ and $C = 21$. This leads to the following *machine code*, *i.e.*, a list of instructions in binary form that the machine (the microprocessor) can directly execute:

IN	R0 ← input	1 1 1 0 | 0 0 0 0	0
STR	R0 → mem8[19]	0 0 0 | 1 0 0 1 1	1
IN	R0 ← input	1 1 1 0 | 0 0 0 0	2

STR	R0 $\rightarrow$ mem8[*20*]	0 0 0 *1 0 1 0 0*	3
LDR	R0 $\leftarrow$ mem8[*18*]	0 0 1 *1 0 0 1 0*	4
STR	R0 $\rightarrow$ mem8[*21*]	0 0 0 *1 0 1 0 1*	5
LDR	R0 $\leftarrow$ mem8[*20*]	0 0 1 *1 0 1 0 0*	6
IFZ	if R0 $= 0$ then jump to *14*	1 0 1 *0 1 1 1 0*	7
SUB	R0 $\leftarrow$ R0 $-$ mem8[*17*]	0 1 1 *1 0 0 0 1*	8
STR	R0 $\rightarrow$ mem8[*20*]	0 0 0 *1 0 1 0 0*	9
LDR	R0 $\leftarrow$ mem8[*21*]	0 0 1 *1 0 1 0 1*	10
ADD	R0 $\leftarrow$ R0 $+$ mem8[*19*]	0 1 0 *1 0 0 1 1*	11
STR	R0 $\rightarrow$ mem8[*21*]	0 0 0 *1 0 1 0 1*	12
JMP	jump to *6*	1 0 0 *0 0 1 1 0*	13
LDR	R0 $\leftarrow$ mem8[*21*]	0 0 1 *1 0 1 0 1*	14
OUT	R0 $\rightarrow$ output	1 1 1 1 *0 0 0 0*	15
JMP	jump to *0*	1 0 0 *0 0 0 0 0*	16
(data)	the value 1	0 0 0 0 0 0 0 1	17
(data)	the value 0	0 0 0 0 0 0 0 0	18
(data)	the a number	0 0 0 0 0 0 0 0	19
(data)	the b number	0 0 0 0 0 0 0 0	20
(data)	the c number	0 0 0 0 0 0 0 0	21

To run it one fist need to enter its 17 instructions, plus the ONE value, followed by a 0, with the boot program.

5.2.2 Prime numbers enumerator

A third and last example is a program which outputs all the prime numbers less than or equal to 255 (the maximum value of an 8-bit number). For this the general algorithm is the following:

1. add 1 to n, supposed initially equal to 1.
2. if n is prime go to step 4.
3. go back to step 1.
4. output n and go back to step 1 to find the next prime number.

This algorithm tests each number n one by one, in increasing order, and starting from $n = 2$. Step 2 needs to check whether n can be divided by some number f in $[2, n[$. For this, a simple method is to check all values of f in decreasing order, from $n - 1$ to 2. Step 2 can then be replaced with the following algorithm:

1. initialize f to n.
2. subtract 1 from f.
3. if $f = 1$, n is prime, stop (all f values have been tested and no divisor was found).
4. if f does not divide n, go back to step 2.

Finally, to check whether f divides n or not, and since the Arithmetic Unit cannot perform divisions, we can use an algorithm which repeatedly subtracts f from n. If this process ends with 0 then f divides n, otherwise it does not:

1. initialize r to n.
2. subtract f from r.
3. if $r = 0$, f divides n, n is not prime, stop.
4. if $r < 0$, f does not divide n, stop.
5. go back to step 2 to continue the division of n by f.

By putting together the above partial algorithms we get the following complete algorithm:

1. add 1 to n, supposed initially equal to 1, and initialize f to n.
2. subtract 1 from f.
3. if $f = 1$ then n is prime, go to step 9.
4. initialize r to n.
5. subtract f from r.
6. if $r = 0$, f divides n, n is not prime. Go to step 1 to try the next n.
7. if $r < 0$, f does not divide n, go back to step 2 to try the next f.
8. go back to step 5 to continue the division of n by f.
9. output n and go back to step 1 to find the next prime number.

Assuming that n and f are stored at addresses N and F, respectively, that the value at address ONE is 1, and by storing r in R0, this gives the following (abstract) instructions:

- step 1: LDR N, ADD ONE, STR N (add 1 to n), STR F (initialize f to n).
- step 2: LDR F, SUB ONE, STR F (subtract 1 from f).
- step 3: SUB ONE, IFZ "step 9" (if $f - 1 = 0$ go to step 9).
- step 4: LDR N (initialize r to n).
- step 5: SUB F (subtract f from r).
- step 6: IFZ 0 (if $r = 0$, go back to step 1).
- step 7: IFC "step 2" (if $r < 0$, go back to step 2).
- step 8: JMP "step 5"
- step 9: LDR N, OUT, JMP 0 (output n and go back to step 1).

Finally, with the same method as for the multiplier program, we can replace IFZ "step 9", IFC "step 2", JMP "step5" with IFZ 14, IFC 4, and JMP 10, respectively. And we can store n, f and 1 after the last instruction, *i.e.*, starting at address 17. By using $ONE = 17$, $N = 18$, and $F = 19$ we get the following machine code:

LDR	R0 ← mem8[*18*]	0 0 1 | *1 0 0 1 0*	0
ADD	R0 ← R0 + mem8[*17*]	0 1 0 | *1 0 0 0 1*	1
STR	R0 → mem8[*18*]	0 0 0 | *1 0 0 1 0*	2
STR	R0 → mem8[*19*]	0 0 0 | *1 0 0 1 1*	3
LDR	R0 ← mem8[*19*]	0 0 1 | *1 0 0 1 1*	4

SUB	R0 $\leftarrow$ R0 $-$ mem8[*17*]	0 1 1 *1 0 0 0 1*	5
STR	R0 $\rightarrow$ mem8[*19*]	0 0 0 *1 0 0 1 1*	6
SUB	R0 $\leftarrow$ R0 $-$ mem8[*17*]	0 1 1 *1 0 0 0 1*	7
IFZ	if R0 $= 0$ then jump to *14*	1 0 1 *0 1 1 1 0*	8
LDR	R0 $\leftarrow$ mem8[*18*]	0 0 1 *1 0 0 1 0*	9
SUB	R0 $\leftarrow$ R0 $-$ mem8[*19*]	0 1 1 *1 0 0 1 1*	10
IFZ	if R0 $= 0$ then jump to *0*	1 0 1 *0 0 0 0 0*	11
IFC	if carry $\neq 0$ then jump to *4*	1 1 0 *0 0 1 0 0*	12
JMP	jump to *10*	1 0 0 *0 1 0 1 0*	13
LDR	R0 $\leftarrow$ mem8[*18*]	0 0 1 *1 0 0 1 0*	14
OUT	R0 $\rightarrow$ output	1 1 1 1 *0 0 0 0*	15
JMP	jump to *1*	1 0 0 *0 0 0 0 1*	16
(data)	the value 1	0 0 0 0 0 0 0 1	17
(data)	the value n	0 0 0 0 0 0 0 1	18
(data)	the value f	0 0 0 0 0 0 0 0	19

To run it one fist need to enter its 17 instructions, plus the ONE and initial n values, followed by a 0, with the boot program.

Conclusion

Microprocessors are made of electrically controlled switches, assembled into logic gates, themselves assembled into arithmetic and logic circuits, registers, memories, and control circuits. Most of them use the architecture presented in this part, made of a control unit, a processing unit, input and output mechanisms, and a memory containing data and instructions. It is called the von Neumann architecture because it was first published by John von Neumann in 1945.

The toy microprocessor presented in the previous chapter is extremely limited, especially because it can only address 32 bytes of memory. To solve this problem, most modern microprocessors use a 32 or 64-bit architecture. A 32-bit architecture is based on 32-bit registers, buses, and Arithmetic and Logic Units. A 32-bit microprocessor generally use 32-bit addresses, and can thus use up to $2^{32} = 4$ GB of memory. It can also perform computations on 32-bit values, called *words*[1]. However, even with more memory, microprocessors can only do computations on numbers. But then how can computers create or edit text, music or videos?

Data representation

The answer is that text, sound, images, videos, or in fact any information, can be represented with numbers. And doing some computations on these numbers can edit this information.

For instance, one could represent letters from "a" to "z" with numbers from 0 to 26 (excluded), letters from "A" to "Z" with numbers from 26 to 52 (excluded), a space with 52, a dot with 53, etc. Then, for instance, a program reading bytes and adding 26 to those smaller than 26 can capitalize text. A program can also edit text based on keys typed by the user, as shown in Chapter 14. Or transform text into program instructions which, as explained in Section 4.2, can also be represented with numbers (see Part 3).

Similarly, one can *digitize* sound, which is a pressure wave, by measuring the pressure many times per second, and by storing each measurement in one byte (or more). Then, for instance, a program reading pairs of bytes, and computing their average, can mix two sound samples to produce a new one.

Likewise, one can digitize an image by decomposing it in a grid of *pixels* (for "picture element"), and by representing the luminosity of each pixel with one byte (from 0 for black to 255 for white). These values can then be stored in memory one after the other, for example from left to right and from top to bottom. Then, for instance, a program reading bytes and subtracting them from 255 can compute the

[1] A word is a 32-bit value in a 32-bit microprocessor, a 64-bit value in a 64-bit microprocessor, etc.

negative of an image. A color image can be represented in a similar way, with 3 values per pixel (for the intensity of the red, green, and blue components). And a video can be represented as a sequence of images (typically 24 or 25 per second).

Further readings

The design of our toy microprocessor, although sufficient to explain the main ideas, is not fully representative of real ones. For instance, real microprocessors do not wait for input values. Instead, when a new input value is available, it interrupts the microprocessor (like a notification or a phone call interrupts what you are doing). To learn more about the design and architecture of real microprocessors, you can read the following books:

- "Digital Design and Computer Architecture, RISC-V Edition" [12]. This book presents the CMOS technology (the most frequently used to build real microprocessors), gives more details about logical operations (negation, conjunctions, disjonctions, etc), explains how programs can be used to design complex circuits, and presents some advanced microprocessor architectures used to improve performance.
- "Computer Architecture" [10]. This book gives a historic perspective on computer design, gives more details about number systems, data representation, input and output mechanisms, etc. It also presents several microprocessor architecture styles.

PART

2 A Basic Input Output System

Introduction

This part has two goals. The first one is to assemble the hardware components of our toy computer, mainly a keyboard, a screen, and an *Arduino Due* board. This board provides in a single chip a microprocessor, various memories, and integrated circuits to control external devices (such as light emitting diodes, motors, sensors, but also keyboards or displays). We present how these components work, how to connect them together, and how programs can use them. The second goal of this part is to install on this assembled computer a very basic input and output system, using the keyboard and the screen, in order to make it completely autonomous.

Normally the Arduino Due is *not* autonomous: it requires an external computer to be used. The usual process is the following. Users write their application (*e.g.*, a mobile robot controller) with a text editor in some programming language. They transform it with a program called a compiler into machine code that the Arduino can execute. Finally, they send this machine code with a third program to the Arduino, as a series of 0s and 1s (via a USB cable). All this happens on an existing computer, with already existing programs (operating system, text editor, compiler, etc).

Since our goal in this book is to program a toy computer *from scratch*, we should in theory avoid using any existing computer, already programmed. Otherwise we would need to show how this existing computer was programmed in the first place (the answer is from yet another already programmed computer – and so on). To solve this chicken and egg problem, we should instead send 0s and 1s "manually" to the Arduino. Doing this completely by hand, with a switch, is not possible because the Arduino expects to receive these 0s and 1s by groups of 8 bits, each group being transmitted at 115,000 bits per second (*i.e.*, in about $70\mu s$). Of course, no one can operate a switch at this speed. Instead, we could build a small digital circuit, connected to a keyboard, which would send a specific group of 8 bits at 115,200 bits per second each time a key is pressed (for instance the group 01000000_2 for 'A', 01000001_2 for 'B', etc). This circuit would not need to be programmed, which would avoid the above chicken and egg problem. However, doing this would be very impractical and error-prone (typos would be hard to detect without a visual feedback, *i.e.*, some kind of display).

In this part, we therefore use an external computer to program the Arduino. However, we try to use it in a minimal way to show convincingly that avoiding its use altogether would be possible. In particular, instead of using a text editor and a compiler to produce the 0s and 1s from a program written in some programming language, we compute these bits manually. The only program we use on the external computer is the one used to send these bits to the Arduino, at the expected speed. Moreover, we use this method only to install a small initial program on the Arduino, namely a very basic input and output system. Its goal is to read other programs (still

in binary form at first) input on a keyboard connected to the Arduino, to output them on a screen also connected to the Arduino, and to execute them. In other words, its goal is to make our toy computer completely autonomous, *i.e.*, to avoid any further need of an external computer (including in Parts 3 and 4).

In order to do this, our basic input output system has the following components:

- a keyboard driver. Connecting a keyboard to the Arduino does not "just work". A small program is needed on the Arduino to decode the signals sent by the keyboard and to interpret them as characters. This small program is called a keyboard driver.
- a "graphics card" driver. Similarly, connecting a display to the Arduino does not "just work". For the display we want to use, things are even worse: we can't even connect the display directly to the Arduino because it does not have the necessary connector. To solve this we use an intermediate board, which we call the graphics card. It can connect to the display on one end, and to the Arduino on the other. But here again, a small program is needed on the Arduino to send the correct signals to the graphics card, in order to display the desired characters on the screen. This small program is the graphics card driver.
- a memory editor. This extremely basic editor uses the above drivers to display the memory's content on the screen (in hexadecimal format), and to allow the user to edit it, with the keyboard. It can also execute a program at a given location in memory. These features allow users to store programs in memory and to run them.
- a virtual machine. Even if the above programs are small, they would still require hundreds of machine code instructions, which use a complicated binary format. Writing all these instructions manually is possible but painful and error prone. In order to simplify this a bit, we use a tiny *virtual machine*. This program simulates a virtual microprocessor using very simple instructions called *bytecode instructions*. This small program must be written in Arduino Due's machine code but, once this is done, all other programs can be written in simpler bytecode instructions (simulated by the virtual machine). This is what we do for the above drivers and the memory editor.

The rest of this part presents the hardware components of our toy computer, shows how they are assembled, and explains how our input output system is built and installed on the Arduino. It is organized as follows:

- Chapter 6 gives an overview of the Arduino Due board and of its main chip. We use this at the end to control a LED with commands sent "manually" to the Arduino.
- Chapter 7 gives an overview of the microprocessor inside the Arduino's main chip. It presents a subset of its registers and instructions. We use this at the end to blink a LED, with a first program made of a few machine code instructions.
- Chapter 8 builds on this to implement our toy virtual machine, written in Arduino's machine code, and to store it in flash memory.
- Chapter 9 presents the clock used by the Arduino, and explains how to change its frequency. We then use this knowledge to implement a small program, using bytecode instructions, to set the clock to its maximum frequency.

- Chapter 10 presents how Liquid Crystal Displays work in general, how our specific Thin Film Transistor display works, what the graphics card is doing, how to communicate with it from the Arduino, and how to use it. It then provides the graphics card driver implementation. We test it at the end to display the traditional "Hello, World!" message.
- Chapter 11 presents how keyboards work in general, and PS/2 keyboards in particular. It then provides a small keyboard driver. We test it at the end with a small "echo" program, which simply displays on screen each key typed on the keyboard.
- Chapter 12 uses all the previous components to implement a basic memory editor, finally making our toy computer completely autonomous.

CHAPTER

6 First Steps with the Arduino Due

This chapter gives a short overview of the Arduino Due board. We use this at the end to control a LED with commands sent "manually" to the Arduino.

But first, why do we use an Arduino Due? The short answer is that it is well suited for our purpose. A longer answer is the following. First of all, desktop and laptop computers already include a Basic Input Output System (the BIOS), using the keyboard and the screen. We do not have to use it, but since our goal is to program a toy computer from scratch, it is better if we really have to implement a BIOS.

Single board computers, such as Raspberry Pi boards, are complete computers on a single printed circuit board. They can run usual operating systems such as Windows or Linux. They generally do not have a BIOS but, at least for Raspberry Pi boards, they have a complex boot process which makes it hard to start completely from scratch, with a few hand-written machine code instructions.

Single board *microcontrollers* are even simpler than single board computers. As their name suggest, they provide two things:

- a microcontroller. This is a single chip which contains everything we need for a toy computer: a microprocessor, volatile and non-volatile memory, and dedicated circuits to control external devices in a simple way.
- a printed circuit board on which the microcontroller is soldered. A microcontroller alone is just a chip, which would be hard to use directly. Its pins are very small, yet they need to be connected to a power source, to external devices which have "large" connectors, etc. The board makes this easy to do.

Single board microcontrollers do not have a BIOS and generally have a simpler boot process than single board computers. We therefore chose such a board to build our toy computer. More specifically, we chose the Arduino Due board because it is easy to use completely from scratch. And also because Arduino is a very popular platform for makers. In this way, after having built your toy computer with it, you will have everything you might need to reuse the Arduino Due for other purposes.

6.1 Overview of the Arduino Due

The Arduino Due board is based on the SAM3X8E microcontroller from Atmel, which is presented in the next section. This microcontroller is soldered on a printed circuit

board (see Figure 6.1), which connects its 144 pins – 36 on each side of the chip – to several other components on this board, including:

- 86 female header pins on 3 sides of the board. These pins provide a convenient way to connect wires to the otherwise very tiny pins of the microcontroller itself. Each board pin has a number and/or a name printed next to it (some of them are shown, in white, in Figure 6.1). Most of these board pins are directly connected to the SAM3X8E pins, whose names are shown in yellow in the figure. For instance, pin 19 is connected to the PA10 pin of the microcontroller, pin 13 is connected to the PB27 pin, etc. Likewise, the "GND" (ground) and "3.3V" pins are connected to the USB ports, which provide power, and to the microcontroller. They can be used to power external devices which require a 3.3V power source. As an exception, the "5V" pin is *not* connected to the microcontroller. It can be used to power external devices which require a 5V power source, such as PS/2 keyboards. *All the other pins of the board output 3.3V, and only support up to 3.3V inputs*.
- A few male header pins, including 6 very close to the microcontroller, labeled "SPI". These male pins are also directly connected to corresponding pins of the microcontroller, including the PA25, PA26 and PA27 pins (see Figure 6.1).
- A few built-in LEDs. A green one, labeled "ON" indicates when the card is powered. The one next to it, orange and labeled "L", is connected to the PB27 pin of the microcontroller. It can be turned on and off by programs running on the microcontroller.
- Two micro-USB ports. They can be used to power the card. At the same time, they can also be used by the microcontroller's *boot assistant*, a program which can read and execute other programs sent by an external computer. Finally, the "native" port can be used to plug USB devices, for instance a keyboard, to be used by applications running on the microcontroller. However, doing so requires a USB driver program, and writing one from scratch is not a trivial task.
- A "RESET" button, in a corner of the board. Pushing this button is equivalent to turning off and on again the whole card. In particular, it erases the content of the volatile memory, and restarts the microcontroller from a well defined initial state.
- An "ERASE" button. Pushing this button while the card is powered erases the content of the *non-volatile* memory. This is equivalent to erasing the content of the hard drive of a desktop or laptop computer.

The components and connections of the Arduino Due which are used in the rest of this book are shown in Figure 6.1. If you want to know more, the full list can be found on the Arduino Due web page at https://store.arduino.cc/products/arduino-due. This page links to specifications of the full mapping between the board pin names and the microcontroller pin names [3], and of all the board components and connections [11].

At this stage we can do our first tests with the Arduino. First, connect one of its USB ports to your computer with a USB cable, or directly to an outlet with a USB phone charger. The "ON" and "L" LEDs should be on. As shown in Figure 6.1, the "L" LED is connected on one end to pin PB27, which is also connected to pin 13. The other end is connected to the ground. With a male-male wire, connect the "GND" and

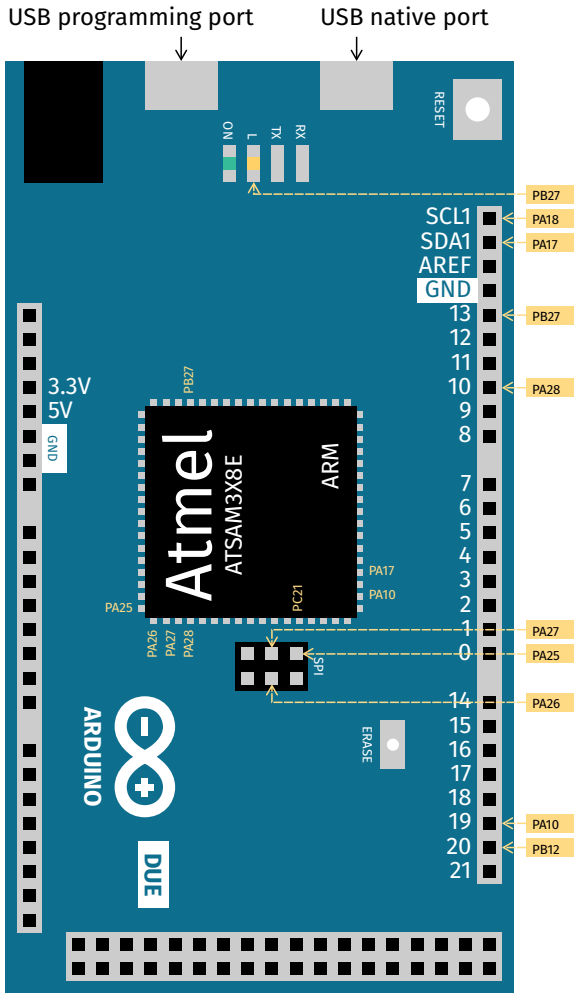


FIGURE 6.1 The components of the Arduino Due board [3, 11] used in this book.

13 pins. What happens? The LED turns off because it is no longer powered. Finally, remove the wire and unplug the Arduino.

6.2 Overview of the Atmel SAM3X8E

As indicated above, the Arduino Due is based on the Atmel SAM3X8E [8]. This section gives a brief overview of the main internal components of this microcontroller, ignoring those we don't need in our toy computer. The next sections and chapters give more details about each of them. These components are the following (see Figure 6.2):

- an ARM Cortex-M3 microprocessor, which has a 32-bit architecture and can execute up to 84 million instructions per second. It contains sub components such as a System Timer (SysTick) and a Memory Protection Unit (MPU). See Section 7.1.
- 512 KB of non-volatile flash memory, split in two memory banks of 256 KB, each with its own controller (to erase data, *flash* new data, etc). See Section 6.5.
- 16 KB of ROM, providing a boot assistant. As explained earlier, this program can read and execute other programs sent by an external computer. See Section 6.4.
- 96 KB of volatile RAM, split in one bank of 64 KB and one bank of 32 KB.
- 4 Parallel Input Output (PIO) controllers, named A, B, C and D. Each controller manages up to 32 pins of the microcontroller. For instance, controller A manages the 30 pins named PA0 to PA29, while controller D manages only 10 pins named PD0 to PD9. Each pin can be configured as an input or as an output. See Section 6.6.
- Many *peripheral controllers* to communicate with a large variety of external devices. These include a Serial Peripheral Interface (SPI) controller to interact with devices supporting the SPI protocol (see Section 10.3), and a Universal Synchronous Asynchronous Receiver Transmitter (USART) for external devices using a serial connection (*i.e.*, transmitting one bit at a time – see Section 11.2). There is also a Pulse Width Modulation (PWM) controller (which can be used to control motors, loud speakers, etc), an Analog to Digital converter, etc. The peripheral controllers do not interact with the microcontroller pins directly, but instead go through the PIO controllers (see Figure 6.2).
- A Power Management Controller (PMC), connected to an external crystal oscillator. This component provides clock signals (in green in Figure 6.2) to the other components, including the microprocessor and the peripheral controllers. It can generate clock signals at various frequencies. See Section 9.1.
- A Reset Controller (RSTC), which resets all the components to their initial state when the RESET button, connected to the "NRST" pin of the microcontroller, is pushed on the Arduino board.
- A Watchdog Timer (WDT), which also resets all the components to their initial state (via the Reset Controller) when the timer expires. To avoid being reset, the program running on the microcontroller must periodically reset this timer (or disable it).
- A memory bus interconnecting all these components, thus allowing the microprocessor to use them. See Section 6.3.

The SAM3X8E microcontroller contains many other components, not shown in

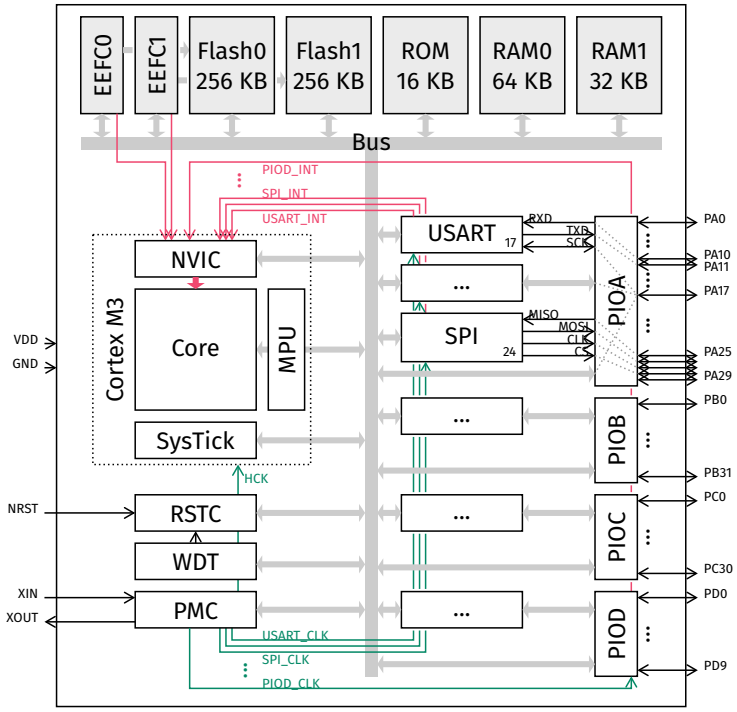


FIGURE 6.2 The components of the SAM3X8E microcontroller [8] used in this book.

Figure 6.2. A more complete diagram can be found in Figure 2-3 at page 6 of the SAM3X/3A Series Datasheet [8]. This very long reference manual (1459 pages!) gives an exhaustive description of all the components, except the microprocessor itself (described in other reference manuals, including the 858 pages long Armv7-M Architecture Reference Manual [17]). The next sections and chapters give a summary of the parts of these manuals that we need to program our toy computer.

6.3 Memory bus

Each component inside the microcontroller, except the memory banks, contains one or more 32-bit "registers" which allow the microprocessor to interact with it. For instance, the Parallel Input Output A controller has a 32-bit register whose bits reflect the input voltage of the pins connected to it. For instance, if the pins PA2 and PA5 are at 3.3V and if all the other PA pins are at 0V, then all the bits of this register are 0, except bits 2 and 5 (*i.e.*, the value of the register is $100100_2 = 36$). Another register of this controller allows the microprocessor to set the voltage of the desired pins to 3.3V. For example, writing the value $17 = 10001_2$ in this register, sets pins PA0 and PA4 to 3.3V, and leaves the other pins unchanged.

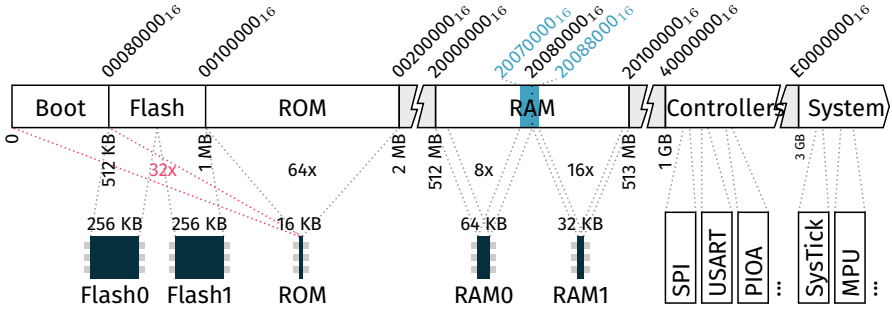


FIGURE 6.3 The memory map after a full erase, showing how memory banks and microcontroller registers (bottom) are mapped to memory addresses (top).

These registers must not be confused with the microprocessor registers. The latter can be used directly by arithmetic instructions, such as "add the value of register number 1 to the value of register number 3 and store the result in register number 4". This is not the case of the former. Instead, each microcontroller register has a predefined memory address, and can only be used by reading or writing values in memory, at this address. For example, the Parallel Input Output A controller registers mentioned above have the addresses $400E0E30_{16}$ and $400E0E3C_{16}$. If the only PA pins at 3.3V are PA2 and PA5, then the value in memory at address $400E0E30_{16}$ is 36. Similarly, writing the value 17 at memory address $400E0E3C_{16}$ sets the pins PA0 and PA4 to 3.3V[1].

The role of the memory bus is to send each memory access request to the component or memory bank which is responsible for it. In the previous example, the bus sends the load request for address $400E0E30_{16}$ to the PIO A controller, because this is the component responsible for this address. It does the same with the store request for address $400E0E3C_{16}$, for the same reason.

In order to do this, the bus must have a "map" of the memory, indicating which component "lives" at which address(es). After a full erase, with the ERASE button, this *memory map* is the one illustrated in Figure 6.3. An important thing to note is that different addresses can map to the same physical location (just as a building at the intersection between two roads could have two addresses, one per road):

- The 16 KB the ROM memory bank are mapped to the addresses in $[0,4000_{16}[$ (4000_{16}=16 KB), but also to the addresses in $[4000_{16},8000_{16}[$, and so on up to $[7C000_{16},80000_{16}[$. In total, this physical memory region is mapped 32 times in the 512 KB "Boot" region in Figure 6.3. It is also mapped 64 more times in the 1 MB "ROM" region, between addresses 100000_{16} and 200000_{16} (excluded).
- The 64 KB of the RAM0 memory bank are mapped 8 times in the first 512 KB of the "RAM" region, between addresses 20000000_{16} and 20080000_{16} (excluded).

[1]Provided a few other registers of this controller are set up correctly, see Section 6.6.

- The 32 KB of the RAM1 memory bank are mapped 16 times in the second 512 KB of the "RAM" region, between addresses 20080000_{16} and 20100000_{16} (excluded).

Another thing to note is that most memory addresses are not mapped to anything. This is the case, for instance, of the 510 MB between the "ROM" and "RAM" regions, and of a 1 GB range before the "System" region (see Figure 6.3). These regions are "reserved" for future versions and should not be used.

The registers of the microcontroller components are mapped in the "Controllers" region, a 1 MB region starting at address 40000000_{16}. This includes the Parallel Input Output controllers, the Serial Peripheral Interface controller, the Power Management Controller, etc. Some details of this region are presented in the next sections and chapters, for the registers we need (a full description of the whole memory map can be found in chapter 7 of [8], which refines the map defined in chapter 4 of [16]).

Finally, as illustrated in Figure 6.3, the registers of the microprocessor's internal components, such as the System Timer and the Memory Protection Unit, are mapped in the last part of the memory, starting at address $E0000000_{16}$ (only a few kilobytes inside this 1 GB region are actually used).

In the following we do not use the whole RAM region, but only its central part, in $[20070000_{16}, 20088000_{16}[$ (see Figure 6.3). This is the only part where the two RAM banks are seen as a contiguous sequence of $96 * 1024$ bytes, without repetition.

6.4 Boot assistant

When it is reset after a full erase, the Arduino Due runs its *boot assistant* program, stored in ROM (like the boot program of Section 5.1.2). As said earlier, this program can read and execute other programs, sent by an external computer. This section explains how. We use this at the end to explore the Arduino's memory.

6.4.1 User interface

The boot assistant, in an endless loop, waits a for request sent by an external computer, executes it, and returns a response. A request is a sequence of characters ending with a sharp (#). The boot assistant supports 12 types of requests. The main ones are the following:

- w*address*,# reads a 32-bit word in memory at *address*, and returns the result (both in hexadecimal). For example, w2008000C,# returns the word at address $2008000C_{16}$.
- W*address*,*value*# writes the 32-bit hexadecimal *value* in memory, at *address*. For example, W2008000C,1234ABCD# writes $1234ABCD_{16}$ at address $2008000C_{16}$.
- h*address*,# and H*address*,*value*# do the same thing, but for 16-bit half words.
- o*address*,# and O*address*,*value*# do the same thing, but for bytes.
- G*address*# runs the program starting at *address*. More precisely, it runs the program starting at the address stored in memory at $address + 4$. This command is explained in more details in Section 7.5.3.

- `V#` returns the boot assistant's version number.

Using these requests, an external computer can send a program to the Arduino, word by word (or byte by byte), and can then run it on the Arduino.

6.4.2 Communication protocol

The above requests must be sent character by character, by using the *ASCII code* [14]. This American Standard Code for Information Interchange assigns a number to each letter. For instance, # is represented with $35 = 00100011_2$. Thus, to send a #, the external computer must send the byte 00100011_2 to the Arduino, one bit at a time, starting with the least significant one. This can be done in 3 different ways:

- via the pin 0 / RX0 on the Arduino board. In this case each byte must be sent at 115,200 *bauds*, *i.e.*, bits per second. Responses from the boot assistant are sent in the same format on pin 1 / TX0. As said in introduction of Part 2, this method could *theoretically* be used to program the Arduino without any external computer.
- via the USB "programming port" (see Figure 6.1). This port is connected, via a small chip, to the above RX0 and TX0 pins. This small chip (an ATmega16U2 microcontroller, with only 512 *bytes* of RAM) converts the USB signals into the above format.
- via the USB "native port" (see Figure 6.1). This port is directly connected to an USB controller component in the SAM3X8E (not shown in Figure 6.2).

6.4.3 Experiments

Lets put what we learned so far into practice. For this you need a small program on your computer to send requests to the Arduino, via a USB cable. Download https://ebruneton.github.io/toypc/scripts.zip, unzip it, and follow the installation instructions in its `README` file. Open a terminal on your computer, hereafter called the "host", and go to the directory containing the unzipped files. Then, connect the Arduino Due's *native* port to your computer with a USB cable. Press the ERASE button for about 2 seconds, then press the RESET button. At this stage the boot assistant is running on the Arduino and is waiting for requests.

In the host terminal, type `python3 boot_helper.py` to connect to the Arduino (for completeness, this script is also given in Appendix E). You should see a prompt `>`. Type `V#` and press Enter to check that everything works fine. You should see the boot assistant's version number (which might differ from the one shown here):

```
user@host:~$ python3 boot_helper.py
>V#
v1.1 Dec 15 2010 19:25:04
```

Lets read some words in the "Boot" region (see Figure 6.3). Type the following requests, and observe the responses:

```
>w10,#
0x001000C7
>w4010,#
0x001000C7
>w8010,#
0x001000C7
```

This shows that the ROM is indeed mapped several times in this region (the `0x` prefix means that the following value is in hexadecimal; it is not part of the value itself). We could also check that it is mapped several times in the "ROM" region too (try reading the values at 100010_{16} or 104010_{16}). A look at the "Flash" region shows that all its bits have been erased to 1s:

```
>w80000,#
0xFFFFFFFF
>w80004,#
0xFFFFFFFF
```

Lets now try to write some words in memory. We can first check that the ROM is really read-only, by trying to write in it:

```
>W10,12345678#
>w10,#
0x001000C7
```

The write has no effect, as expected. Writes in "Flash" have no effect either:

```
>W80000,12345678#
>w80000,#
0xFFFFFFFF
```

In fact it is possible to write in flash memory, but this requires a more complex process, explained in Section 6.5. Lets now try to write some value in the "RAM" region. The first 4 KB of RAM are used by the boot assistant, and we don't want to change them. Lets use the first word after this:

```
>W20071000,12345678#
>w20071000,#
0x12345678
```

It works! Here we could check again that the RAM0 bank is mapped several times in the "RAM" region (try reading the value at 20081000_{16}). A more interesting test is to read the 4 bytes starting at 20071000_{16}:

```
>o20071000,#
0x78
>o20071001,#
0x56
>o20071002,#
0x34
>o20071003,#
0x12
```

We see that the least significant byte of 12345678_{16}, namely 78_{16}, is stored at address 20071000_{16}. The most significant one, 12_{16}, is stored at address 20071003_{16}. This is called the *little endian* order. The opposite – least significant byte at 20071003_{16} and most significant at 20071000_{16} – is called the *big endian* order.

Finally, lets see what happens if we try to read a value in the "reserved" region after the "ROM" region:

```
>w200000,#
ERROR: no response from device.
```

You should see nothing during 5s, then the host program exits with an error message. This is because the boot assistant crashed while trying to read a reserved memory address! In such cases you can restart it by pressing the RESET button on the Arduino. You can then restart the `boot_helper.py` program. To exit it normally, type `exit#` (this command is not sent to the Arduino; instead the host program stops itself).

6.5 Flash controller

Flash memory has the advantage of being non-volatile. Unfortunately, this comes with a cost: the memory must be erased before it can be written, and erasing it is slow (a few milliseconds, whatever the number of bytes erased). It is thus impractical to write into it word by word (this is why, as we saw above, writing values directly in flash memory has no effect). Instead, to amortize this cost, flash memory is divided into *pages* of 256 bytes (or 64 words), and is erased and written one page at a time. This process is controlled by an Enhanced Embedded Flash Controller (EEFC), one per flash memory bank (see Figure 6.2). This section presents these controllers, and shows how they can be used via the boot assistant.

6.5.1 Memory pages

Erasing and writing a page into flash memory requires the following process:

1. Write the 64 words of the page into 64 internal "registers", shared between the two flash controllers. *All the registers must be written to*, even if we only care about some of them (otherwise the next step does not work).
2. Ask the flash memory bank's controller to write the content of these registers into a specific page, which is erased first. The page is identified by its index, from 0 to 1023 (included).
3. Wait a few milliseconds until the flash controller is done writing the registers in memory. During this time the flash memory bank cannot be used, and the above registers must not be modified.

The first step of the above process is simple, because the above 64 internal registers are mapped repeatedly (2048 times) inside the 512 KB "Flash" region. This means that writing a word in this memory region writes a value in one of these registers[2].

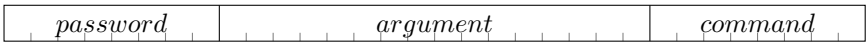
[2] In this region, writing a word at an address which is not a multiple of 4 is not supported. Writing half

Name	Type	EEFC0	EEFC1
Mode Register	Read-Write	$400E0A00_{16}$	$400E0C00_{16}$
Command Register	Write-Only	$400E0A04_{16}$	$400E0C04_{16}$
Status Register	Read-Only	$400E0A08_{16}$	$400E0C08_{16}$

TABLE 6.1 The Enhanced Embedded Flash Controller registers used in this book.

Note that these registers are only used for writes. Reading a value in this region reads the flash memory, not these internal registers.

The second step is done by writing a specific value in the Command Register of the memory bank's controller (see Table 6.1). This 32-bit value must have the following binary form, where $password = 5A_{16}$, $command = 3$ (which means "erase and write page"), and $argument$ is the page index:



The third step is done by reading the Status Register of the memory bank's controller repeatedly, until its least significant bit is 1. Indeed, this bit is 0 while the controller is writing the page into flash memory, and 1 when the process is done.

To illustrate this, lets assume we want to write the fourth page (counting from 0) controlled by the second flash controller, *i.e.*, the 64 words in $[C0400_{16}, C0500_{16}[$. We first need to write the 64 words at these addresses (writing them in $[80000_{16}, 80100_{16}[$, for instance, would work too, since the registers are mapped there too). We must then write the $5A000403_{16}$ command in the EEFC1 Command Register, *i.e.*, at address $400E0C04_{16}$. Finally, we need to wait until bit 0 of the EEFC1 Status Register, at address $400E0C08_{16}$, is 1.

6.5.2 Boot mode

Besides controlling the flash memory banks, the flash controllers also control 3 additional bits of non-volatile memory: bit 0 (security), bit 1 (boot mode selection), and bit 2 (flash selection). In this book we use only the boot mode selection bit. When this bit is 0, which is the case after a full erase, the memory mapped in the "Boot" region $[0, 80000_{16}[$ is the ROM (see Figure 6.3). When it is 1, the memory mapped in this region is the flash memory (see Figure 6.4).

These bits can be modified with the EEFC0 Command Register. With the values $password = 5A_{16}$, $command = B_{16}$, and $argument = n$, bit n is set to 1. With $command = C_{16}$ instead, bit n is reset to 0. Thus, writing $5A00010B_{16}$ at address $400E0A04_{16}$ sets the boot mode selection bit to 1, and writing $5A00010C_{16}$ instead resets it to 0. In both cases, the least significant bit of the Status Register is set to 1 when the operation completes, as for a page write command.

words or bytes is not supported either.

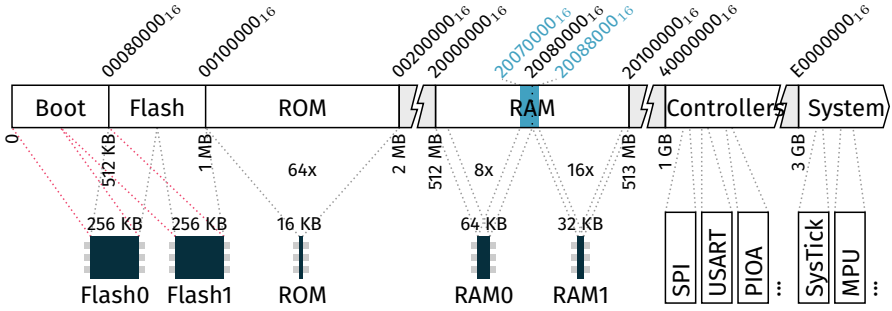


FIGURE 6.4 The memory map when the "boot mode selection" bit is 1, showing how memory banks and microcontroller registers (bottom) are mapped to memory addresses (top).

6.5.3 Experiments

We can now put this new knowledge into practice. First, connect the Arduino to your computer, open a terminal as you did in Section 6.4.3, and start the `boot_helper.py` program:

```
user@host:~$ python3 boot_helper.py
```

Lets then write some values in the fourth page of the second flash memory bank, $[C0400_{16}, C0500_{16}[$, as described in Section 6.5. We first need to write the values in the internal registers of the flash controllers. This is a bit painful because we need to write 64 values (you can "cheat" by using copy paste):

```
>WC0400,1#   >WC0434,14#  >WC0468,27#  >WC049C,40#  >WC04D0,53#
>WC0404,2#   >WC0438,15#  >WC046C,28#  >WC04A0,41#  >WC04D4,54#
>WC0408,3#   >WC043C,16#  >WC0470,29#  >WC04A4,42#  >WC04D8,55#
>WC040C,4#   >WC0440,17#  >WC0474,30#  >WC04A8,43#  >WC04DC,56#
>WC0410,5#   >WC0444,18#  >WC0478,31#  >WC04AC,44#  >WC04E0,57#
>WC0414,6#   >WC0448,19#  >WC047C,32#  >WC04B0,45#  >WC04E4,58#
>WC0418,7#   >WC044C,20#  >WC0480,33#  >WC04B4,46#  >WC04E8,59#
>WC041C,8#   >WC0450,21#  >WC0484,34#  >WC04B8,47#  >WC04EC,60#
>WC0420,9#   >WC0454,22#  >WC0488,35#  >WC04BC,48#  >WC04F0,61#
>WC0424,10#  >WC0458,23#  >WC048C,36#  >WC04C0,49#  >WC04F4,62#
>WC0428,11#  >WC045C,24#  >WC0490,37#  >WC04C4,50#  >WC04F8,63#
>WC042C,12#  >WC0460,25#  >WC0494,38#  >WC04C8,51#  >WC04FC,64#
>WC0430,13#  >WC0464,26#  >WC0498,39#  >WC04CC,52#
```

We then write the "erase and write page" command in the Command Register, and check that it is done with the Status Register:

```
>W400E0C04,5A000403#
>w400E0C08,#
0x00000001
```

Finally, we can check that the operation was successful by reading some values in the

page we just wrote:

```
>wC0408,#
0x00000003
>wC0424,#
0x00000010
>exit#
```

We can also verify that these values are not lost when the Arduino is turned off. For this, unplug the Arduino, plug it again, and read again the value at $C0424_{16}$:

```
user@host:~$ python3 boot_helper.py
>wC0424,#
0x00000010
```

Let us now test the boot mode selection bit. Set this bit to 1 as described above:

```
>W400E0A04,5A00010B#
```

The page we just wrote is now mapped in the "Boot" region as well, offset by 512 KB from the "original" (see Figure 6.4), *i.e.*, in $[40400_{16},40500_{16}[$:

```
>w40408,#
0x00000003
```

Now reset the boot mode selection bit. This maps the ROM again in the "Boot" region:

```
>W400E0A04,5A00010C#
>w40408,#
0xB004F8CA
>exit#
```

Finally, press the Arduino's ERASE button for a few seconds, to erase the page we wrote, and unplug it.

6.6 Parallel Input Output controller

To conclude this chapter, we present here the Parallel Input Output (PIO) controllers. They are an important part of the microcontroller, since all the peripheral controllers use them to access the microcontroller's pins. We use this at the end to turn a LED on and off via the boot assistant.

6.6.1 User interface

The main goal of the PIO controllers is to enable the microcontroller to communicate with external devices. Another important goal is to enable the microprocessor and the peripheral controllers (USART, SPI, etc – see Figure 6.2) to *share* the microcontroller's pins in a principled way (*i.e.*, one at a time). If each peripheral controller was using its own private pins, then the SAM3X8E chip would need much more than 144 pins to connect all of them. Hopefully, users rarely need to use them all at the same time. The pins can thus be used by a subset of the peripherals for some time, then used by another subset, and so on.

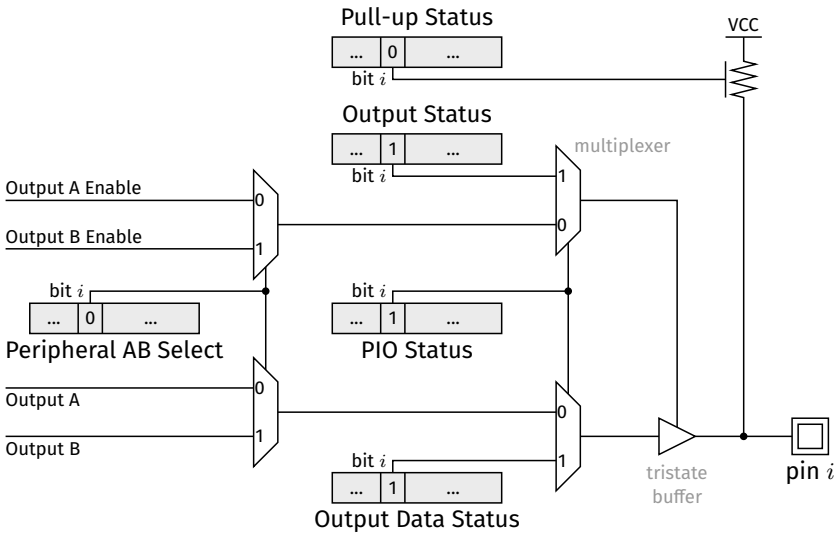


FIGURE 6.5 A simplified view of the circuit and registers (in gray) controlling the output of each pin. The input part is not shown. See Figure 31-3 in [8].

This sharing is done as follows: each peripheral controller uses a fixed set of pins, but a pin can be used by up to two peripherals. For instance, the USART peripheral always uses pins PA10 and PA11 to receive and transmit data – this can't be changed. But PA10 can also be used by the Digital-to-Analog peripheral. Similarly, PA11 can also be used by the Analog-to-Digital peripheral. In addition, all the PIO pins can also be used directly by the microprocessor. This means that each pin can be controlled by up to 3 entities (one at a time): two peripherals, hereafter named A and B[3], and the microprocessor.

To choose the entity controlling a pin at a given time, each PIO controller has the following registers (see Figure 6.5 – here we use the PIO A controller as an example; the others work in the same way):

- The PIO Status Register: if its nth bit is 1, then the PAn pin is controlled by the microprocessor. This register can only be read. Writing to it has no effect. Instead, one must write in the PIO Enable Register (resp. the PIO Disable Register) to set (resp. clear) bits in the Status Register. For instance, writing 101_2 in the Enable Register sets bits 0 and 2 to 1 in the Status Register, and leaves the other bits unchanged. Writing 1000_2 in the Clear Register sets bit 3 to 0 in the Status Register, and leaves the other bits unchanged.
- The Output Status Register: if the PAn pin is controlled by the microprocessor, then the nth bit of this register indicates if this pin is an output (bit equal to 1), or a pure input (bit equal to 0). As for the Status Register, this register can only be read. It

[3]For PA10, peripheral A is the USART, and peripheral B is the Digital-to-Analog converter.

Name	Type	PIO A	PIO B
PIO Enable Register	Write-Only	$400E0E00_{16}$	$400E1000_{16}$
PIO Disable Register	Write-Only	$400E0E04_{16}$	$400E1004_{16}$
PIO Status Register	Read-Only	$400E0E08_{16}$	$400E1008_{16}$
Output Enable Register	Write-Only	$400E0E10_{16}$	$400E1010_{16}$
Output Disable Register	Write-Only	$400E0E14_{16}$	$400E1014_{16}$
Output Status Register	Read-Only	$400E0E18_{16}$	$400E1018_{16}$
Set Output Data Register	Write-Only	$400E0E30_{16}$	$400E1030_{16}$
Clear Output Data Register	Write-Only	$400E0E34_{16}$	$400E1034_{16}$
Output Data Status Register	Read-Only	$400E0E38_{16}$	$400E1038_{16}$
Pull-up Disable Register	Write-Only	$400E0E60_{16}$	$400E1060_{16}$
Pull-up Enable Register	Write-Only	$400E0E64_{16}$	$400E1064_{16}$
Pull-up Status Register	Read-Only	$400E0E68_{16}$	$400E1068_{16}$
Peripheral AB Select Register	Read-Write	$400E0E70_{16}$	$400E1070_{16}$

TABLE 6.2 The Parallel Input Output registers used in this book.

can only be changed via the Output Enable Register or the Output Disable Register.

- The Output Data Status Register: if the PA*n* pin is controlled by the microprocessor, and if it is configured as an output, then the *n*th bit of this register defines the pin's output value. This register can only be changed via the Set Output Data Register or the Clear Output Data Register.
- The Peripheral AB Select Register: if the PA*n* pin is *not* controlled by the microprocessor, then the *n*th bit of this register indicates whether this pin is controlled by peripheral A (bit equal to 0) or B (bit equal to 1). Unlike the above registers, this register can be read *and* written. If a pin is controlled by a peripheral, this peripheral decides whether it should be used as an output or not and, if yes, which value to output.

In addition to this, *i.e.*, independently of which entity controls it, each pin can optionally be connected to the 3.3V power source via a *pull-up* resistor (see Figure 6.5). When a pin is configured as an input but is not connected to an external device, this resistor "pulls" the pin voltage up to 3.3V. Without it, a disconnected pin could be at any voltage, depending on external interferences. Note that setting a pin's output value to 0 with the above registers sets the pin voltage to 0V (if it is configured as an output), even if the pull-up resistor is enabled. The Pull-up Status Register indicates which pins have their pull-up *disabled*: if its *n*th bit is 1, then the pull-up is disabled. This register is read-only. It can only be modified by writing into the Pull-up Disable Register or the Pull-up Enable Register.

A simplified representation of a small part of the digital circuit controlling each pin is represented in Figure 6.5. The addresses of some of the registers of each PIO controller are given in Table 6.2. The full circuit and the complete list of registers can

be found in Chapter 31 of [8].

6.6.2 Experiments

Lets put this new knowledge into practice. We know that the "L" LED on the board is connected to pin PB27 (see Figure 6.1). We have seen above how to control a pin by writing values in memory. And we have already used the boot assistant to write values in memory. Thus, we should be able to control the LED via the boot assistant. For this, connect the Arduino to your computer and open a terminal as you did in Section 6.4.3. Lets first check which entity controls the PB27 pin, by reading the PIO Status Register:

```
user@host:~$ python3 boot_helper.py
>w400E1008,#
0x0FFFFFFF
```

We see that bit 27 is 1, meaning that the pin is controlled by the microprocessor. Lets check the Output Status Register and the Output Data Status Register:

```
>w400E1018,#
0x00000000
>w400E1038,#
0x00000000
```

They are both 0, meaning that the pin is not used as output, *i.e.*, it is disconnected from the PIO controller circuit. Yet the LED is on. This probably means that the pull-up resistor is enabled. We can check this by reading the Pull-up Status Register:

```
>w400E1068,#
0x00000000
```

All bits are 0, in particular bit 27, meaning that the pull-up is not *disabled*, *i.e.*, it is enabled as we suspected. Let us now configure the pin as an output, by setting the 27^{th} bit ($= 2^{27} = 800000_{16}$) to 1 in the Output Status Register. As said above, this must be done by using the Output Enable Register:

```
>W400E1010,8000000#
```

At this point you should see that the LED is off! Indeed we just configured the pin as an output, and we saw above that the output value in the Output Data Status Register was 0. The pin has thus been connected to the ground, as we did with a wire at the end of Section 6.1. We can turn the LED on again by setting the pin's output value to 1, with the Set Output Data Register:

```
>W400E1030,8000000#
```

You can now `exit#` and unplug the Arduino.

CHAPTER

7 First Steps with the Cortex M3

This chapter gives a short overview of the Arduino Due microprocessor, the ARM Cortex M3. This knowledge is necessary to go beyond our first steps with the Arduino, in the previous chapter. We use it at the end to write our first program, to blink a LED.

7.1 Overview of the Cortex M3

The Cortex M3 microprocessor has a core instruction processing part and a few other internal components (see Figure 6.2). The instruction processing part loads instructions from memory, decodes them and executes them, in an endless loop. These instructions form the Cortex M3 *machine language*. They are encoded in 16 or 32 bits, and must always start at even addresses. They can be divided in 3 main categories, namely *data processing*, *load and store*, and *conditional and jump* instructions:

- The data processing instructions perform arithmetic and logic operations (addition, multiplication, bitwise and, etc). They can only operate on values stored in registers (or encoded in the instruction itself), and can only store the result in registers. The Cortex M3 has 16 32-bit registers available for this purpose, named R0 to R15. An example data processing instruction is "add the values in R1 and R2, and store the result in R5".
- The load and store instructions load values from memory into registers, and store values from registers in memory. An example load instruction is "load the value from memory at the address in R3, offset by 10, and put the result in R0". If R3 contains 60, this instruction loads the value at address 70 and store it in R0.
- The conditional and jump instructions can modify the normal flow of execution. Instructions are normally executed sequentially, in increasing address order. In other words, in the normal case, after the instruction i at address a is executed, the instruction at address $a + 2$ (or $a + 4$ if i is a 32-bit instruction) is executed. This can be changed with conditional instructions or jump instructions. As their name suggest, conditional instructions are skipped if some condition is met. And jump instructions cause the execution flow to *jump*, *i.e.*, to continue at an arbitrary address.

The other components of the Cortex M3 include a timer, an interrupt controller, and a memory protection unit (they are presented in more details later in this book):

- The timer component, called SysTick, can be used to measure time. It decrements a 24 bit counter by 1 at each clock cycle, and restarts it to a configurable value when it reaches 0.
- The interrupt controller, called the Nested Vector Interrupt Controller (NVIC), provides another way to modify the normal execution flow, other than the conditional and jump instructions. It handles errors, also called *exceptions*, such as trying to read the memory at a reserved address. It also handles external events, also called *interrupts*, such as the reception of some data on an input pin. When an error or event occurs, the NVIC causes the execution flow to jump to a predefined *handler* address, associated with each error or event source.
- The memory protection unit can be used to control memory accesses. It can divide the memory into regions, and can associate different access rights with each region. For instance, one region can be made inaccessible, another read-only, etc. This can be used to protect programs from each other, *i.e.*, to avoid a crash in a program to crash another one, or to prevent a program from reading sensitive data (*e.g.*, passwords) in another program's memory (see Chapter 26).

7.2 Registers

As said above, the Cortex M3 instructions can use 16 registers named R0 to R15. The first 13 registers can be used for any purpose, but the last 3 have specific usages. They are called the Program Counter, Link Register and Stack Pointer, and their main goal is to make it easier to split programs into smaller building blocks called *subroutines*. This section presents these registers and how subroutines work.

7.2.1 Program Counter

The R15 register is also called the *Program Counter* (PC). This register contains the address of the currently executing instruction. More precisely, just before executing an instruction at address a, it contains the value a. However, *during the execution of an instruction at address a, the PC contains the value $a + 4$.*

Writing into the PC is also possible, and causes a jump. Since instructions must start at even addresses, the least significant bit of the jump address is always ignored. For instance, attempting to write 17 into the PC actually writes 16, and jumps to address 16. In fact, due to historical reasons[1], some instructions *require* writing $a + 1$ into the PC in order to jump to the instruction at address a. $a + 1$ is called the instruction's *interworking address*.

7.2.2 Link Register

The R14 register is also called the *Link Register* (LR). Some jump instructions, called *branch with link*, set this register to the *interworking* address of the next instruction

[1]Some ARM processors support two instruction sets. They use this bit to specify the instruction set to use after the jump. The Cortex M3 supports only one instruction set but still uses this mechanism.

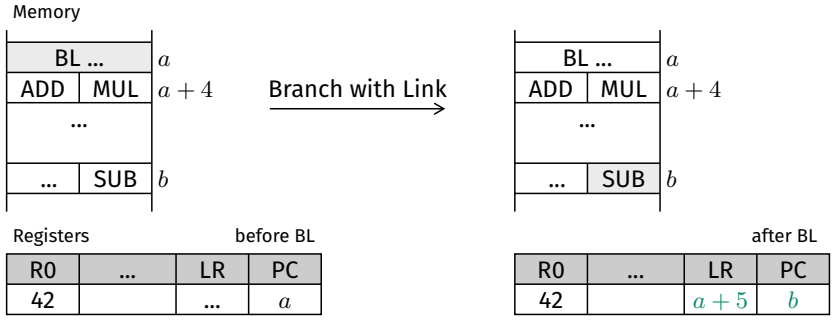


FIGURE 7.1 The Link Register. Branch with Link (BL) instructions set the LR to the interworking address of the next instruction (here MUL) when jumping to another instruction (here SUB). Copying the LR into the PC "resumes" execution (here at MUL).

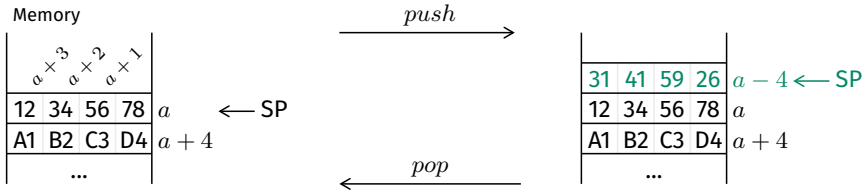


FIGURE 7.2 The Stack Pointer. When the SP contains the value a, pushing a 32-bit value, here 31415926_{16}, stores it at address $a-4$ and updates the SP to $a-4$. Popping gets the value stored at the address given by the SP, and adds 4 to the SP.

in sequential order. For instance, a 32-bit branch with link instruction at address a, jumping to address b, sets the LR to $a+5$. This allows the code starting at b, when its task is done, to "resume" the execution of the initial code sequence, *i.e.*, to jump to the instruction at $a+4$. For this, it just needs to write the value stored in the LR into the PC (see Figure 7.1).

7.2.3 Stack Pointer

The R13 register is also called the *Stack Pointer* (SP). A *stack* is similar to a pile of plates: one can only add, or *push*, a value (a plate) on top of the stack (the pile), not inside it. Similarly, one can only remove, or *pop*, a value from the top of the stack. A *pointer* is a register or memory location which contains the address of some value, called the pointer target. The Stack Pointer contains the address of the top stack value. It is used by two instructions called PUSH and POP, which can push the values in some registers into the stack, and pop values from the stack into registers. This stack is called *descending*[2] because pushing a new value *decreases* the Stack Pointer's value. For instance, if the SP contains the value a, pushing x means storing x in memory at

[2]Note that it may *look* ascending, if addresses are representing in ascending order from top to bottom.

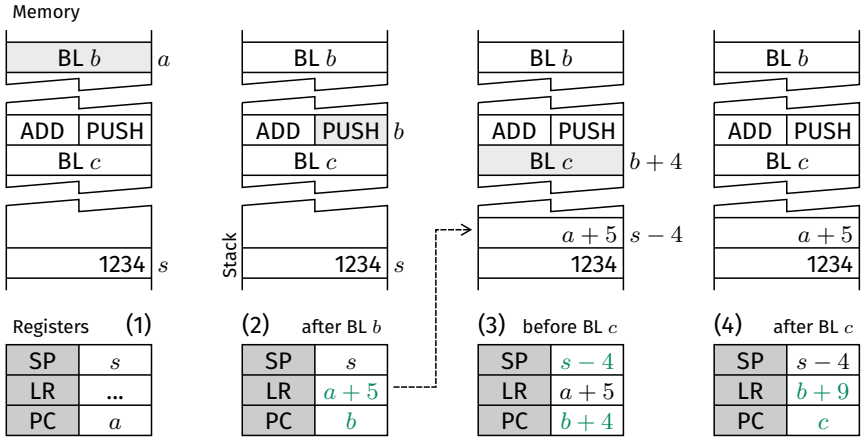


FIGURE 7.3 Nested subroutine calls. Before doing a branch with link to c (3), subroutine B pushes the LR (*i.e.*, the interworking return address $a + 5$ in A) onto the stack (2).

address $a - 4$, and updating the SP value to $a - 4$ (see Figure 7.2).

Note: from now on we use register names to designate either the register itself, or the value it contains, depending on the context. For instance, in "add R1 and R2 and store the result in R3", R1 and R2 designate the values stored in these registers, while R3 designates the register itself.

7.2.4 Subroutines

The main goal of the above registers is to organize programs into smaller subprograms of increasingly higher abstraction levels, in a similar way as digital circuits, or even living organisms, are organized (made of atoms, grouped in molecules, grouped in cells, grouped in organs, grouped into organisms). For instance, a program to draw figures could have a subprogram to draw text, which could itself use a "sub" subprogram to draw characters. The advantage of this method is that each level can be understood and programmed without knowing how the lower levels work internally.

At the machine code level considered here, these subprograms are called *subroutines*. A subroutine is a list of instructions, some of which can *call* other subroutines. After it has been executed, a subroutine *returns* to the subroutine which called it. A call from an instruction at address a in subroutine A (*e.g.*, "draw figure"), to subroutine B (*e.g.*, "draw text"), is a jump to B's first instruction, preceded by instructions allowing B to return in subroutine A. This includes storing the (interworking) *return address* $a + 5$ in the Link Register, as described above. However, this is generally not sufficient. Consider the case where subroutine B needs to call a third subroutine C (*e.g.*, "draw character"), from address $b + 4$ (see Figure 7.3). It cannot simply store its own (interworking) return address $b + 9$ in the Link Register, otherwise it would loose

the value $a + 5$ already stored there, and would no longer be able to return to A! To solve this, B needs to save the current LR value first. The solution is to push it on the stack. This is generally done at the very beginning of each subroutine (see Figure 7.3). Returning to the *caller*, *i.e.*, to A, can then be done by popping this value from the stack, into the PC.

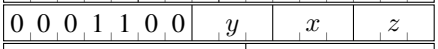
7.3 Instruction set

The Cortex M3 machine language has about 100 instructions. However, some instructions have several variants and several encodings, on 16 or 32 bits. For instance, the ADD instruction has 4 variants and 14 encodings. In total, there are more than 350 different encodings. Here we present only the main instructions and encodings used in this book (about 40 – a few more are presented later), and we give only a short overview for each. All the details about all the instructions can be found in [17].

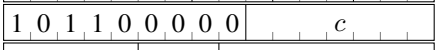
7.3.1 Data processing instructions

ADD $\mathrm{R}z \leftarrow \mathrm{R}z + c$

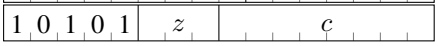
ADD $\mathrm{R}z \leftarrow \mathrm{R}x + \mathrm{R}y$



ADD $\mathrm{SP} \leftarrow \mathrm{SP} + 4 * c$

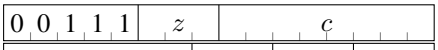


ADD $\mathrm{R}z \leftarrow \mathrm{SP} + 4 * c$

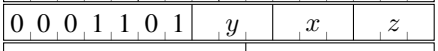


The ADD instruction adds two registers, or a register and a constant, depending on the variant used, and stores the result in a register. For instance, the first variant, $\mathrm{R}z \leftarrow \mathrm{R}z + c$, adds the constant c to register $\mathrm{R}z$ and stores the result in $\mathrm{R}z$. It is encoded on 16 bits. The most significant ones are fixed to 00110_2. The next 3 bits define which register is used, from R0 to R7. The last 8 bits define the constant to add to this register. For instance, by replacing z with $11_2 = 3$ and c with $101101_2 = 45$ we get the instruction $\mathrm{R3} \leftarrow \mathrm{R3} + 45$. Its encoding is obtained by replacing z and c with their values in the encoding schema, yielding $0011001100101101_2 = 332\mathrm{D}_{16}$.

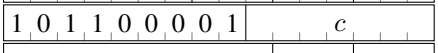
SUB $\mathrm{R}z \leftarrow \mathrm{R}z - c$



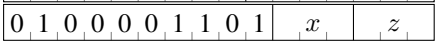
SUB $\mathrm{R}z \leftarrow \mathrm{R}x - \mathrm{R}y$



SUB $\mathrm{SP} \leftarrow \mathrm{SP} - 4 * c$

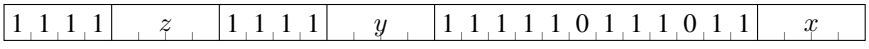


MUL $\mathrm{R}z \leftarrow \mathrm{R}z * \mathrm{R}x$



The SUB and MUL instructions perform subtractions and multiplications. Note that additions, subtractions, and multiplications are done modulo 2^{32}, and can thus overflow (see Section 4.2.3).

UDIV $\mathrm{R}z \leftarrow \mathrm{R}x \,/\, \mathrm{R}y$



The UDIV instruction, encoded on 32-bit, performs integer divisions, such as $\lfloor 14/4 \rfloor = 3$. Rx, Ry, and Rz must not be the SP or the PC.

AND $\mathrm{R}z \leftarrow \mathrm{R}z \wedge \mathrm{R}x$
ORR $\mathrm{R}z \leftarrow \mathrm{R}z \vee \mathrm{R}x$

0 1 0 0 0 0 0 0 0 0	x	z
0 1 0 0 0 0 1 1 0 0	x	z

The AND instruction performs bitwise AND operations, such as $1010_2 \wedge 1100_2 = 1000_2$. Similarly, the ORR instruction performs bitwise OR operations, such as $1010_2 \vee 1100_2 = 1110_2$.

LSL $\mathrm{R}z \leftarrow \mathrm{R}x \ll c$
LSL $\mathrm{R}z \leftarrow \mathrm{R}z \ll (\mathrm{R}x \ mod \ 32)$

0 0 0 0 0	c	x	z
0 1 0 0 0 0 0 0 1 0	x	z	

The Logical Shift Left (LSL) instruction shifts the 32 bits of a register to the left by a certain amount. For instance, shifting the value 1011101_2 to the left by 3 inserts 3 zeros on the right (and drops the 3 most significant bits), yielding 1011101000_2. Shifting to the left by n bits is equivalent to multiplying by 2^n (modulo 2^{32}).

LSR $\mathrm{R}z \leftarrow \mathrm{R}x \gg c$
LSR $\mathrm{R}z \leftarrow \mathrm{R}z \gg (\mathrm{R}x \ mod \ 32)$

0 0 0 0 1	c	x	z
0 1 0 0 0 0 0 0 1 1	x	z	

The Logical Shift Right (LSR) instruction shifts the 32 bits of a register to the right by a certain amount. For instance, shifting the value 1011101_2 to the right by 3 drops the 3 least significant bits, yielding 1011_2. Shifting to the right by n bits is equivalent to dividing by 2^n.

MOV $\mathrm{R}z \leftarrow c$
MOV $\mathrm{R}z_1{:}z_0 \leftarrow \mathrm{R}x$

0 0 1 0 0	z	c	
0 1 0 0 0 1 1 0	z_1	x	z_0

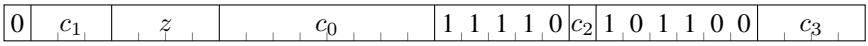
The Move (MOV) instructions "move" (or more precisely copy) a value from a register or a constant into a register. The $z_1{:}z_0$ notation above denotes a concatenation of binary numbers. For instance, replacing z_1 with 1_2 and z_0 with 011_2 gives 1011_2, and thus $\mathrm{R}z_1{:}z_0 = \mathrm{R}11$. The MOV: $\mathrm{R}z_1{:}z_0 \leftarrow \mathrm{R}x$ instruction can thus access the 16 registers R0 to R15.

MOVW $\mathrm{R}z \leftarrow c_3{:}c_2{:}c_1{:}c_0$

0	c_1	z	c_0	1 1 1 1 0	c_2	1 0 0 1 0 0	c_3

The Move Wide (MOVW) instruction copies a 16-bit constant into a register, which must not be the SP or the PC. It can copy values up to 65535, whereas the MOV instruction is restricted to values up to 255.

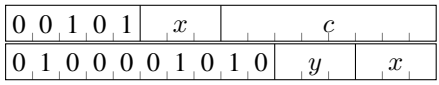
MOVT $\mathrm{R}z[31..16] \leftarrow c_3{:}c_2{:}c_1{:}c_0$



The Move Top (MOVT) instruction copies a 16-bit constant into the 16 most significant bits of a register (which must not be the SP or the PC), leaving the others unchanged. Together with MOVW, it can be used to load a 32-bit constant in a register. For this, first load the least significant bits with MOVW (which erases the most significant bits), then load the most significant bits with MOVT.

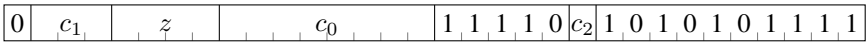
CMP $compare(\mathrm{R}x, c)$

CMP $compare(\mathrm{R}x, \mathrm{R}y)$



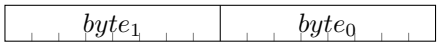
The Compare (CMP) instructions compare two registers, or a register and a constant. They store the comparison result, *e.g.*, whether $\mathrm{R}x$ is less than, equal to, or greater than $\mathrm{R}y$ in a special register, different from the R0-R15 ones. This special register, similar to the Carry register in Section 3.4, is used by conditional instructions[3] (see Section 7.3.4).

ADR $\mathrm{R}z \leftarrow \lfloor \mathrm{PC} \rfloor_4 - c_2{:}c_1{:}c_0$

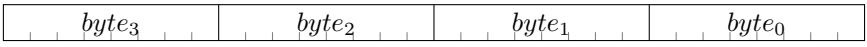


The Address To Register (ADR) instruction subtracts a constant from the PC and stores the result in a register (which must not be the SP or the PC). More precisely, it subtracts a constant from the largest multiple of 4 which is less than or equal to the PC, noted $\lfloor \mathrm{PC} \rfloor_4$. For instance, if the PC value is $102 = 4 * 25 + 2$, then $\lfloor \mathrm{PC} \rfloor_4 = 100$. If the PC value is 100, then $\lfloor \mathrm{PC} \rfloor_4 = 100$. Do not forget also that when an instruction executes, the PC value is the address of this instruction plus 4. Hence, an ADR instruction at address a subtracts a constant from $\lfloor a + 4 \rfloor_4 = \lfloor a \rfloor_4 + 4$.

Incidentally, this raises the question of how 16 and 32-bit instructions are stored in memory. We saw in Section 6.4.3 that 32-bit values are stored in little endian order. In fact everything is stored in this order, including instructions. When a 16-bit instruction of the form



is stored at address a, then $byte_0$ is stored at address a, and $byte_1$ at address $a + 1$. Similarly, when a 32-bit instruction of the form



is stored at address a, then $byte_0$ is stored at address a, $byte_1$ at address $a + 1$, $byte_2$ at address $a + 2$, and $byte_3$ at address $a + 3$[4].

[3] Other instructions can store results in this special register. For simplicity, we don't use this feature.

[4] These bytes are displayed in a different order in [17], see Figure A3-5, pA3-68.

7.3.2 Load and store instructions

LDR	$\mathrm{R}z \leftarrow \mathrm{mem32}[\mathrm{R}x + 4 * c]$	0 1 1 0 1 | c | x | z
LDR	$\mathrm{R}z \leftarrow \mathrm{mem32}[\mathrm{SP} + 4 * c]$	1 0 0 1 1 | z | c
LDR	$\mathrm{R}z \leftarrow \mathrm{mem32}[\lfloor \mathrm{PC} \rfloor_4 + 4 * c]$	0 1 0 0 1 | z | c
LDRH	$\mathrm{R}z \leftarrow \mathrm{mem16}[\mathrm{R}x + 2 * c]$	1 0 0 0 1 | c | x | z
LDRB	$\mathrm{R}z \leftarrow \mathrm{mem8}[\mathrm{R}x + c]$	0 1 1 1 1 | c | x | z

The Load Register (LDR*) instructions load a value from memory and store it in a register. There are 3 variants, LDR, LDRH(alf), and LDRB(yte), to load 32-bit, 16-bit, and 8-bit values from memory. The address from which the value must be loaded can be in one of the R0-R7 registers, in the SP, or in the PC (in this case the memory address actually used is $\lfloor \mathrm{PC} \rfloor_4 = \lfloor a + 4 \rfloor_4$, where a is the instruction's address). In all cases, a constant offset c can be added to this address.

STR	$\mathrm{R}z \rightarrow \mathrm{mem32}[\mathrm{R}x + 4 * c]$	0 1 1 0 0 | c | x | z
STR	$\mathrm{R}z \rightarrow \mathrm{mem32}[\mathrm{SP} + 4 * c]$	1 0 0 1 0 | z | c
STRH	$\mathrm{R}z \rightarrow \mathrm{mem16}[\mathrm{R}x + 2 * c]$	1 0 0 0 0 | c | x | z
STRB	$\mathrm{R}z \rightarrow \mathrm{mem8}[\mathrm{R}x + c]$	0 1 1 1 0 | c | x | z

The Store Register (STR*) instructions read a value in a register and store it in memory. There are 3 variants, STR, STRH(alf), and STRB(yte), to store the 32-bits of the register's value, only its 16 least significant bits, or only its 8 least significant bits. The address at which these bits must be stored can be in one of the R0-R7 registers, or in the SP. In all cases, a constant offset c can be added to this address.

PUSH	$registers, l \rightarrow \mathrm{stack}$	1 0 1 1 0 1 0 | l | $registers$

The PUSH instruction pushes one or more of the R0-R7 registers, and optionally the LR, onto the stack. A register Rn is pushed if and only if bit n of the $registers$ field is 1. For instance, if $registers$=11010_2 then registers R1, R3 and R4 are pushed. The LR is pushed if the l field is 1. These registers are pushed in *decreasing index order* (*i.e.*, first the LR if it is selected, then R7 if selected, and so on down to R0). The selected register with the smallest index thus ends up on top of the stack (see Figure 7.4).

POP	$registers, p \leftarrow \mathrm{stack}$	1 0 1 1 1 1 0 | p | $registers$

The POP instruction pops some values from the stack, and stores them in one or more of the R0-R7 registers, and optionally in the PC. A popped value is stored in register Rn if and only if bit n of the $registers$ field is 1. A popped value is stored in the PC if the p field is 1. In this case, it must be an interworking address. The number of values popped from the stack is equal to the number of bits set to 1 in the previous

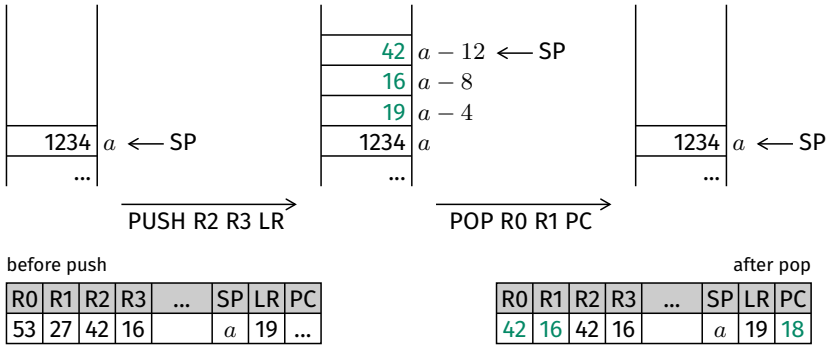


FIGURE 7.4 The PUSH and POP instructions. Registers are pushed in decreasing index order (here LR, then R3, then R2). They are popped in increasing index order (here R0, then R1, then PC – set to 18 = 19 - 1 because of interworking addresses).

fields. The popped values are stored in registers in *increasing index order* (*i.e.*, first in R0 if it is selected, then in R1 if selected, and so on up to the PC – see Figure 7.4).

7.3.3 Jump instructions

B $\quad \mathrm{PC} \leftarrow \mathrm{PC} + signed_{12}(2 * c)$ $\quad$ 1 1 1 0 0 | c

The Branch (B) instruction jumps to an address which is obtained by adding a constant $2 * c$ to the PC. More precisely:

- if $2 * c < 2^{12-1}$, *i.e.*, if $2 * c < 2048$, $2 * c$ is added to the PC,
- otherwise, $2 * c - 2^{12} = 2 * c - 4096$, which is negative, is added to the PC.

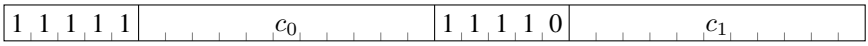
BX $\quad \mathrm{PC} \leftarrow \mathrm{R}x - 1$ $\quad$ 0 1 0 0 0 1 1 1 0 | x | 0 0 0

The Branch and Exchange (BX) instruction jumps to an *interworking* target address stored in a register. It thus jumps to the instruction at address t if the register contains an odd value $t + 1$.

BLX $\quad \mathrm{PC} \leftarrow \mathrm{R}x - 1, \mathrm{LR} \leftarrow a + 3$ $\quad$ 0 1 0 0 0 1 1 1 1 | x | 0 0 0

The Branch with Link and Exchange (BLX) instruction does a branch *with link* operation. It does the same jump as the BX instruction, but it also sets the LR to the *interworking address* of the instruction just after itself. This is done so that the LR can be directly copied into the PC to return from the called subroutine, without having to think about interworking addresses. $\mathrm{R}x$ must not be the PC.

BL $\quad \mathrm{PC} \leftarrow \mathrm{PC} + signed_{23}(2 * c_1{:}c_0), \mathrm{LR} \leftarrow a + 5$

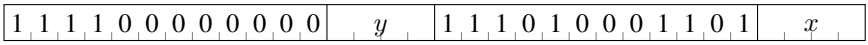


The Branch with Link (BL) instruction[5] jumps to an address which is obtained by adding a constant $2 * c$ to the PC (where $c = c_1{:}c_0$). More precisely:

- if $2 * c < 2^{23-1}$, *i.e.*, if $2 * c < 4$ MB, $2 * c$ is added to the PC,
- otherwise, $2 * c - 2^{23} = 2 * c - 8$ MB, which is negative, is added to the PC.

In addition, the BL instruction sets the LR to the *interworking address* of the instruction just after itself. This is done so that the LR can be directly copied into the PC to return from the called subroutine, without having to think about interworking addresses.

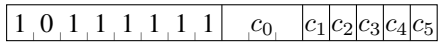
TBB $\text{PC} \leftarrow \text{PC} + 2 * \text{mem8}[\text{R}x + \text{R}y]$



The Table Branch Byte (TBB) instruction jumps to an address which is obtained by adding to the PC (the double of) a byte offset, read in a table. For instance, suppose there is a list of 5 bytes, $[12, 34, 56, 78, 90]$, starting at address a in memory. If $\text{R}x = a$ and $\text{R}y = 3$, then this instruction adds to the PC the double of the 3^{rd} value in this table (counting from 0), *i.e.*, $2 * 78$. Note that $\text{R}x$ can be the PC (but not $\text{R}y$ – none of them can be the SP). In this case the table must start at the PC, *i.e.*, just after the TBB instruction itself (because the PC is the instruction's address plus 4).

7.3.4 Conditional instructions

IT if $c_0{:}c_n$ then I_n



The If Then (IT) instruction makes the following one to four instructions conditional. This means that these instructions, noted I1, I2, I3, and I4, can either be executed normally, or skipped. For instance, I1 and I3 could be executed, while I2 and I4 are skipped (*i.e.*, after I1, execution would continue with I3, ignoring I2). Whether In is executed or skipped depends on c_0, c_1, etc, and on the result of the last CMP instruction[6], as explained below.

First, I1 is always made conditional by the IT instruction, but I2, I3, and I4 can be made conditional or not. This depends on $c_2{:}c_3{:}c_4{:}c_5$, which must not be 0:

- if $c_2{:}c_3{:}c_4{:}c_5 = 1000_2$, then only I1 is made conditional,
- if $c_2{:}c_3{:}c_4{:}c_5 = c_2100_2$, then only I1 and I2 are made conditional,
- if $c_2{:}c_3{:}c_4{:}c_5 = c_2c_310_2$, then only I1, I2 and I3 are made conditional,
- if $c_2{:}c_3{:}c_4{:}c_5 = c_2c_3c_41_2$, then I1, I2, I3 and I4 are made conditional.

[5] For simplicity, we use a restricted version of the BL instruction, with 2 bits fixed to 1. The unrestricted instruction can support ±16 MB jumps.

[6] For simplicity, we use IT instructions only immediately after a CMP instruction.

Value	0000_2	0001_2	0010_2	0011_2	1000_2	1001_2
Meaning	$a = b$	$a \neq b$	$a \geq b$	$a < b$	$a > b$	$a \leq b$

TABLE 7.1 Meaning of the $cond_n = c_0{:}c_n$ values used in this book, for an IT instruction following a CMP $compare(a, b)$ instruction.

Second, if an instruction In is made conditional, it is executed if and only if the last comparison result is the one corresponding to $c_0{:}c_n$ (noted $cond_n$), as defined in Table 7.1. As an example, if the IT instruction follows a CMP R1 R2 instruction, then to execute I1 if R1 $<$ R2, and I2 and I3 otherwise (with I4 unconditional), we must use $c_0 = 001_2$, $c_1 = 1_2$ and $c_2{:}c_3{:}c_4{:}c_5 = 0010_2$.

Finally, it should be noted that I1, I2, I3, I4 cannot be arbitrary instructions. For instance, they cannot be IT instructions themselves. Jump instructions are also forbidden, except for the last conditional instruction.

7.4 Vector Table

When it *boots*, *i.e.*, when it is powered on or after a reset, the Cortex M3 starts by reading the *Vector Table*, a list of 32 bit words beginning at address 0_{16}. More precisely, the microprocessor initializes the Link Register to FFFFFFFF$_{16}$, the Stack Pointer to the value at offset 0_{16} in this table, and the Program Counter to the interworking address at offset 4_{16}, also called the *Reset handler*. It then starts executing instructions from there.

When the boot selection bit (cf. Section 6.5.2) is 0, addresses 0_{16} and 4_{16} are mapped to the ROM, and the Vector Table is thus read from ROM. The Cortex M3 then starts executing the boot assistant program stored in ROM. When the boot selection bit is 1, these addresses are mapped to the flash memory, and the Vector Table is thus read from flash. In this case, the flash memory must contain valid initial values for the Stack Pointer and the Program Counter, at addresses 80000_{16} and 80004_{16}, respectively.

The other values in the Vector Table are either reserved for future use, or contain interworking addresses used to handle errors and external events. There is one value for each type of error (also called exceptions) and external events (also called interrupts). For instance, the value at offset 14_{16} is used to handle "bus faults", such as trying to access a reserved memory address. When such an error occurs, execution jumps to the interworking address stored at this offset (this process is described in more details in Section 11.3). Another value, at offset 84_{16}, is used to handle events occuring in the Universal Synchronous Asynchronous Receiver Transmitter (USART) – such as the reception of new data. When such an interrupt occurs, execution jumps to the interworking address stored at offset 84_{16}. By default, however, these specific exception and interrupt handlers are not enabled (they must be enabled explicitly). Instead, all exceptions and interrupts are handled by a generic "hard fault" handler, at offset C$_{16}$. In other words, by default, if any exception or interrupt occurs (other than

a reset), execution jumps to the interworking address stored at offset C_{16}.

The Vector Table can be moved to another location in memory, for instance in RAM. For this a new table must first be written somewhere in memory, starting at an address which is multiple of 256. This address must then be written in the Vector Table Offset Register, at address $E000ED08_{16}$ in the "System" memory region (see Figure 6.3).

7.5 First program

We now know enough about the Arduino to be able to write our first program. We have already seen how to turn a LED on and off "manually", with a wire or by writing values in memory with the boot assistant. Lets now do the same with a program. More precisely, lets write a program to blink the "L" LED on the Arduino. At a high level, this program should execute the following steps:

1. initialize the PIO controller to control the LED,
2. turn the LED on with the PIO controller,
3. wait some time,
4. turn the LED off with the PIO controller,
5. wait some time,
6. go back to step 2.

We see that steps 3 and 5 are the same. To avoid duplicating the instructions doing this, we can put them in a subroutine, called twice from the main program.

7.5.1 Subroutine

Lets start by writing the subroutine. The most basic method to wait for some time is to count to some value, as in a hide and seek game. To do this we can use two registers, say R0 and R1, with R0 containing the current count, and R1 the value at which the count must be stopped. At a high level, we can thus use the following algorithm for our subroutine:

1. set R0 to 0,
2. set R1 to the maximum counter value,
3. add 1 to R0,
4. if R0 $\neq$ R1 go back to step 3,
5. return to the caller.

Each step can be implemented with Cortex M3 instructions, as follows:

- Step 1 can be done with a MOV R0 $\leftarrow$ 0 instruction.
- Step 2 needs to store a large value in R1, lets say 1 million (the Cortex M3 counts fast). This can't be done with a MOV instruction, nor with a MOVW instruction. We could use MOVW and a MOVT, but using a LDR instruction with the PC as a

base address is simpler and shorter. The maximum counter value can then be put at the end of the subroutine, after the instruction for step 5.

- Step 3 is a simple ADD R0 ← R0 + 1 instruction.
- Step 4 must be done with a CMP R0 R1 instruction, followed by an IT instruction to make the "go back to step 3" part conditional. This optional jump is a simple B instruction.
- Step 5 must move the LR into the PC. This could be done with a MOV instruction, but lets use a POP instruction instead, for illustrative purpose. For this the LR must be pushed on the stack first. We can add a step 0 for this. This step can also push R0 and R1 at the same time. These registers can then be restored at the end, so that calling the subroutine has no "side effect".

Going down to the machine code level, steps 0 and 1 give

PUSH R0 R1 $LR \rightarrow$ stack	1011010**1**00000011	B503	000
MOV R0 ← *0*	00100**000**00000000	2000	002

where the first column is the instruction's name and high level description, the second one is its binary encoding (obtained from the patterns in Section 7.3), the third is the corresponding hexadecimal value, and the fourth is the instruction's offset from the beginning of the program, in hexadecimal too. For step 2 we need to know where the maximum counter value will be stored. Since we don't know this yet, lets skip this instruction for now. We know it will use 2 bytes, so the instruction for step 3 starts at offset 6:

ADD R0 ← R0 + *1*	00110**000**00000001	3001	006

For step 4, the optional jump must go back to step 3, *i.e.*, at offset 6. This can be done with a B instruction after the CMP and IT instructions, at offset 12. When this instruction executes, the PC contains $12 + 4 = 16$. Hence, this instruction must subtract $16 - 6 = 10$ from the PC to jump back to step 3. Due to the encoding format of the B instruction, we must then use $c = 2043$ (because $2 * 2043 - 4096 = -10$):

CMP	$compare$(R**0**, R*1*)	0100001010**001000**	4288	008
IT	if ≠ then	10111111**0001***1000*	BF18	00A
B	PC ← PC + 2 * **2043** − 4096	11100**11111111011**	E7FB	00C

We finish the subroutine with a POP instruction for step 5, as discussed above, followed by the maximum counter value ($F4240_{16}$=1000000):

POP	R**0** R**1** $PC \leftarrow$ stack	1011110**1**00000011	BD03	00E
data	(maximum counter value)	000F4240		010

We now know that this maximum value is at offset 16, and we want to load it with a LDR instruction at offset 4. When this instruction executes, the PC contains $4 + 4 = 8$. *Assuming that the subroutine starts at an address which is a multiple of 4*, then $\lfloor PC \rfloor_4$ is also equal to 8. This instruction must thus add $16 - 8 = 8 = 4 * 2$ to the PC to load the above value. This gives our missing LDR instruction:

LDR $\quad R\mathbf{1} \leftarrow \text{mem32}[\lfloor \text{PC} \rfloor_4 + 4 * \mathit{2}]$ $\quad$ 01001**001***00000010* $\quad$ 4902 $\quad$ 004

Putting all this together, we get the following machine code for the subroutine:

```
000F4240 BD03E7FB BF184288 30014902 2000B503   000
```

where bytes are shown, in increasing address order, *from right to left*. Indeed, the left to right order would show all the bytes in the reverse order, which would make it hard to recognize the above instruction encodings. It would also make it harder to store this program in memory with the boot assistant word by word. Indeed, we would have to reverse each group of 4 bytes to get a value which could be entered in the boot assistant. With the right to left order, we get these values directly.

7.5.2 Main program

We can now implement the main program. Its first step consists in initializing the PIO controller so that the pin to which the LED is connected, PB27, is configured as an output pin, controlled by the microprocessor. In such cases we don't need the pull-up resistor, so we want to disable it as well. Going back to Section 6.6, this requires to write the value 2^{27} in the PIO Enable Register, Output Enable Register, and Pull-up Disable Register of the PIO B controller (*i.e.*, at addresses $400E1000_{16}$, $400E1010_{16}$, and $400E1060_{16}$). For this we must load this value and the 3 addresses in registers first[7]. We can then use 3 STR instructions to store the value at the 3 addresses. In fact, since STR instructions can store a value at an address *plus an offset*, we just need to load one address in a register, namely $400E1000_{16}$. We can then get the other 2 addresses with the offsets 10_{16} and 60_{16}. As for the subroutine, lets use 2 LDR instructions to load the $400E1000_{16}$ and 2^{27} values, stored at the end of the program, in registers R0 and R1 respectively. Since we don't know the program size yet, lets skip these two LDR instructions for now. We thus start with the 3 STR instructions instead, at offset 24=18_{16} (*i.e.*, 4 bytes after the end of the subroutine, to leave space for the 2 LDR instructions):

STR $\quad R\mathbf{1} \rightarrow \text{mem32}[\text{R}\mathit{0} + 4 * 0]$ $\quad$ 01100 00000 *000* **001** $\quad$ 6001 $\quad$ 018

STR $\quad R\mathbf{1} \rightarrow \text{mem32}[\text{R}\mathit{0} + 4 * 4]$ $\quad$ 01100 00100 *000* **001** $\quad$ 6101 $\quad$ 01A

STR $\quad R\mathbf{1} \rightarrow \text{mem32}[\text{R}\mathit{0} + 4 * 24]$ $\quad$ 01100 11000 *000* **001** $\quad$ 6601 $\quad$ 01C

The second step of the main program is to turn the LED on. This requires writing the value 2^{27} in the Set Output Data Register, *i.e.*, at address $400E1030_{16}$. This can be done with another STR instruction:

STR $\quad R\mathbf{1} \rightarrow \text{mem32}[\text{R}\mathit{0} + 4 * 12]$ $\quad$ 01100 01100 *000* **001** $\quad$ 6301 $\quad$ 01E

We now want to wait some time, by calling the subroutine. For this we need a Branch with Link instruction. This instruction is at offset 32, and the subroutine starts at offset 0, so we need to subtract $32 + 4 = 36$ from the PC to jump there. Due to

[7] In the following, for brevity, we often use "we" instead of "the program" or "some instructions". For instance, here, this sentence means "For this *the program* must load . . .".

the encoding format of the BL instruction, this means we need to use $c = 4194286$ (because $2 * 4194286 - 8 * 1024 * 1024 = -36$):

BL	PC $\leftarrow$ PC $+ 2 *$ **4194286** $-$ 8 MB	11110**11111111111**	F7FF	020
		11111**11111101110**	FFEE	022

After that we want to turn the LED off and wait some time again. This requires writing the value 2^{27} in the Clear Output Data Register, *i.e.*, at address $400E1034_{16}$, and calling the subroutine again. Following the same method as above, we get:

STR	R**1** $\rightarrow$ mem32[R*0* $+ 4 *$ 13]	01100**01101***000***001**	6341	024
BL	PC $\leftarrow$ PC $+ 2 *$ **4194283** $-$ 8 MB	11110**11111111111**	F7FF	026
		11111**11111101011**	FFEB	028

The last step of the main program is go back to step 2, with a B instruction. Following the same reasoning as we did for the subroutine, we find that we need to use $c = 2040$, to jump from offset 42 to offset 30. With the two values $400E1000_{16}$ and 2^{27} just after this last instruction we obtain:

B	PC $\leftarrow$ PC $+ 2 *$ **2040** $- 4096$	11100**11111111000**	E7F8	02A
data	(Start address of PIO B registers)		400E1000	02C
data	(Pin 27)		08000000	030

Knowing the location of these two values, *i.e.*, offsets 44 and 48, we can now implement the two LDR instructions we skipped at the beginning (for the two LDR instructions, $\lfloor PC \rfloor_4$ contains $24 = 14_{16} + 4$):

LDR	R**0** $\leftarrow$ mem32[$\lfloor$PC$\rfloor_4 + 4 * 5$]	01001**000***00000101*	4805	014
LDR	R**1** $\leftarrow$ mem32[$\lfloor$PC$\rfloor_4 + 4 * 6$]	01001**001***00000110*	4906	016

Putting everything together, we obtain the complete machine code for the main program and its subroutine, with the main program starting at offset 14_{16}:

```
49064805 000F4240 BD03E7FB BF184288 30014902 2000B503   000
400E1000 E7F8FFEB F7FF6341 FFEEF7FF 63016601 61016001   018
                                    08000000             030
```

7.5.3 Run from RAM

As mentioned in Section 6.4, the boot assistant has a G*address*# command to run a program. We show here how to use it to run our "blink LED" program.

The G*address*# command seems to take as argument the start address of a "mini Vector Table" (this is not documented in [8]). Indeed, this command does not jump to *address*, but to the *interworking address* stored at *address* $+ 4$ (just as the Cortex M3 starts by jumping to the interworking address stored at address 4). Presumably, the value at *address* is meant to contain an initial Stack Pointer value, as in the Vector Table. However, experiments show that this is not the case, *i.e.*, programs run with

this command execute on the stack used by the boot assistant itself. They can return to the boot assistant by moving the LR to the PC, but the SP must be the same on return as what it was on entry.

To run our program with the boot assistant, the easiest way is to store it in RAM. Lets put it after the first 4 KB, used by the boot assistant, at address 20071000_{16} (a multiple of 4, as we assumed when we wrote it). To run it with the `G` command we must have the interworking address of its first instruction (20071000_{16} + 14_{16} + 1) somewhere in memory too. We can put it just after our program, at address 20071034_{16}. We should then be able to run the program with the `G20071030#` command. To verify this, connect the Arduino to your computer and open a terminal as you did in Section 6.4.3. Then write the program into RAM with the following commands:

```
user@host:~$ python3 boot_helper.py
>W20071000,2000B503#
>W20071004,30014902#
>W20071008,BF184288#
>W2007100C,BD03E7FB#
>W20071010,000F4240#
>W20071014,49064805#
>W20071018,61016001#
>W2007101C,63016601#
>W20071020,FFEEF7FF#
>W20071024,F7FF6341#
>W20071028,E7F8FFEB#
>W2007102C,400E1000#
>W20071030,08000000#
>W20071034,20071015#
```

And launch it with

```
>G20071030#
ERROR: no response from device.
```

If you didn't make any typo in the above commands, you should see the LED blinking! After 5 seconds the `boot_helper.py` program exits because it hasn't received any response from the boot assistant, but our program continues to run on the Arduino. You can then reset the Arduino to go back to the boot assistant (at this point our program is lost).

7.5.4 Run from flash

To avoid losing our program after it is run, we can store it in flash memory. We can even run it directly when the Arduino boots, without going through the boot assistant. This requires two things: storing a proper Vector Table in flash memory (cf. Section 7.4), and setting the boot mode selection bit to boot from flash (cf. Section 6.5).

By default, only the first, second and fourth words of the Vector Table are used (for the initial SP, initial PC, and the "hard fault" handler – the third one is "reserved"), since the exception and interrupt handlers are disabled by default. We can thus store our program after these 4 words, *i.e.*, starting at offset 10_{16} (a multiple of 4, as needed). The initial PC must then be set to 80000_{16} (the beginning of the flash memory region), plus 10_{16}, plus 14_{16} (the offset of the main program after the subroutine), plus 1 (an interworking address is required here). The initial SP can be set to almost any address in RAM (our program pushes at most 3 words on the stack, so any value larger

than 12 bytes after the beginning of the RAM region is fine). Lets use the end of the contiguous RAM region, 20088000_{16} (see Figure 6.3). Finally, the "hard fault" handler can be set to the same value as the initial PC, 80025_{16}, so that, in case of errors, our program restarts from the beginning.

The above Vector Table and our program use 17 words in total, much less than the 64 words of a flash memory page. However, as we have seen in Section 6.5, writing a page in flash memory requires writing 64 words in all cases. To avoid this extra work, we provide a program called `flash_helper.py`. This program extends `boot_helper.py` with an additional command named `flash#`. When it runs on the host computer, this program does the following:

- When it receives a `W`*address*`,`*value*`#` command with an address in flash memory, *instead of sending it to the Arduino*, `flash_helper.py` sends 64 `w` commands to read the corresponding page. It stores the result in memory (on the host computer), and writes *value* in this copy of the page.
- When it receives the `flash#` command, `flash_helper.py` writes the (modified) page copies in the Arduino's flash memory. For this, for each page, it sends 64 `W` commands, followed by a `W` command in the Flash Controller Command Register to write the page, followed by `w` commands to read the Status Register until the write is done (cf. Section 6.5).
- When it receives any other boot assistant command, `flash_helper.py` sends it directly to the Arduino.

Note that with this program, we can modify a single word in flash memory, without modifying the 63 other words of the page, with only two commands (namely a `W` and a `flash#` – with `boot_helper.py` more than 128 commands would be needed). Lets use it to write our program:

```
user@host:~$ python3 flash_helper.py
>W80000,20088000#
Reading page 0... Done.
>W80004,00080025#
>W8000C,00080025#
>W80010,2000B503#
>W80014,30014902#
>W80018,BF184288#
>W8001C,BD03E7FB#
>W80020,000F4240#
>W80024,49064805#
>W80028,61016001#
>W8002C,63016601#
>W80030,FFEEF7FF#
>W80034,F7FF6341#
>W80038,E7F8FFEB#
>W8003C,400E1000#
>W80040,08000000#
>flash#
Writing page 0... Done.
```

At this point we could run our program with a `G80000#` command, and we would no longer loose it after a reset. Instead, lets set the boot mode selection bit to boot from flash (cf. Section 6.5):

```
>W400E0A04,5A00010B#
```

Then press the RESET button on the Arduino: if you didn't make any typo in the above commands, you should see the LED blinking. However, two things can be noticed. First, the LED blinks slower than before (about 1 blink every 2 seconds).

This is because, by default, the Arduino's clock runs at 4MHz. But when the boot assistant starts, it sets the clock to a larger frequency. Second, if you watch the LED at least 30 seconds, you can see that some blinks are shorter than others. This is because our program is reset by the Watchdog Timer (cf. Section 6.2). The boot assistant disables it when it starts, which is why we didn't have this issue before. We explain in Chapter 9 how to configure the clock and how to disable the Watchdog Timer.

Our program is now persistent and runs autonomously, without needing the boot assistant. But what if we want to modify it, for instance to make it blink faster? For this we need to go back to the boot assistant. The only way to do this at this point is to do a full ERASE, in order to reset the boot mode selection bit to boot from ROM. Unfortunately, this also erases the flash memory, and thus our program too. We would then need to flash the whole program again, even if we just need to change one word (the one containing the maximum counter value). To avoid this issue, the solution is to add a few instructions at the beginning of our program, in order to set the boot mode selection to boot from ROM as soon as our program starts. In this way, when we reset it, we will automatically run the boot assistant again, without needing to do an ERASE.

CHAPTER

8 Virtual Machine

As we have seen in the previous chapter, writing programs in machine code is quite complex and error prone. The machine code instructions are complex mainly because the machine itself (*i.e.*, the Cortex M3 microprocessor) is complex. The machine code instructions also have complex encodings. This complexity is manageable for very small programs, but the basic input output system we want to write in this part is not so small. In order to simplify its implementation, a solution is to use a simpler instruction set, which in turn requires a simpler machine. This chapter defines such a machine, and its set of instructions, called *bytecode instructions*. It is called a *virtual machine* because it is not a physical chip. Instead, we provide in this chapter an *interpreter* for this virtual machine. This small program simulates programs running on the virtual machine. It does this by executing, for each bytecode instruction, an equivalent sequence of Cortex M3 instructions.

8.1 Overview

The main source of complexity in the Cortex M3, and in most microprocessors, are the registers. They are quite complex to use because there are only a small number of them (16 on the Cortex M3). When they are all "full", some of them must be copied into memory (usually on the stack), so that they can store new values without losing the old ones. The old values copied in memory must usually be copied back into the registers at some point. This makes it hard to keep track of which value is stored where at any point in time.

In order to get a simpler machine, a solution is thus to *remove the registers*. This leaves only the stack and the (rest of the) memory. This also means that instructions for this machine can only use the stack and the memory. For instance, an arithmetic instruction can no longer read its operands in registers, nor write the result in a register. A solution is to pop the operands from the stack, and to push the result on the stack. This example shows another advantage of removing the registers: instructions no longer need to encode on which registers they operate. In fact, arithmetic instructions no longer need to have any argument. They can be encoded with a single constant value. Moreover, since we no longer need some bits to encode the register indices, this value can be small, only one byte. Hence the name *byte*code instructions. A virtual machine for such stack-based instructions is called a (virtual) *stack machine*.

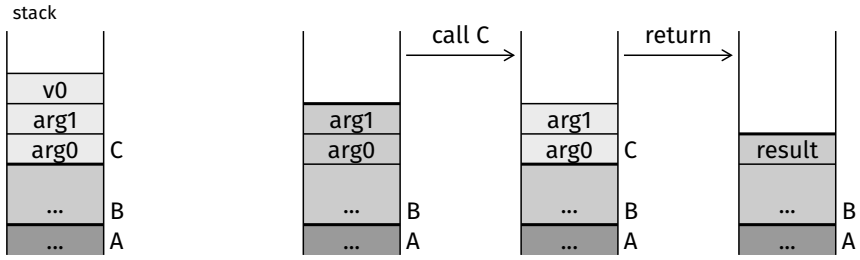


FIGURE 8.1 Stack frames and function calls. Left: a stack with 3 stack frames, corresponding to a function A calling a function B itself calling a function C. Each stack frame contains the arguments passed to the function (*e.g.*, arg0 and arg1 passed in the call from B to C), and optionally some values pushed during the function's execution (*e.g.*, v0). Right: the callee C pops its arguments (arg0, arg1) from the caller's stack frame (B), and pushes its result on the caller's stack frame when it returns.

Removing the registers also impacts subroutines. Subroutines usually take some *arguments* as input, and may return a result as output. For instance, a subroutine to draw a character could take a character and a position on the screen as input, and could return the position where the next character should be drawn. In general, these input and output values could be stored in registers, on the stack, or a mix of the two. But if we remove the registers, the virtual machine subroutines, called *functions*, are forced to use the stack. This means that, as arithmetic instructions, a function must pop its arguments from the stack, and push its result on the stack.

The consequence is that the stack is organized into *stack frames* as shown in Figure 8.1. A stack frame is a contiguous part of the stack, corresponding to a function call. Its contains the function arguments pushed by the caller, optionally followed by the intermediate values pushed by the function's instructions executed so far. When this function calls another one, the arguments it pushed on the stack become the beginning of a new stack frame, corresponding to the callee (see Figure 8.1). When the callee returns, its entire stack frame is popped and replaced with the result value.

As arithmetic and logic instructions are simplified by using the stack instead of registers, the other instructions can be simplified by using stack frames instead of lower level concepts (*e.g.*, a Link Register). For instance, we can define an instruction to get the i^{th} argument of the top stack frame (*i.e.*, of the currently executing function), and to push it on the stack. Doing this with Cortex M3 instructions would require several instructions dealing explicitly with the Stack Pointer. As another example, we can define an instruction to return a value from a function. As explained above, this requires popping the top stack frame, and pushing the return value. Doing this with Cortex M3 instructions would require several instructions.

In summary, for all the reasons explained above, the virtual machine designed and implemented in this chapter is a virtual stack machine, with instructions using the stack, stack frames and functions instead of registers. The next section defines its

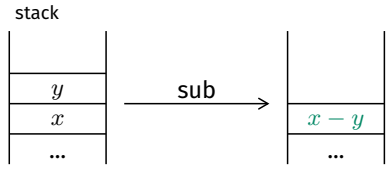


FIGURE 8.2 The sub instruction pops two values x and y from the stack and pushes their difference $x - y$.

instruction set.

8.2 Bytecode instructions

8.2.1 Arithmetic and logic instructions

cst_0	$push(0)$		00
cst_1	$push(1)$		01
cst8	$push(c)$	c	02
cst	$push(c)$	c	03

The `cst_0` instruction, encoded with the byte 00_{16} (also called *opcode*, for operation code), pushes the value 0 on the stack. The `cst_1` instruction does the same with the value 1. The `cst8` instruction pushes an 8-bit value on the stack. It is encoded with the opcode 02_{16}, followed by the byte value to push (bytes are shown, in increasing address order, *from right to left*). Finally, the `cst` instruction pushes a 32-bit value on the stack. This value is put directly after the opcode, without any encoding (compare this with the MOVW and MOVT instructions). Note that `cst_0`, `cst_1` and `cst8` are not strictly necessary (a `cst` instruction can be used instead). They are provided to reduce the size of programs.

add	$y \leftarrow pop(), x \leftarrow pop(), push(x + y)$	04
sub	$y \leftarrow pop(), x \leftarrow pop(), push(x - y)$	05
mul	$y \leftarrow pop(), x \leftarrow pop(), push(x * y)$	06
div	$y \leftarrow pop(), x \leftarrow pop(), push(x/y)$	07
and	$y \leftarrow pop(), x \leftarrow pop(), push(x \wedge y)$	08
or	$y \leftarrow pop(), x \leftarrow pop(), push(x \vee y)$	09
lsl	$y \leftarrow pop(), x \leftarrow pop(), push(x \ll y)$	0A
lsr	$y \leftarrow pop(), x \leftarrow pop(), push(x \gg y)$	0B

The arithmetic and logic instructions are direct analogues of the Cortex M3 instructions with the same name. As discussed above, they use the stack instead of

registers, and can thus be encoded with only one byte each. For instance, the sub instruction pops a value from the stack, say y, then pops another value, say x, and then pushes $x - y$ on stack (and not $y - x$ – see Figure 8.2).

8.2.2 Jump instructions

iflt	$y \leftarrow pop(), x \leftarrow pop()$, jump to c if $x < y$	c	0C
ifeq	$y \leftarrow pop(), x \leftarrow pop()$, jump to c if $x = y$	c	0D
ifgt	$y \leftarrow pop(), x \leftarrow pop()$, jump to c if $x > y$	c	0E
ifle	$y \leftarrow pop(), x \leftarrow pop()$, jump to c if $x \leq y$	c	0F
ifne	$y \leftarrow pop(), x \leftarrow pop()$, jump to c if $x \neq y$	c	10
ifge	$y \leftarrow pop(), x \leftarrow pop()$, jump to c if $x \geq y$	c	11

The "if less than" (`iflt`) instruction pops a value y from the stack, then pops a value x, and finally jumps to offset c if $x < y$. The other instructions are similar, for equal (eq), greater than (gt), less than or equal (le), not equal (ne), and greater than or equal (ge) conditions. To simplify the encoding, the c offset is always interpreted as a nonnegative value (compare this with the B instruction). In return, this offset is defined relatively to the beginning of the currently executing function, *i.e.*, to the address of its first instruction.

Note that these *conditional jump* instructions are sufficient to make any other instruction or sequence of instructions conditional. Indeed, for this, it suffice to use a conditional jump instruction before the sequence, jumping after it if some condition happens. We therefore do not use an IT-like instruction, nor a CMP-like one.

goto	jump to c	c	12

The `goto` instruction unconditionally jumps to offset c, like the B instruction. This offset is defined as above, *i.e.*, as a nonnegative offset from the beginning of the current function.

8.2.3 Memory and stack instructions

load	$x \leftarrow pop(), push(\text{mem32}[x])$	13
store	$v \leftarrow pop(), x \leftarrow pop(), \text{mem32}[x] \leftarrow v$	14

The `load` and `store` instructions are direct analogues of the Cortex M3 LDR and STR instructions, using the stack instead of registers. More precisely, the `load` instruction pops an address from the stack, reads a 32-bit value at this address in memory, and pushes it on the stack. The `store` instruction pops a value, then pops an address, and finally stores the 32-bit value at this address in memory. To simplify the interpreter, we do not define similar instructions for 16 and 8-bit values.

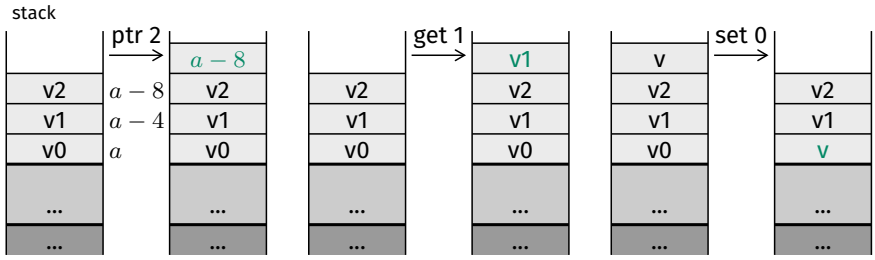


FIGURE 8.3 The stack frame instructions. The `ptr` i instruction (left) pushes the address of the i^{th} value in the top stack frame (in light gray), counting from 0. The `get` i instruction (middle) pushes the i^{th} value itself. Finally, the `set` i instruction (right) replaces the i^{th} value with a new value popped from the stack.

ptr	$push(\text{address of frame}[i])$	i	15
get	$push(\text{frame}[i])$	i	16
set	$\text{frame}[i] \leftarrow pop()$	i	17

The `ptr` instruction pushes a *pointer* on the stack, namely the address of the i^{th} word in the top stack frame (counting from 0). The `get` instruction reads the i^{th} word in the top stack frame, and pushes it on the stack. The `set` instruction pops a value from the stack, and stores it in the i^{th} word in the top stack frame (see Figure 8.3). These instructions can be used to access the arguments of the currently executing function, but also the values it has pushed on the stack during its execution.

pop	$pop()$		18

Finally, the pop instruction simply pops a value from the stack, and discards it.

8.2.4 Function instructions

fn	$push\_frame(n)$	n	19

The `fn` instruction starts a function with n arguments. It must be the first instruction of any function. Its role is to define a new stack frame, using the top n values on the stack (see Figure 8.4). This is needed for the `ptr`, `get` and `set` instructions presented above to "know" where the top stack frame begins in the stack.

call	call function at $C0000_{16} + c$	c	1A
callr	call function at $a - c$	c	1B
calld	$x \leftarrow pop()$, call function at x		1C

The `call` instruction calls a function at an offset c from the start of the Flash1 memory bank, $C0000_{16}$ (the instruction at this address must be a `fn`). This gives a

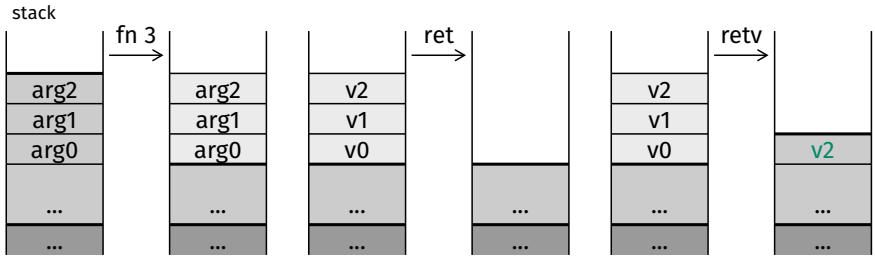


FIGURE 8.4 The function instructions. The `fn` n instruction (left) creates a new top stack frame (light gray) with the top n values of the previous top stack frame (gray). The `ret` instruction (middle) pops the top stack frame. The `retv` instruction (right) pops a value from the stack, pops the top stack frame (light gray), and then pushes this value on the new top stack frame (gray).

very simple encoding (compare this with the BL instruction), but restricted to a fixed 64kB address range. To support more addresses, we also define `callr` and `calld` instructions. The former calls a function at an offset $-c$ from its own address, noted a. The latter pops a value from the stack, interpreted as an address, and calls the function at this address.

ret $pop\_frame()$ 1D

retv $x \leftarrow pop(), pop\_frame(), push(x)$ 1E

The return (`ret`) instruction returns from a function without output value. It pops the top stack frame, and resumes execution in the caller. The return value (`retv`) instruction returns from a function with a result value. It pops this result from the stack, pops the top stack frame, and finally pushes back the result value (see Figure 8.4). It then resumes execution in the callee.

blx $x \leftarrow pop(), \text{BLX } x$ 1F

Finally, the `blx` instruction calls a Cortex M3 subroutine at an interworking address popped from the stack, with a BLX instruction. It can be used to execute Cortex M3 instructions without an equivalent bytecode instruction.

8.3 Interpreter

We can now implement an *interpreter* for the virtual machine and the instruction set defined above. A bytecode interpreter executes bytecode instructions given as input, like a microprocessor executes machine code instructions. A simple method to do this is to read a bytecode instruction, execute an equivalent sequence of machine code instructions, read another bytecode instruction, execute the equivalent machine instructions, and so on. In our case, this gives the following overall algorithm:

ALGORITHM 8.1 The overall virtual machine algorithm.

1. read the first byte of the current bytecode instruction,
2. if it is an undefined opcode (*i.e.*, strictly larger than 32), trigger an exception,
3. if it is 0 (`cst_0`), execute a sequence of Cortex M3 instructions to push a 0 on the stack, advance to the next instruction, and go back to step 1,
4. if it is 1 (`cst_1`), execute a sequence of Cortex M3 instructions to push a 1 on the stack, advance to the next instruction, and go back to step 1,
5. if it is 2 (`cst8`), read the next byte of the current instruction, execute a sequence of Cortex M3 instructions to push this byte on the stack, advance to the next instruction, and go back to step 1,
6. ... and so on for the remaining 29 instructions ...

8.3.1 Registers

In order to implement this algorithm we need to use some registers. Indeed, although the bytecode instructions do not use registers directly, their interpreter is made of machine code instructions, which do use registers. We can deduce the registers we need from the above algorithm and from the instruction definitions:

- For each step above we need a register holding the address of the current bytecode instruction. We call it the Instruction Counter (IC), by analogy with the Cortex M3 Program Counter.
- For the arithmetic and logic instructions we need two registers to store the operands and the operation result. We can use the R0 and R1 Cortex M3 registers for this. We also need a stack pointer, to know where to pop the operands from, and where to push the result. The simplest is to use the Cortex M3 Stack Pointer (SP) for this, to take advantage of the PUSH and POP instructions.
- For the jump instructions we need a register containing the address of the first instruction of the currently executing function. Indeed, the offset used by these instructions is defined relatively to this address. We call this register the Function Address (FA).
- For the stack frame instructions we need a register storing the address of the top stack frame, *i.e.*, the address of its 0^{th} value. Indeed, this is necessary to compute the address of an i^{th} value. This register is usually called the Frame Pointer (FP).
- For the function instructions, in particular the `ret` and `retv` instructions, we need a register pointing to the caller's instruction to return to, similar to the Link Register. We call it the Return Address (RA).

Finally, at each function call, we need to save the registers containing information about the caller, so that we can update them with data about the callee instead. This is the case of the Function Address, the Return Address, and the Frame Pointer. We can save them by pushing them on *some* stack (see below), and pop them when the callee returns. This requires another stack pointer, and thus one last register. We call it the Backup Pointer (BP). In summary, besides the Stack Pointer, we need a total of

Index	Symbol	Name	Pointer target
R0	R0	–	–
R1	R1	–	–
R2	IC	Instruction Counter	Current instruction
R3	RA	Return Address	Instruction to return to
R4	FA	Function Address	0^{th} instruction of current function
R5	FP	Frame Pointer	0^{th} value of top stack frame
R6	BP	Backup Pointer	Last saved register value

TABLE 8.1 The registers used by our interpreter.

7 registers. We map them to the R0 to R6 registers as shown in Table 8.1

8.3.2 Stack frame layout

As discussed above, we need a stack to save and restore the RA, FA, and FP registers. Using a custom stack requires deciding where to store it in memory, and how much memory to reserve for it. To avoid these issues, we use the same stack as the one used for stack frames, *i.e.*, the one managed with the SP. The consequence is that stack frames contain saved registers between[1] the callee's arguments, and values pushed by the callee (see Figure 8.5). This is not ideal since bytecode programs become dependent on implementation details of the interpreter. For instance, if the interpreter is updated to save an additional register, the indices used in `ptr`, `get` and `set` instructions must be updated as well. But we don't plan to change the interpreter, so this is not really an issue.

Another consequence of saving the registers on the same stack as stack frames is that the Backup Pointer must be saved on the stack as well. This register contains the address of the last saved register value in memory. If a separate stack was used to save the registers, we could just increment or decrement it by $12 = 3 * 4$ since we would push and pop registers by groups of 3 (RA, FA, FP). But when registers are saved on the same stack as stack frames, the offset between each group of saved registers varies (see Figure 8.5). By saving the BP inside each group we get in each group the previous BP value, pointing to the previously saved group. The result is a *linked list* data structure, called this way because each list element has a link (a pointer) to the next element (see Figure 8.5).

8.3.3 Initialization

In order to facilitate its use from *native programs*, *i.e.*, programs written in machine code, the microprocessor's native language, we propose to implement the interpreter

[1] We could also put the saved registers between the caller and callee stack frames, but this is more complex to implement.

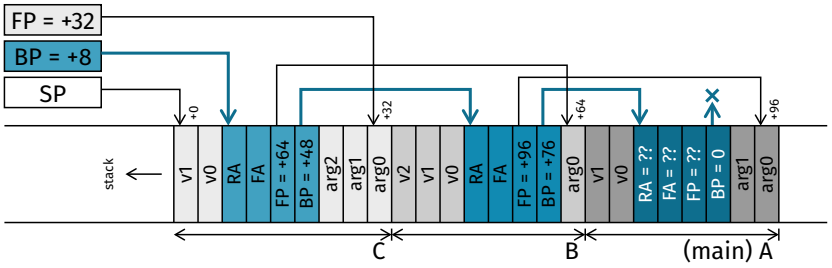


FIGURE 8.5 The stack frame layout. Each stack frame (here for a function A calling B, itself calling C) contains, in this order, the callee's arguments (arg0, etc), the saved caller's registers (blue), and values pushed by the callee (v0, etc). The Frame Pointer FP points to the bottom of the top stack frame. The Backup Pointer BP points to the last saved register. Values noted $+x$ are offsets from the Stack Pointer SP.

as a subroutine. This allows a native program P to call the interpreter with some bytecode program Q to interpret, which returns to P when it is done. More precisely, we define the following interface:

- P saves the registers it uses if necessary (the interpreter might override them).
- P optionally pushes some arguments on the stack, for Q.
- P stores the address of Q's first instruction in R2, the Instruction Counter.
- P calls the interpreter subroutine with a BL or BLX instruction.
- The interpreter executes Q, and finally returns to P.

For this the interpreter must be able to detect when a bytecode program ends. To this end, we assume that bytecode programs always start with a `fn` instruction[2]. The end of the program is then defined as the point where this function, called the *main* function, returns. We can then detect the end of the program as follows:

- Initialize the Backup Pointer to 0, an invalid stack address (the stack is restricted to the RAM region, which does not contain this address).
- The end of the program is reached if and only if the Backup Pointer is 0 when a function returns.

Indeed, if the BP is 0 this means that the current function was not called by another function, *i.e.*, is the main function. Otherwise the BP would contain a valid stack address, pointing to saved registers.

In order to implement the above interface, we need to add an initialization step before Algorithm 8.1. This step only needs to initialize the Backup Pointer to 0 (the initial value of R0, R1, RA, FA, and FP is never used, and IC is set by the caller):

instruction		encoding	offset
MOV R6 ← *0*	00100**110***00000000*	2600	000

[2]We could add machine code to check this during the initialization step but we don't, to simplify.

8.3.4 Instruction dispatch

We can now implement step 1 of Algorithm 8.1. This step reads a byte at the address stored in the Instruction Counter, and stores it in R0. It then increments the IC by 1 to prepare reading the instruction's arguments, or the next instruction:

LDRB $\mathrm{R}\mathbf{0} \leftarrow \mathrm{mem8}[\mathrm{R}2 + 0]$	01111 00000 *010* **000**	7810	002
ADD $\mathrm{R}\mathbf{2} \leftarrow \mathrm{R}\mathbf{2} + 1$	00110 **010** *00000001*	3201	004

Step 2 must trigger an exception if the opcode in R0 is undefined, *i.e.*, if it is strictly larger than 31. If this happens, execution will then jump to the Hard Fault handler (cf. Section 7.4). If it is a blink LED subroutine for instance, as we did in Section 7.5.4, the LED will blink when an undefined opcode is found. The Cortex M3 has an instruction which is precisely done for this case. It is called "Permanently Undefined" (UDF), and triggers an Undefined Instruction exception. It is encoded as 1 1 0 1 1 1 1 0 | c, where c is ignored. This gives the following instructions for step 2:

CMP $compare(\mathrm{R}\mathbf{0}, 31)$	00101 **000** *00011111*	281F	006
IT if $>$ then	10111111 **1000** *1000*	BF88	008
UDF Undefined Instruction exception	11011110 **00000000**	DE00	00A

The rest of the interpreter's code can be divided into 32 subprograms (*i.e.*, 32 sequences of instructions), one for each bytecode instruction. These subprograms can be put one after the other, in increasing order of opcodes. First the subprogram for `cst_0`, then the one for `cst_1`, etc up to the subprogram for `blx` (see Figure 8.6). The i^{th} subprogram could test if the opcode in R0 is equal to i. If it is, it would execute its sequence of instructions. Otherwise, it would jump to the next subprogram, which would test if the opcode is equal to $i + 1$, and so on. But doing so would be quite inefficient. Indeed, for a `blx` instruction for instance, we would need to do 32 comparisons to finally find which subprogram to execute. A shorter and more efficient method is to use a TBB instruction:

TBB $\mathrm{PC} \leftarrow \mathrm{PC} + 2 * \mathrm{mem8}[\mathrm{R}\mathbf{15} + \mathrm{R}0]$	111010001101 **1111**	E8DF	00C
	111100000000 *0000*	F000	00E

Indeed, this instruction jumps by an offset which is twice the i^{th} byte after the instruction itself, with $i = \mathrm{R}0$. For instance, if this instruction is followed by the bytes $[42, 13, 17, 21, \ldots]$, then it jumps by an offset $2 * 42$ if R0 is 0, by an offset $2 * 13$ if it is 1, $2 * 17$ if it is 2, and so on. We can thus add a table with 32 bytes after this instruction, corresponding to the offsets between this instruction and each of the 32 subprograms. We don't have the subprograms yet, so we can't compute these offsets for now. Instead, we leave space for them (32 bytes) and start implementing the subprograms.

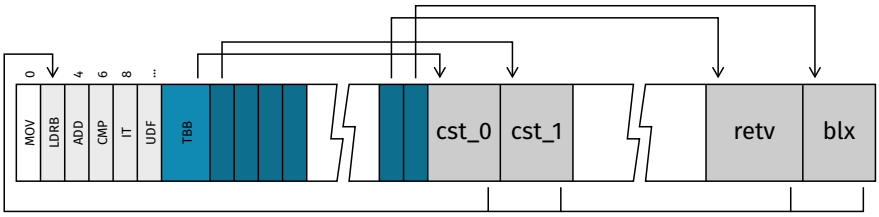


FIGURE 8.6 The interpreter code structure. After the initialization code (white) and the code for step 1 and 2 of Algorithm 8.1 (light gray), a TBB instruction uses a table of 32 offsets (blue) to jump to the subprogram (dark gray) corresponding to the current instruction. Each of the 32 subprograms then jumps back to step 1.

8.3.5 Arithmetic and logic instructions

cst_0 The `cst_0` subprogram, at offset 30_{16} (32 bytes after the TBB instruction), just needs to push 0 on the stack. This is easy to do with MOV and PUSH instructions. It must then go back to step 1 of Algorithm 8.1, *i.e.*, at offset 2_{16}. This can be done with a B instruction using the negative offset $2_{16} - (34_{16} + 4) = -54$:

MOV	R**0** ← *0*	00100**000***00000000*	2000	030
PUSH	R**0** → stack	10110100**00000001**	B401	032
B	PC ← PC + 2 ∗ **2021** − 4096	11100**11111100101**	E7E5	034

cst_1 The `cst_1` subprogram is similar:

MOV	R**0** ← *1*	00100**000***00000001*	2001	036
PUSH	R**0** → stack	10110100**00000001**	B401	038
B	PC ← PC + 2 ∗ **2018** − 4096	11100**11111100010**	E7E2	03A

cst8 This subprogram needs to push the 8-bit value following the instruction's opcode. This can be done by replacing the MOV instruction with a LDRB instruction, to read the byte at the address given by the IC (remember that IC was incremented in step 1, for this purpose). Finally, before going back to step 1, it must increment the IC again, to point to the next bytecode instruction:

LDRB	R**0** ← mem8[R*2* + 0]	01111**00000***010***000**	7810	03C
PUSH	R**0** → stack	10110100**00000001**	B401	03E
ADD	R**2** ← R**2** + *1*	00110**010***00000001*	3201	040
B	PC ← PC + 2 ∗ **2014** − 4096	11100**11111011110**	E7DE	042

cst This subprogram is similar, but reads a word instead of a byte:

LDR	R**0** ← mem32[R*2* + 4 ∗ 0]	01101**00000***010***000**	6810	044
PUSH	R**0** → stack	10110100**00000001**	B401	046
ADD	R**2** ← R**2** + *4*	00110**010***00000100*	3204	048

B	PC ← PC + 2 ∗ **2010** − 4096	11100**11111011010**	E7DA	04A

add This subprogram pops two values, adds them, and pushes the result:

POP	R**0** R**1** ← stack	1011110*0***00000011**	BC03	04C
ADD	R**1** ← R*1* + R0	0001100000*001***001**	1809	04E
PUSH	R**1** → stack	1011010*0***00000010**	B402	050
B	PC ← PC + 2 ∗ **2006** − 4096	11100**11111010110**	E7D6	052

sub This subprogram is similar, but the order in which the registers are subtracted is important. If the top two stack values are x and y, as in Figure 8.2, popping them in R0 and R1 stores y in R0 and x in R1 (cf. Section 7.3.2). Hence, R0 must be subtracted from R1 to get the desired result $x - y$:

POP	R**0** R**1** ← stack	1011110*0***00000011**	BC03	054
SUB	R**1** ← R*1* − R0	0001101000*001***001**	1A09	056
PUSH	R**1** → stack	1011010*0***00000010**	B402	058
B	PC ← PC + 2 ∗ **2002** − 4096	11100**11111010010**	E7D2	05A

mul, div, and, or, lsl, lsr These subprograms are similar, with the SUB instruction replaced with MUL, UDIV, ORR, LSL, and LSR, respectively. Here we just give the result of their encoding:

```
B4024001 BC03E7C9 B402F1F0 FBB1BC03 E7CEB402 4341BC03   05C
B40240C1 BC03E7BD B4024081 BC03E7C1 B4024301 BC03E7C5   074
                                             E7B9       08C
```

8.3.6 Jump instructions

iflt This subprogram must first pop two values x and y and compare them (the order in which the registers are compared is important):

POP	R**0** R**1** ← stack	1011110*0***00000011**	BC03	08E
CMP	*compare*(R**1**, R*0*)	0100001010*000***001**	4281	090

Then there are two cases. If $x < y$ it needs to jump, *i.e.*, to update the IC to the Function Address (in R4), plus the 16-bit offset stored just after the instruction opcode. This offset can be read with a LDRH instruction, and then added to the FA to get the new IC. If $x \geq y$, it just needs to increment the IC by 2 to skip the offset and go to the next instruction. All this can be done with an IT instruction making the 3 following instructions conditional on $x < y$, $x < y$, and $x \geq y$, respectively:

IT	if < then *then else*	10111111**001***1*0*1*0	BF3A	092
LDRH	R**0** ← mem16[R*2* + 2 ∗ 0]	10001**00000***010***000**	8810	094
ADD	R**2** ← R*4* + R0	0001100000*100***010**	1822	096

ADD	$\mathrm{R}\mathbf{2} \leftarrow \mathrm{R}\mathbf{2} + \mathit{2}$	00110**010***00000010*	3202	098
B	$\mathrm{PC} \leftarrow \mathrm{PC} + 2 * \mathbf{1970} - 4096$	11100**11110110010**	E7B2	09A

ifeq, ifgt, ifle, ifne, ifge These subprograms are similar, with only the IT instruction changing (and the offset of the B instruction). We just give the result of their encoding:

```
18228810 BF864281 BC03E7AB 32021822 8810BF06 4281BC03   09C
BF1A4281 BC03E79D 32021822 8810BF9A 4281BC03 E7A43202   0B4
    E78F 32021822 8810BF26 4281BC03 E7963202 18228810   0CC
```

goto This subprogram is simpler since it does an unconditional jump:

LDRH	$\mathrm{R}\mathbf{0} \leftarrow \mathrm{mem16}[\mathrm{R}\mathit{2} + 2 * 0]$	10001 00000 *010* **000**	8810	0E2
ADD	$\mathrm{R}\mathbf{2} \leftarrow \mathrm{R}\mathit{4} + \mathrm{R0}$	0001100 000 *100* **010**	1822	0E4
B	$\mathrm{PC} \leftarrow \mathrm{PC} + 2 * \mathbf{1932} - 4096$	11100**11110001100**	E78C	0E6

8.3.7 Memory and stack instructions

load This subprogram pops an address, reads the memory at this address, and pushes the value read:

POP	$\mathrm{R}\mathbf{1} \leftarrow \mathrm{stack}$	1011110 *0* **00000010**	BC02	0E8
LDR	$\mathrm{R}\mathbf{0} \leftarrow \mathrm{mem32}[\mathrm{R}\mathit{1} + 4 * 0]$	01101 00000 *001* **000**	6808	0EA
PUSH	$\mathrm{R}\mathbf{0} \rightarrow \mathrm{stack}$	1011010 *0* **00000001**	B401	0EC
B	$\mathrm{PC} \leftarrow \mathrm{PC} + 2 * \mathbf{1928} - 4096$	11100**11110001000**	E788	0EE

store This subprogram pops an address and a value, and stores the value at this address (as above, the order in which the registers are popped is important):

POP	$\mathrm{R}\mathbf{0}\ \mathrm{R}\mathbf{1} \leftarrow \mathrm{stack}$	1011110 *0* **00000011**	BC03	0F0
STR	$\mathrm{R}\mathbf{0} \rightarrow \mathrm{mem32}[\mathrm{R}\mathit{1} + 4 * 0]$	01100 00000 *001* **000**	6008	0F2
B	$\mathrm{PC} \leftarrow \mathrm{PC} + 2 * \mathbf{1925} - 4096$	11100**11110000101**	E785	0F4

ptr This subprogram must read the byte after the instruction's opcode, containing an index i. It must then subtract $4 * i = i \ll 2$ from the Frame Pointer (in R5), to get the address of the i^{th} 32-bit value in the top stack frame:

LDRB	$\mathrm{R}\mathbf{0} \leftarrow \mathrm{mem8}[\mathrm{R}\mathit{2} + 0]$	01111 00000 *010* **000**	7810	0F6
LSL	$\mathrm{R}\mathbf{0} \leftarrow \mathrm{R}\mathit{0} \ll 2$	00000 00010 *000* **000**	0080	0F8
SUB	$\mathrm{R}\mathbf{0} \leftarrow \mathrm{R}\mathit{5} - \mathrm{R0}$	0001101 000 *101* **000**	1A28	0FA

Finally, it must push this address a on the stack, and increment the IC to go the next instruction:

PUSH	$\mathrm{R}\mathbf{0} \rightarrow \mathrm{stack}$	1011010 *0* **00000001**	B401	0FC
ADD	$\mathrm{R}\mathbf{2} \leftarrow \mathrm{R}\mathbf{2} + \mathit{1}$	00110**010***00000001*	3201	0FE

B	PC ← PC + 2 ∗ **1919** − 4096	11100**11101111111**	E77F	100

get This subprogram starts with the exact same LDRB, LSL, and SUB instructions (not shown), but then reads the memory at a and pushes the value read, instead of pushing a:

LDR	R**1** ← mem32[R*0* + 4 ∗ 0]	01101 00000 *000* **001**	6801	108
PUSH	R**1** → stack	1011010 *0* **00000010**	B402	10A
ADD	R**2** ← R**2** + *1*	00110 **010** *00000001*	3201	10C
B	PC ← PC + 2 ∗ **1912** − 4096	11100**11101111000**	E778	10E

set This subprogram also starts with the exact same LDRB, LSL, and SUB instructions (not shown), but then pops a value and stores it at address a:

POP	R**1** ← stack	1011110 *0* **00000010**	BC02	116
STR	R**1** → mem32[R*0* + 4 ∗ 0]	01100 00000 *000* **001**	6001	118
ADD	R**2** ← R**2** + *1*	00110 **010** *00000001*	3201	11A
B	PC ← PC + 2 ∗ **1905** − 4096	11100**11101110001**	E771	11C

pop This subprogram just pops a value and discards it:

POP	R**0** ← stack	1011110 *0* **00000001**	BC01	11E
B	PC ← PC + 2 ∗ **1903** − 4096	11100**11101101111**	E76F	120

8.3.8 Function instructions

fn This subprogram starts by saving the RA, FA, FP and BP registers. It then updates the Backup Pointer to point to this new group of saved registers:

PUSH	R**3** R**4** R**5** R**6** → stack	1011010 *0* **01111000**	B478	122
MOV	R**6** ← R*13*	01000110 **0** *1101* **110**	466E	124

The FA and FP can then be updated too. The FA must be set to the `fn` instruction's address, IC - 1 (since IC was incremented in step 1):

MOV	R**4** ← R*2*	01000110 **0** *0010* **100**	4614	126
SUB	R**4** ← R**4** − *1*	00111 **100** *00000001*	3C01	128

The FP must be set to the bottom of the stack frame, initially at offset $4 * (n - 1)$ from the SP (just copied in the BP), but now at offset $4 * (n - 1) + 16 = n \ll 2 + 12$ since 4 values were just pushed (where n is the byte just after the `fn` opcode):

LDRB	R**0** ← mem8[R*2* + 0]	01111 00000 *010* **000**	7810	12A
LSL	R**0** ← R*0* ≪ 2	00000 00010 *000* **000**	0080	12C
ADD	R**0** ← R**0** + *12*	00110 **000** *00001100*	300C	12E
ADD	R**5** ← R*6* + R0	0001100 000 *110* **101**	1835	130

Finally, the `fn` subprogram increments the IC to the beginning of the next instruction, and jumps back to step 1:

ADD	R**2** ← R**2** + *1*	00110**010***00000001*	3201	132
B	PC ← PC + 2 ∗ **1893** − 4096	11100**11101100101**	E765	134

call This subprogram must update the IC to the 16-bit offset after this opcode, added to $C0000_{16}$ (which can be done with a MOVT). Before this, it needs to set the Return Address (in R3) to the address of the next instruction, IC + 2:

MOV	R**3** ← R*2*	01000110**0***0010***011**	4613	136
ADD	R**3** ← R**3** + *2*	00110**011***00000010*	3302	138
LDRH	R**2** ← mem16[R*2* + 2 ∗ 0]	10001**00000***010***010**	8812	13A
MOVT	R**2**[31..16] ← *12*	11110*0*101100*0000*	F2C0	13C
		0*000***0010***00001100*	020C	13E
B	PC ← PC + 2 ∗ **1887** − 4096	11100**11101011111**	E75F	140

callr This subprogram is similar, with the MOVT replaced with two SUB instructions, to update the IC to the instruction's address (IC - 1), minus its 16-bit offset argument:

MOV	R**3** ← R*2*	01000110**0***0010***011**	4613	142
ADD	R**3** ← R**3** + *2*	00110**011***00000010*	3302	144
LDRH	R**0** ← mem16[R*2* + 2 ∗ 0]	10001**00000***010***000**	8810	146
SUB	R**2** ← R**2** − *1*	00111**010***00000001*	3A01	148
SUB	R**2** ← R*2* − R0	0001101**000***010***010**	1A12	14A
B	PC ← PC + 2 ∗ **1881** − 4096	11100**11101011001**	E759	14C

calld This subprogram is simpler since the Return Address is just the IC, and the new IC value is just popped from the stack:

MOV	R**3** ← R*2*	01000110**0***0010***011**	4613	14E
POP	R**2** ← stack	1011110*0***00000100**	BC04	150
B	PC ← PC + 2 ∗ **1878** − 4096	11100**11101010110**	E756	152

ret This subprogram must pop the top stack frame, *i.e.*, set the SP to FP + 4. It must also restore the saved registers by popping them, *after* FP + 4 has been computed (otherwise the *caller's* FP would be used to update the SP, which is incorrect). To this end, the subprogram first computes FP + 4 in R0, then pops the saved registers (which requires setting the SP to the BP first), and finally sets the SP to R0:

MOV	R**0** ← R*5*	01000110**0***0101***000**	4628	154
ADD	R**0** ← R**0** + *4*	00110**000***00000100*	3004	156
MOV	R**13** ← R*6*	01000110**1***0110***101**	46B5	158
POP	R**2** R**4** R**5** R**6** ← stack	1011110*0***01110100**	BC74	15A

MOV	R**13** ← R*0*	01000110**1***0000***101**	4685	15C

Note that the POP instruction pops the saved Return Address in the Instruction Counter (in R2). Everything is thus ready at this point to back to step 1. Before this, however, the subprogram tests if the BP is 0 and, if so, returns from the interpreter (as explained in Section 8.3.3), by moving the LR into the PC:

CMP	*compare*(R**6**, *0*)	00101**110***00000000*	2E00	15E
IT	if = then	10111111**0000***1000*	BF08	160
BX	PC ← R**14** − 1	010001110**1110**000	4770	162
B	PC ← PC + 2 ∗ **1869** − 4096	11100**11101001101**	E74D	164

retv This subprogram is similar, but pops the return value (in R1) and pushes it again before and after doing the same 5 instructions as the `retv` subprogram. The next chapters don't need a main function returning a value from the interpreter. This subprogram thus goes back to step 1 without checking if BP is 0[3]:

POP	R**1** ← stack	1011110*0***00000010**	BC02	166
MOV	R**0** ← R*5*	01000110**0***0101***000**	4628	168
ADD	R**0** ← R**0** + *4*	00110**000***00000100*	3004	16A
MOV	R**13** ← R*6*	01000110**1***0110***101**	46B5	16C
POP	R**2** R**4** R**5** R**6** ← stack	1011110*0***01110100**	BC74	16E
MOV	R**13** ← R*0*	01000110**1***0000***101**	4685	170
PUSH	R**1** → stack	1011010*0***00000010**	B402	172
B	PC ← PC + 2 ∗ **1861** − 4096	11100**11101000101**	E745	174

blx Finally, the last subprogram does a Branch with Link and Exchange to an address popped from the stack. Since the BLX instruction updates the LR, which we need to return from the interpreter, we save it first and restore it upon return (the callee is responsible for saving and restoring the interpreter registers):

POP	R**0** ← stack	1011110*0***00000001**	BC01	176
MOV	R**1** ← R*14*	01000110**0***1110***001**	4671	178
BLX	PC ← R**0** − 1, LR ← *a* + 3	010001111**0000**000	4780	17A
MOV	R**14** ← R*1*	01000110**1***0001***110**	468E	17C
B	PC ← PC + 2 ∗ **1856** − 4096	11100**11101000000**	E740	17E

8.3.9 Final code

The last step to finish the interpreter is to compute the 32 values for the TBB table. Each value is half the difference between the address of the first instruction of a

[3]This could be done without additional instructions, by jumping to `ret`'s CMP instruction instead.

subprogram and the start of the table, 10_{16}. For instance, for the `blx` subprogram starting at 176_{16}, we get $B3_{16}$. By doing this for the 31 other subprograms we get:

```
80797370 6C69625B 544D463F 3B37332F 2A26221E 1A161310   010
                                    B3ABA29F 99938987   028
```

By putting everything together, we finally get the full interpreter code:

```
2A26221E 1A161310 F000E8DF DE00BF88 281F3201 78102600   000
B3ABA29F 99938987 80797370 6C69625B 544D463F 3B37332F   018
B4016810 E7DE3201 B4017810 E7E2B401 2001E7E5 B4012000   030
4341BC03 E7D2B402 1A09BC03 E7D6B402 1809BC03 E7DA3204   048
BC03E7C5 B4024001 BC03E7C9 B402F1F0 FBB1BC03 E7CEB402   060
BC03E7B9 B40240C1 BC03E7BD B4024081 BC03E7C1 B4024301   078
32021822 8810BF06 4281BC03 E7B23202 18228810 BF3A4281   090
8810BF9A 4281BC03 E7A43202 18228810 BF864281 BC03E7AB   0A8
4281BC03 E7963202 18228810 BF1A4281 BC03E79D 32021822   0C0
E788B401 6808BC02 E78C1822 8810E78F 32021822 8810BF26   0D8
1A280080 7810E77F 3201B401 1A280080 7810E785 6008BC03   0F0
BC01E771 32016001 BC021A28 00807810 E7783201 B4026801   108
4613E765 32011835 300C0080 78103C01 4614466E B478E76F   120
4613E759 1A123A01 88103302 4613E75F 020CF2C0 88123302   138
BC02E74D 4770BF08 2E004685 BC7446B5 30044628 E756BC04   150
E740468E 47804671 BC01E745 B4024685 BC7446B5 30044628   168
```

Lets store it in the 2^{nd} flash memory bank, at address $C0000_{16}$. For this we can use the `flash_helper.py` program and its `flash#` command, presented in Section 7.5.4. To avoid typing the 96 necessary `W` commands, we provide them in `part2/interpreter.txt` (you should have this file in the same directory as the `flash_helper.py` program if you downloaded https://ebruneton.github.io/toypc/scripts.zip). Connect the Arduino to your computer and open a terminal. Then run the commands in this file as follows (the < operator makes `flash_helper.py` read and execute all the commands in the specified file; in this mode these commands are not printed):

```
user@host:~$ python3 flash_helper.py < part2/interpreter.txt
>Reading page 1024... Done.
Reading page 1025... Done.
Writing page 1024... Done.
Writing page 1025... Done.
>Done.
```

8.4 Example program

We can now program the Arduino with bytecode instructions, which should be easier than with Cortex M3 machine code. To verify this, we can reimplement the program

to blink a LED from Section 7.5, but in bytecode. We can use the same structure, with a function to wait for some time, and a main function calling it. Lets start with the waiting function, which just counts to some large value. Here it is a bit easier to count down to 0, as follows:

1. push a large value on the stack,
2. subtract 1 from the top stack value,
3. if it is not 0, go back to step 2,
4. return to the caller.

This translates directly to bytecode instructions. We start the function with a `fn 0` instruction (this function does not have any argument), and then push $100000{=}186A0_{16}$ with a `cst` instruction (the right column is the offset from the current function's start or, for `fn` instructions, from the program's start):

instruction		encoding (right to left)	offset
fn	`0`	`00` **19**	`00000`
cst	`186A0`	`000186A0` **03**	`+002`

Step 2 can done by pushing 1 on the stack and then subtracting the top two stack values (which at this point are the counter and 1):

cst_1		**01**	`+007`
sub		**05**	`+008`

For step 3, to compare the counter with 0 and optionally jump back to step 2 (*i.e.*, at offset 7), we could push a 0 and use an `ifne` instruction. But this would pop our counter from the stack, which would then be lost. To avoid this we need to push a copy of it before pushing 0. The counter is the 4^{th} value on the function's stack frame, counting from 0 (because there are no function arguments but 4 saved registers below it). We can thus push a copy of it with a `get` 4 instruction:

get	`4`	`04` **16**	`+009`
cst_0		**00**	`+00B`
ifne	`7`	`0007` **10**	`+00C`

Finally, the last step is trivial:

ret		**1D**	`+00F`

This function should probably look simpler to you compared with the equivalent machine code in Section 7.5. In particular for the encoding of instructions. The main function is even simpler. Recall that it must first write the value 2^{27} in the PIO B Enable Register ($400E1000_{16}$), Output Enable Register ($400E1010_{16}$), and Pull-up Disable Register ($400E1060_{16}$). This is trivial to do with `cst` and `store` instructions:

fn	`0`	`00` **19**	`00010`
cst	`400E1000`	`400E1000` **03**	`+002`
cst	`8000000`	`08000000` **03**	`+007`
store		**14**	`+00C`

```
cst    400E1010                    400E1010 03   +00D
cst    8000000                     08000000 03   +012
store                                      14   +017
cst    400E1060                    400E1060 03   +018
cst    8000000                     08000000 03   +01D
store                                      14   +022
```

After that the main function must turn the LED on by writing the same value in the Set Output Data Register ($400E1030_{16}$), call the waiting function, turn the LED off by writing 2^{27} in the Clear Output Data Register $400E1034_{16}$, call the waiting function again, and finally go back to the beginning. Again, this translates directly to bytecode instructions which are easy to encode (we assume that the waiting function is stored at address 20071000_{16}):

```
cst    400E1030                    400E1030 03   +023
cst    8000000                     08000000 03   +028
store                                      14   +02D
cst    20071000                    20071000 03   +02E
calld                                      1C   +033
cst    400E1034                    400E1034 03   +034
cst    8000000                     08000000 03   +039
store                                      14   +03E
cst    20071000                    20071000 03   +03F
calld                                      1C   +044
goto   23                                0023 12   +045
```

This gives the final bytecode of our new LED blinking program:

```
03400E10 00030019 1D000710 00041605 01000186 A0030019   000
00000340 0E106003 14080000 0003400E 10100314 08000000   018
0E103403 1C200710 00031408 00000003 400E1030 03140800   030
               0023121C 20071000 03140800 00000340   048
```

Lets store it in RAM in order to test it with the boot assistant. The commands to do this are provided in `part2/interpreter_blink_led.txt`. Run them as follows:

```
user@host:~$ python3 flash_helper.py < part2/interpreter_blink_led.txt
>Done.
```

To run our program we need to write a bit more machine code. Indeed, we need to load the address of the main function's first instruction (20071000_{16}+ 10_{16}) in the Instruction Counter (R2), and then call the interpreter subroutine. The former can be done with a LDR instruction. The latter can be done with a LDR instruction to load the interpreter's interworking address ($C0000_{16}$+1) in R0, followed by a BX R0. Assuming these instructions start at an address which is a multiple of 4, we get:

LDR R**2** $\leftarrow$ mem32[$\lfloor$PC$\rfloor_4 + 4 * 1$] | 01001**010***00000001* | 4A01 | 000

LDR R**0** $\leftarrow$ mem32[$\lfloor$PC$\rfloor_4 + 4 * 2$] | 01001**000***00000010* | 4802 | 002

BX	PC $\leftarrow$ R**0** $-$ 1	0100011100000000	4700	004
data	(padding, unused)		0000	006
data	(address of main function's 1^{st} instruction)		20071010	008
data	(interworking address of interpreter)		000C0001	00C

We can store these instructions at 20071100_{16} and finally call them with a Ga# command (recall that Ga# jumps to the interworking address stored at $a + 4$):

```
user@host:~$ python3 boot_helper.py
>W20071100,48024A01#
>W20071104,00004700#
>W20071108,20071010#
>W2007110C,000C0001#
>W20071204,20071101#
>G20071200#
ERROR: no response from device.
```

At this point you should normally see the LED blinking! Note that it blinks at about the same speed as the machine code version running from the boot assistant. Yet it counts to 100000 instead of counting to 1 million between each step. This shows that our bytecode program is about 10 times slower than the equivalent machine code version! This is not an issue here because our interpreter is only a temporary program, similar to a scaffolding, to help us build our toy computer. Indeed, we discard it at the end of Chapter 25.

CHAPTER

9 Clock Driver

Thanks to the virtual machine implemented in the previous chapter we can now start to implement our basic input output system, in a much simpler way than with machine code. Recall that our goal is to make the Arduino completely autonomous, with its own keyboard and screen to write programs and run them (instead of needing a host computer and a USB cable). For this we need small programs to interact with the keyboard and the screen, called drivers. Before writing them, however, it is useful to have a *clock driver*[1]. That is, some functions to configure the Arduino's clock and to wait for some time in a more precise way than with a software counter.

This chapter presents the SAM3X8E components which are needed for this, namely the Power Management Controller and the System Timer. We give an overview of these components, and explain how programs can use them. We then use this knowledge to implement the above clock driver functions. Finally, we test these functions with a third version of our LED blinking program.

9.1 Power Management Controller

The Power Management Controller (PMC) component generates clock signals for the other components of the microcontroller, including the microprocessor and the peripheral controllers (see Figure 6.2). It can turn these clock signals on and off, and can change their frequency. The higher the clock frequency, the more power is consumed, but components whose clock signal is off do not consume power. The Power Management Controller thus indirectly controls how much power is consumed, hence its name.

The clock signals are generated by an oscillator circuit at a fixed frequency, but can optionally go through other circuits which can multiply this frequency by a configurable factor. The PMC has several such circuits. The ones used in this book are represented in Figure 9.1:

- The Resistor Capacitor (RC) is an oscillator circuit providing a 4MHz clock signal by default (it can be configured to output 4, 8 or 12MHz). It is integrated in the SAM3X8E chip.

[1]This is not strictly necessary. The clock driver could be written only once the Arduino is autonomous, with the Arduino's keyboard and screen, but this would be a bit more complex.

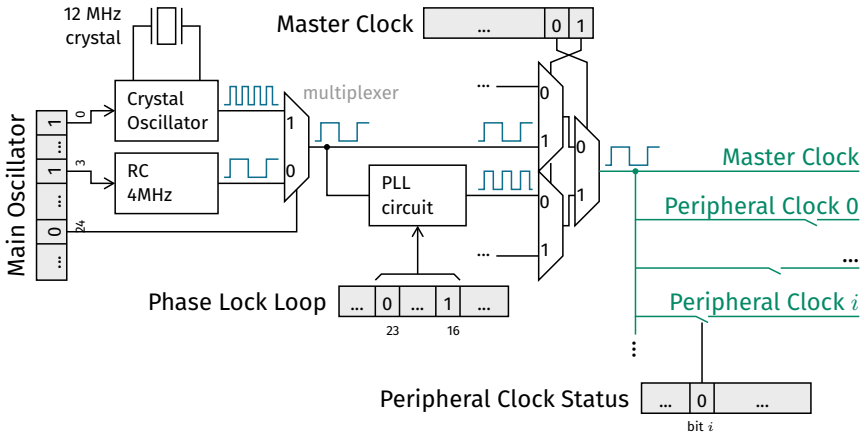


FIGURE 9.1 A simplified representation of the Power Management Controller (PMC) circuit and registers (in gray). The PMC provides clock signals (in green) to the other components, generated from a selectable oscillator (left) at a frequency optionally multiplied by a configurable factor (middle). See Figures 28-1 and 28-2 in [8].

- The Crystal Oscillator uses an external crystal on the Arduino board to generate a 12MHz clock signal. This signal is much more stable than the one provided by the Resistor Capacitor oscillator.
- The Phase Lock Loop (PLL) circuit can multiply the frequency output by the RC or by the Crystal oscillator by a configurable factor, up to 2048. Note however that the SAM3X8E components only support frequencies up to 84MHz. Higher frequencies can damage them permanently.

These circuits are controlled by several registers. Some of them are shown in Figure 9.1. The ones used in this book are presented below and in Table 9.1. The full list of the PMC registers and its full diagram can be found in Chapter 28 of [8].

9.1.1 Main Oscillator Register

This register enables or disables the oscillator circuits, and selects which oscillator to use. It has the following binary format (we present only the bits that we use):

0	0	0	0	0	0	0	s	$password$	$startup$	0	0	0	0	c	0	0	r

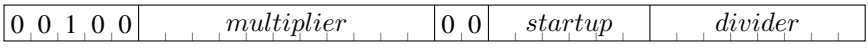
- r enables the RC oscillator if it is 1, and disables it otherwise.
- c enables the Crystal Oscillator if it is 1, and disables it otherwise.
- $startup$ specifies the Crystal Oscillator start-up time, in quarters of milliseconds.
- $password$ must be 37_{16}, otherwise writing into this register has no effect.
- s selects the Crystal oscillator if it is 1, or the RC oscillator otherwise.

Name	Type	Address
Peripheral Clock Enable Register	Write-Only	$400E0610_{16}$
Peripheral Clock Disable Register	Write-Only	$400E0614_{16}$
Peripheral Clock Status Register	Read-Only	$400E0618_{16}$
Main Oscillator Register	Read-Write	$400E0620_{16}$
Phase Lock Loop Register	Read-Write	$400E0628_{16}$
Master Clock Register	Read-Write	$400E0630_{16}$
Status Register	Read-Only	$400E0668_{16}$

TABLE 9.1 The Power Management Controller registers used in this book.

9.1.2 Phase Lock Loop Register

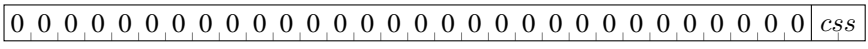
This register defines the frequency multiplication factor to use in the Phase Lock Loop circuit. It has the following binary format:



- $multiplier$ and $divider$ configure the frequency multiplication factor. The PLL circuit multiplies the clock frequency by the fraction $(multiplier + 1)/divider$. To use the PLL output as a clock signal, $divider$ must not be 0.
- $startup$ specifies the PLL circuit start-up time, in quarters of milliseconds.

9.1.3 Master Clock Register

This register defines which signal to use for the Master Clock, used by the microprocessor and optionally by the peripheral controllers (see Figure 9.1). It has the following binary format (we present only the bits that we use):



The clock source selection (css) field can take four possible values, but we use only two. If it is equal to 1, then the Master Clock is the signal from the oscillator selected by the Main Oscillator Register. If it is 2 then the Master Clock is the output of the PLL circuit.

9.1.4 Peripheral Clock Status Register

This register gives the clock signal status of each peripheral component. Peripherals are numbered, for instance the USART controller has number 17, the SPI controller has number 24, etc (see Figure 6.2). Bit number i of this register indicates if the clock signal is enabled for peripheral number i. If enabled, the same signal as the Master Clock is used (see Figure 9.1). As the PIO Status Register, this register can only be read. Writing to it has no effect. Instead, one must write in the Peripheral Clock Enable Register (resp. the Peripheral Clock Disable Register) to set (resp. clear) its bits.

9.1.5 Status Register

This read-only register indicates if the various clock signals inside the Power Management Controller are ready to use or not. It has the following binary format (we present only the bits that we use):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	s	0	0	0	0	0	0	0	0	0	0	0	m	0	p	c

- c is 1 if the Crystal Oscillator is ready, or 0 otherwise. After the Crystal Oscillator is enabled with the Main Crystal Oscillator Register, one must wait until this bit is 1 before using the PMC again.
- p is 1 if the PLL circuit output is ready, or 0 otherwise. After the frequency multiplier is changed with the Phase Lock Loop Register, one must wait until this bit is 1 before using the PMC again.
- m is 1 if the Master Clock is ready, or 0 otherwise. After the clock source selection is changed with the Master Clock Register, one must wait until this bit is 1 before using the PMC again.
- s is 1 if the oscillator selection is ready, or 0 otherwise. After the selected oscillator is changed (to the crystal or the RC oscillator), one must wait until this bit is 1 before using the PMC again.

9.1.6 Configuration procedure

After a reset the PMC is configured to use the RC oscillator directly, without going through the PLL circuit, and all the peripheral clocks are disabled. The Master Clock thus runs at 4MHz by default. To use the maximum 84MHz frequency instead, the following procedure can be used:

1. enable the 12MHz Crystal Oscillator and wait for it to be ready,
2. select the Crystal Oscillator and wait for this selection to be ready,
3. set the PLL multiplier to 6 and its divider to 1, and wait for it to be ready,
4. select the PLL output as the Master Clock source, and wait for it to be ready.

Before doing this, however, the flash controllers must be configured as well. Indeed the flash memory is slow. When the microprocessor sends a request to read a value from it, it has to wait some time before it receives this value. And the higher the clock frequency, the more it has to wait. This waiting time is configured with the flash controller Mode Register (see Table 6.1). It has the following binary format (we present only the bits that we use):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$wait$	0	0	0	0	0	0	0	0

The $wait$ field controls the above waiting time. According to Table 45-62 in [8], it must be at least 4 to use a 84MHz clock. And it must be at least 6 when writing flash memory pages (see Section 49.1.1.1 in [8]).

Name	Type	Address
Control and Status Register	Read-Write	$E000E010_{16}$
Reload Value Register	Read-Write	$E000E014_{16}$
Current Value Register	Read-Write	$E000E018_{16}$

TABLE 9.2 The System Timer registers used in this book.

9.2 System Timer

The System Timer (SysTick) is a 24-bit counter inside the Cortex M3 microprocessor (see Figure 6.2). This counter is decremented by 1 every 8 cycles of the Master Clock, if it is enabled and not 0. If it is enabled and equal to 0, it is instead reset to a configurable value called the *reload value*.

This timer can be used to wait for a calibrated amount of time. For instance, if the Master Clock is set to 84MHz, one can wait for 1 millisecond by resetting the counter to 10500 and by waiting until it is 0 ($10500 * 8$ cycles last $10500 * 8/84.10^6 = 10^{-3}$ s). The System Timer is controlled with 3 registers, presented in Table 9.2:

- The Control and Status Register enables or disables the timer, and indicates if it counted from 1 to 0 since the last time this register was read. It has the following binary format (we present only the bits that we use):

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	z	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	e

  The e field enables the timer if it is 1, and disables it otherwise. The read-only z field indicates if the timer counted to 0 since the last read of this register.
- The Reload Value Register specifies the reload value. Only its 24 least significant bits are used.
- The Current Value Register contains the counter's current value. Writing any value into this register resets this current value to 0 and also clears the z field in the Control and Status Register. The counter is then reset to its reload value, if it is enabled, during one of the next 8 clock cycles.

9.3 Watchdog Timer

Another microcontroller component which is related to time is the Watchdog Timer. Its goal is to detect when the user application is stuck, for instance because of a bug, and to reset the microcontroller if this happens. It does so with a countdown until reset. In order to avoid being reset when the countdown expires, the application must reinitialize it periodically, before it reaches 0. If the application crashes or enters an infinite loop, it cannot do this and will thus eventually be reset. To avoid having to periodically reinitialize the Watchdog Timer, we can simply disable it (it is enabled by default). This can be done by setting to 1 bit 15 of the Watchdog Timer Mode Register, at address $400E1A54_{16}$ (see Chapter 15 in [8]).

9.4 Clock initializer

We can now use the above information to implement our clock driver, in the next flash memory page after the interpreter, *i.e.*, at address $C0200_{16}$. The first goal is to provide a function to set the clock frequency to 84MHz.

```
clock_init()                fn      00              19  C0200
                            cst     400E0A00        03   +002
                            cst     00000600        03   +007
                            store                   14   +00C
                            cst     400E0C00        03   +00D
                            cst     00000600        03   +012
                            store                   14   +017

                            cst     400E0620        03   +018
                            cst     0037FF09        03   +01D
                            store                   14   +022

                            cst     400E0668        03   +023
                            load                    13   +028
                            cst_1                   01   +029
                            and                     08   +02A
                            cst_0                   00   +02B
                            ifeq    0023            0D   +02C

                            cst     400E0620        03   +02F
                            cst     0137FF09        03   +034
                            store                   14   +039
                            cst     400E0668        03   +03A
                            load                    13   +03F
                            cst     00010000        03   +040
                            and                     08   +045
                            cst_0                   00   +046
                            ifeq    003A            0D   +047

                            cst     400E0628        03   +04A
                            cst     20063F01        03   +04F
                            store                   14   +054
                            cst     400E0668        03   +055
```

As explained in Section 9.1.6, this function must configure the flash wait time first. To be conservative, we set the *wait* field to 6 in both EEFC Mode Registers (cf. Table 6.1). We can then implement the 4 steps of the procedure presented in Section 9.1.6.

Step 1 Enable the Crystal Oscillator (while leaving the RC oscillator enabled). For this we must set $c = 1$ and $r = 1$ in the Main Oscillator Register. We don't know the Crystal Oscillator start-up time and thus set *startup* to its maximum value, FF_{16}. With *password* set to 37_{16}, this means that we must write $37FF09_{16}$ at address $400E0620_{16}$.

We then need to wait until the Crystal Oscillator is ready, *i.e.*, until the c bit in the Status Register is 1. For this we read this register, extract its c bit with a bitwise and with 1_{16}, and repeat these steps while the result is 0.

Step 2 Select the Crystal Oscillator. For this we need to set the same value as above in the Main Oscillator Register, with the s field additionally set to 1. That is we must write $137FF09_{16}$. We then need to wait until the s field in the Status Register is 1. This is done as above with a bitwise and with 10000_{16}.

Step 3 Set the PPL *multiplier* to 6 and its *divider* to 1. We don't know the PLL start-up time and thus set *startup* to its maximum value, $3F_{16}$. We then wait

until the Status Register's p bit is 1.

```
load                    13  +05A
cst8  02                02  +05B
and                     08  +05D
cst_0                   00  +05E
ifeq  0055              0D  +05F
```

Step 4 Select the PPL output as the Master Clock source. For this we set css to 2 in the Master Clock Register. We then wait until the Status Register's m bit is 1.

```
cst   400E0630          03  +062
cst8  02                02  +067
store                   14  +069
cst   400E0668          03  +06A
load                    13  +06F
cst8  08                02  +070
and                     08  +072
cst_0                   00  +073
ifeq  006A              0D  +074
```

At this stage the Master Clock runs at 84MHz. To finish this function we can enable the System Timer, disable the Watchdog Timer, and return.

```
cst   E000E010          03  +077
cst8  01                02  +07C
store                   14  +07E
cst   400E1A54          03  +07F
cst   00008000          03  +084
store                   14  +089
ret                     1D  +08A
```

9.5 Delay function

The second goal of our clock driver is to provide a function to wait for a calibrated amount of time. For this we can pass a number $n > 0$ of milliseconds to wait for as an argument to this function. Which can then be implemented by using the System Timer, as follows.

`delay`(n)

```
fn    01                19  C028B
```

Set the System Timer reload value to 10500 times n (10500 = 2904_{16} and n is the stack frame's 0^{th} value).

```
cst   E000E014          03  +002
cst   00002904          03  +007
get   00    n           16  +00C
mul                     06  +00E
store                   14  +00F
```

Reset the System Timer current value.

```
cst   E000E018          03  +010
cst_0                   00  +015
store                   14  +016
```

Wait until the System Timer counts

```
cst   E000E010          03  +017
```

from 1 to 0, *i.e.*, until the Control and Status Register's z bit is 1.

```
load                    13    +01C
cst     00010000        03    +01D
and                     08    +022
cst_0                   00    +023
ifeq    0017            0D    +024
ret                     1D    +027
```

Note that since the System Timer is a 24-bit counter, n is limited to $(2^{24} - 1)/10500 = 1597$. Therefore, a single call to this function can't wait for more than ~ 1.5s. By putting together the encoding of the two above functions we get the final bytecode of our clock driver:

```
14000006 00034E0E 0C000314 00000600 03400E0A 00030019  C0200
03002300 00080113 400E0668 03140037 FF090340 0E062003  C0218
0D000800 01000003 13400E06 68031401 37FF0903 400E0620  C0230
0D000802 02134E0E 06680314 20063F01 03400E06 28030030  C0248
03006A0D 00080802 13400E06 68031402 02400E06 30030055  C0260
E0140301 191D1400 00800003 400E1A54 03140102 E000E010  C0278
13E000E0 10031400 E000E018 03140600 16000029 0403E000  C0290
                           1D0017 0D000800 01000003  C02A8
```

Lets store it in flash memory. To avoid you some typing we provide the necessary boot assistant commands in `part2/clock_driver.txt`. Run them with:

```
user@host:~$ python3 flash_helper.py < part2/clock_driver.txt
>Reading page 1026... Done.
Writing page 1026... Done.
>Done.
```

9.6 Basic input output system foundations

We now have the first elements of our basic input output system stored in flash memory, namely an interpreter and a clock driver. In order to make the Arduino completely autonomous, we must be able to run them automatically after a reset, without the boot assistant. For this we need to setup a Vector Table (cf. Section 7.4).

9.6.1 Vector Table

After a reset we want to initialize the Master Clock with the clock driver and, later on, initialize the screen and keyboard drivers, and start the memory editor. We can do this with a main bytecode function, called upon reset via the Reset handler. In order to leave space for the future drivers, after the clock driver code at C0200_{16}, we can put this main function at address C2000_{16} (see Figure 9.2). The Reset handler should thus call the interpreter with C2000_{16} as initial Instruction Counter. We have already seen how to do that, with 2 LDR and a BX instruction, at the end of the previous

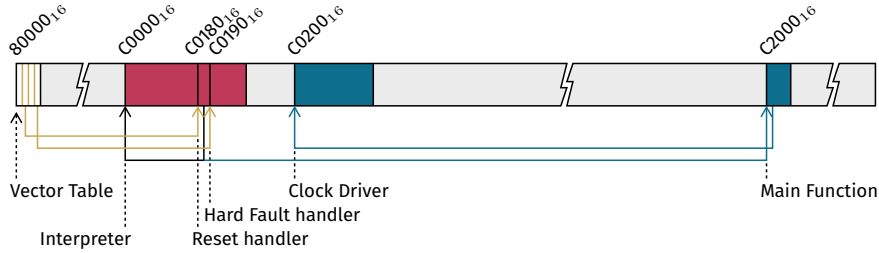


FIGURE 9.2 The initial layout of our basic input output system in flash memory. Execution starts with the Reset handler, which calls the interpreter to execute the Main function. This function then calls other functions, *e.g.*, in the clock driver. Red, blue and gray areas represent machine code, bytecode and unused memory, respectively (not to scale).

chapter. We put the corresponding machine code

```
000C0001 000C2000 00004700 48024A01
```

at address $C0180_{16}$, just after the bytecode interpreter.

To handle potential errors, we should also define a Hard Fault handler. The easiest solution is to make the LED blink in case of error, since we already have a program doing this. The machine code given at the end of Section 7.5.2 can then be directly reused, and put just after the above code, *i.e.*, at offset $C0190_{16}$ (see Figure 9.2).

We also need to define an initial Stack Pointer, in the Vector Table's first entry. Since the stack grows in decreasing address order we initialize it to the end of the contiguous RAM region, 20088000_{16} (see Figure 6.3).

In summary, the above choices lead to the following Vector Table (only the first 4 entries are used; the Hard Fault handler's main function starts at offset 14_{16} after $C0190_{16}$, *i.e.*, at $C01A4_{16}$):

```
000C01A5 00000000 000C0181 20088000
```

9.6.2 Boot mode selection

In the following we would like to test the clock driver, then implement the screen driver and test it, then implement the keyboard driver and test it, etc. For this we need to alternatively boot from flash (to test our system) and from ROM (to store new drivers in flash memory with the boot assistant). However, when the Arduino runs from flash, the only way to boot from ROM again is to do a full erase. This means that, after each test, we would have to flash everything again. To avoid this, as discussed at the end of Section 7.5.4, our main function should set the boot mode selection to boot from ROM. To this end, we provide a function doing this below.

As explained in Section 6.5.2, setting the boot mode to boot from ROM can be done by writing the value $5A00010C_{16}$ in the EEFC0 Command Register at address $400E0A04_{16}$ (and then waiting until the Status Register is 1). After that the ROM is mapped in the "Boot" region, which means that *our Vector Table is no longer mapped*

Function	Address
boot_mode_select_rom()	C02B4 ($C0000_{16}$+692)
clock_init()	C0200 ($C0000_{16}$+512)
delay(n)	C028B ($C0000_{16}$+651)

TABLE 9.3 The basic input output system functions defined in this chapter.

there. In case of error, the ROM Vector Table would thus be used instead of ours. To avoid this, a solution is to first change the location where the Vector Table is read from, with the Vector Table Offset Register (at address $E000ED08_{16}$ – see Section 7.4). More precisely, a solution is to store in this register the "real" address of our Vector Table, 80000_{16}. This leads to the following bytecode function, stored just after the clock driver, at address $C02B4_{16}$ (see Table 9.3):

```
fn      00          19 C02B4 | store           14 +017
cst     E000ED08    03 +002  | cst     400E0A08 03 +018
cst     00080000    03 +007  | load             13 +01D
store               14 +00C  | cst_1            01 +01E
cst     400E0A04    03 +00D  | ifne    0018     10 +01F
cst     5A00010C    03 +012  | ret              1D +022
```

To avoid you some typing, we provide the boot assistant commands necessary to write this function, the above Vector Table, and its Reset and Hard Fault handlers in flash memory. Run them with:

```
user@host:~$ python3 flash_helper.py < part2/foundations.txt
>Reading page 0... Done.
Reading page 1025... Done.
Reading page 1026... Done.
Writing page 0... Done.
Writing page 1025... Done.
Writing page 1026... Done.
>Done.
```

9.7 Experiments

In order to test the above functions we can implement a third version of our LED blinking program where:

- calls to the boot mode selection and clock initializer functions are added at the beginning,
- calls to the waiting function are replaced with calls to the delay function (and the waiting function is deleted).

And to use it as a main function we can put it in flash memory at address $C2000_{16}$. With 500ms delays, we get the following function, based on the version in Section 8.4 ($500 = 1F4_{16}$):

```
fn      00                  19  C2000
call    02B4 …select_rom    1A  +002
call    0200 clock_init     1A  +005
cst     400E1000            03  +008
cst     08000000            03  +00D
store                       14  +012
cst     400E1010            03  +013
cst     08000000            03  +018
store                       14  +01D
cst     400E1060            03  +01E
cst     08000000            03  +023
store                       14  +028
cst     400E1030            03  +029
cst     08000000            03  +02E
store                       14  +033
cst     000001F4            03  +034
call    028B delay          1A  +039
cst     400E1034            03  +03C
cst     08000000            03  +041
store                       14  +046
cst     000001F4            03  +047
call    028B delay          1A  +04C
goto    0029                12  +04F
```

Write it in flash memory with:

```
user@host:~$ python3 flash_helper.py < part2/clock_driver_test.txt
>Reading page 1056... Done.
Writing page 1056... Done.
>Done.
```

To run it we need to change the boot mode selection to boot from flash, and to reset the Arduino. The former can be done by writing $5A00010B_{16}$ at address $400E0A04_{16}$ (cf. Section 6.5.2). And the latter with the RESET button or, with the Reset Controller (see Figure 6.2), by writing $A500000D_{16}$ at address $400E1A00_{16}$ (see Section 12.5.1 in [8]). For convenience, `flash_helper.py` provides a `reset#` command doing these two steps. We can thus run our program with:

```
user@host:~$ python3 flash_helper.py
>reset#
```

At this point you should see the LED blinking once per second[2], without begin reset by the Watchdog timer. Then press the RESET button on the Arduino: the LED should no longer blink, because the Arduino booted from ROM. Finally, lets test our Hard Fault handler by introducing a voluntary error in our LED blinking program. We can do this by replacing the last `goto` insn with an invalid FF_{16} opcode:

```
user@host:~$ python3 flash_helper.py
>WC204C,FF028B1A#
Reading page 1056... Done.
>flash#
Writing page 1056... Done.
>reset#
```

[2]Almost: each cycle lasts 1s plus a few μs due to the instructions between the delay function calls.

You should see the LED blinking once as before (as the beginning of the program executes normally), then blink very fast. Indeed, when the unknown opcode is found, the interpreter triggers an Undefined Instruction exception, which triggers the Hard Fault handler. This calls our machine code LED blinking function, which counts to 1 million between each step. At 84MHz, this yields very fast blinks. You can finally reset and unplug the Arduino.

CHAPTER

10 Graphics Card Driver

So far we have only been able to blink a LED, using various methods. Hopefully, thanks to the work done in the previous chapters, we now have everything we need to go beyond that. This is what we do in this chapter, by connecting a screen to the Arduino, and by writing a program to use it. The screen cannot be connected directly to the Arduino, in particular because their connectors don't match. We therefore use an intermediate component between the two, that we call the graphics card. This chapter presents these two components, explain briefly how they work, how they communicate, how to connect them, and how programs can use them. We then write such a program, called the graphics card driver. Finally, we test it with a small application displaying the traditional "Hello, World!" message.

10.1 Liquid Crystal Display

The screen used in this book is a 7″ Liquid Crystal Display (LCD) with 800×480 pixels (see Table A.1). Each pixel is made of 3 cells with red, green and blue color filters. Each cell is made of a liquid crystal between two electrodes and two perpendicularly oriented light polarizers (see Figure 10.1). The whole screen is lit from behind with LEDs producing a uniform white light with a constant intensity. By default, however, due to the polarizers, light is blocked and the screen appears black. To turn a red, green or blue cell on one must charge the capacitor made by the two electrodes around it. This creates an electric field in the liquid crystal, which has the property of rotating the light's polarization direction in such conditions. Increasing the capacitor charge increases the electric field and the polarization rotation. When the rotation angle is 90°, no light is blocked and the cell has 100% luminosity. Smaller angles lead to partially blocked light thus lower luminosity levels.

In order to charge the capacitors independently from each other, one electrode of each cell is connected to a grid of horizontal and vertical wires via a transistor (see Figure 10.1) – the other electrode is shared between all cells. These transistors are organized in a transparent thin film, hence the Thin Film Transistor (TFT) name. To charge a cell at column x and row y, one must set row y's wire to VCC and then apply a voltage V on column's x red, green or blue wire for a short duration – with V depending on the desired luminosity. This process is done by the row and column driver circuits (see Figure 10.1). For each frame, the row driver sets each row wire to

VCC, one after the other, from top to bottom. While a row is active, the column driver applies the desired voltage on each column wire, one after the other, from left to right.

These circuits use as input 40 pins, providing 40 signals in parallel. The most important ones are 3×8 signals providing the red, green and blue intensity of each pixel, using one byte per color (from 0 = black to 255 = 100% luminosity). There are also vertical, horizontal, and clock synchronization signals indicating when a new frame, a new row of pixels, and a new pixel start, respectively. Finally, other pins provide GND and VCC, a driver circuit on/off signal, a backlight on/off signal, etc. The pixel and synchronization signals must have the form shown in Figure 10.2, subject to the constraints in Table 10.1:

- Each frame must start with a pulse of the vertical synchronization (Vsync) signal.
- Each row must start with a pulse of the horizontal synchronization (Hsync) signal.
- Each frame must start and end with blank rows containing no data, called the vertical back porch and front porch, respectively.
- Each row must start and end with blank pixels containing no data, called the horizontal back porch and front porch, respectively.
- The Data Enable signal must indicate when the Data pins contain actual data.
- The 24 Data pins must send one pixel at each clock cycle where Data Enable is 1.

10.2 Graphics card

The LCD cannot be connected directly to the Arduino, since its does not have a 40-pin connector. An adapter could theoretically be used, since the Arduino has enough pins controllable with the PIO controllers A, B, C and D (see Figure 6.2). Using these controllers, we could in principle write a program driving these pins as output, and producing signals of the above form. However, even if this program could run fast enough, it would require a lot of memory to store the image to display. Even with only one bit per pixel, $800 * 480 = 384000$ bits would be needed, *i.e.*, almost half of the 96 KB of the Arduino's RAM.

For all these reasons we use an intermediate component to connect the screen to the Arduino, namely an AdaFruit RA8875 driver board (see Table A.1). This board provides a RAiO RA8875 chip [13], with a 40-pin connector on one side and slots for 15 header pins on the other (see Figure 10.3). The chip contains 768 KB of RAM, which is enough to store a 800×480 image with 16 bits per pixel. It can be roughly divided in two parts:

- the backend part reads the image stored in RAM and generates corresponding signals on the 40-pin connector, to display it on the screen.
- the frontend part updates the image stored in RAM based on drawing commands received on the header pins.

The frontend can draw text, basic shapes such as lines, rectangles or ellipses, images, and individual pixels. It is based on 8-bit registers. For instance, to draw a line, one must first write the x, y coordinates of its endpoints in specific registers

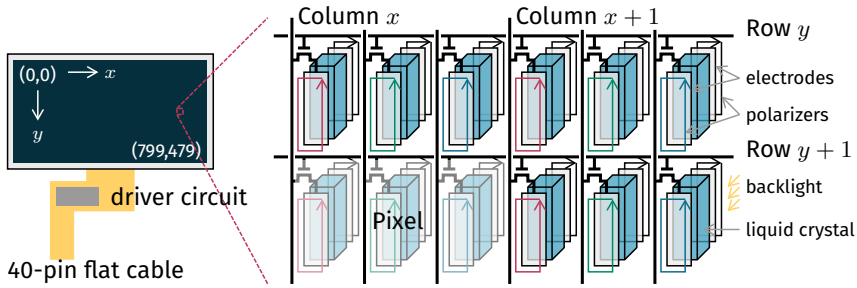


FIGURE 10.1 A schematic view of the Thin Film Transistor (TFT) Liquiq Crystal Display (LCD) used in this book. Each pixel is made of 3 liquid crystal cells between two transparent electrodes and two polarizers, with red, green and blue color filters. Each cell is connected to a grid of wires with its own transistor, and lit from behind.

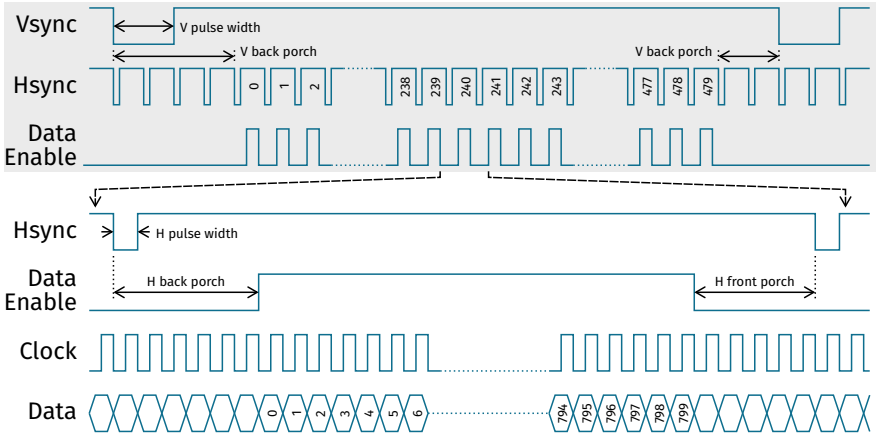


FIGURE 10.2 The input signals needed by the LCD [19]. Each frame starts with a Vsync pulse, and each row with an Hsync pulse. 24 bits of pixel data are required at each clock cycle where "Data Enable" is 1. The bottom part zooms on one row of pixels.

Parameter	Minimum	Typical	Maximum	Unit
Vertical pulse width	1	–	20	Hsync cycle
Vertical back porch	23	23	23	Hsync cycle
Vertical front porch	7	22	147	Hsync cycle
Horizontal pulse width	1	–	40	Clock cycle
Horizontal back porch	46	46	46	Clock cycle
Horizontal front porch	16	210	354	Clock cycle
Clock frequency	–	33.3	50	MHz

TABLE 10.1 The timing constraints of the LCD input signals [19].

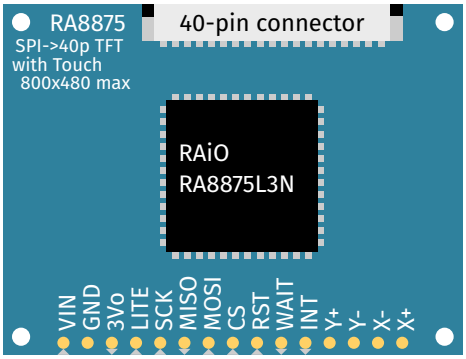


FIGURE 10.3 The AdaFruit RA8875 Driver Board used to connect the LCD to the Arduino.

called "Draw Line/Square Horizontal Start Coordinate", "Draw Line/Square Vertical Start Coordinate", "Draw Line/Square Horizontal End Coordinate", etc. Then, setting a specific bit to 1 in another register called "Draw Line/Circle/Square Control Register" starts drawing a line between the two previous points. The RA8875 has about 170 different registers but we use only a small subset of them, mostly those needed to configure the RA8875, and to draw text. They are presented below.

10.2.1 Configuration registers

The RA8875 chip, hereafter called the graphics card, can be used with a variety of screens with different resolutions and signals timing constraints. Before using it to display images on a given screen, it must be configured for this specific screen.

The first step is to configure the frequency f_s of the internal clock, called the system clock, and the frequency f_p of the clock signal sent to the screen, called the pixel clock. As the Arduino, the graphics card uses an external crystal, here at $f_c = 20$ MHz, and a Phase Lock Loop (PLL) circuit. This circuit can be configured with 3 registers, whose ID, name and binary format are the following:

R88$_{16}$	PLL Control 1	m 0 0 n
R89$_{16}$	PLL Control 2	0 0 0 0 0 0 k
R04$_{16}$	Pixel Clock Setting	i 0 0 0 0 0 p

These registers set f_s to $f_c * (n+1)/[2^k(m+1)]$ and f_p to $f_s/2^p$. In order to get $f_p \in [33.3, 50]$ MHz (see Table 10.1), we can use $n = 6$, $m = 0$, $k = 6$, and $p = 1$, yielding $f_s = 70$ MHz and $f_p = 35$ MHz. In addition, i indicates if pixel data should be sent when the pixel clock signal is rising ($i = 0$) or falling ($i = 1$). According to [19] the former case applies, but in practice only $i = 1$ works. In summary, in our

case, we can set R88, R89 and R04 to 6, 1 and 81_{16}, respectively, with a delay of at least 0.1ms between each step (see Section 5-9 in [19]).

The second step is to configure the LCD signal parameters (see Table 10.1), with the following registers (we show only the bits that we use):

R14$_{16}$	LCD Horizontal Display Width	$0 \mid w$
R15$_{16}$	LCD Horizontal Non-Display Period Fine Tuning	$0\ 0\ 0\ 0 \mid hndft$
R16$_{16}$	LCD Horizontal Non-Display Period	$0\ 0\ 0 \mid hnd$
R17$_{16}$	HSYNC Start Position	$0\ 0\ 0 \mid hsp$
R18$_{16}$	HSYNC Pulse Width	$0\ 0\ 0 \mid hpw$
R19$_{16}$	LCD Vertical Display Height 0	h_0
R1A$_{16}$	LCD Vertical Display Height 1	$0\ 0\ 0\ 0\ 0\ 0\ 0 \mid h_1$
R1B$_{16}$	LCD Vertical Non-Display Period 0	vnd
R1D$_{16}$	VSYNC Start Position 0	vsp
R1F$_{16}$	VSYNC Pulse Width	$0 \mid vpw$

Based on these register values, the graphics card generates LCD signals with the following parameters (see Section 5-3 and Figure 6-29 in [19]):

- Screen width in pixels: $8(w + 1)$
- Horizontal pulse width: $8(hpw + 1)$
- Horizontal back porch: $8(hpw + 1) + 8(hnd + 1) + hndft + 2$
- Horizontal front porch: $8(hst + 1)$
- Screen height in pixels: $256.h_1 + h_0 + 1$
- Vertical pulse width: $vpw + 1$
- Vertical back porch: $(vpw + 1) + (vnd + 1)$
- Vertical front porch: $vst + 1$

In our case, in order to meet the constraints from Table 10.1, we can use $w = 99$, $hpw = 0$, $hndft = 4$, $hnd = 3$, $hst = 25$, $h_0 = 223$, $h_1 = 1$, $vpw = 0$, $vnd = 21$, and $vst = 21$. In other words, we can set R14, R15, R16, R17, R19, R1A, R1B, and R1D to 63_{16}, 4, 3, 19_{16}, DF_{16}, 1, 15_{16}, and 15_{16}, respectively (and leave R18 and R1F to their default value 0).

After this the graphics card memory can be cleared, to start with a black screen, the LCD signals can be enabled, and the screen itself can be turned on. This can be done with the following registers (we show only the bits that we use):

R8E$_{16}$	Memory Clear Control	$s \mid a \mid 0\ 0\ 0\ 0\ 0\ 0$
R01$_{16}$	Power and Display Control	$d \mid 0\ 0\ 0\ 0\ 0\ 0\ 0$
RC7$_{16}$	Extra General Purpose IO	$0\ 0\ 0\ 0\ 0\ 0\ 0 \mid x$
R8A$_{16}$	PWM1 Control	$0 \mid l \mid 0\ 0\ 0\ 0\ 0\ 0$

Indeed, setting s, d, x, and l to 1 clears the memory, enables the LCD signal outputs, turns on the LCD's driver circuit, and turns on the LCD's backlight, respectively[1]. In summary we can set R8E, R01, RC7 and R8A to 80_{16}, 80_{16}, 1, and 40_{16}, respectively. Note that clearing the memory takes some time. When it is done, s is reset to 0. One could thus wait for this after setting R8E, or just wait long enough, *e.g.*, 100ms.

After the above steps the graphics card is ready to draw text or images. Drawing text (resp. images) requires switching the card to text mode (resp. graphics mode) first. In both cases the *active window*, which restricts where text or graphics can be drawn, must also be configured (it is empty by default). This can be done with the following registers (we show only the bits that we use):

R34$_{16}$	Horizontal End Point of Active Window 0	X_0
R35$_{16}$	Horizontal End Point of Active Window 1	0 0 0 0 0 0 0 X_1
R36$_{16}$	Vertical End Point of Active Window 0	Y_0
R37$_{16}$	Vertical End Point of Active Window 1	0 0 0 0 0 0 0 0 Y_1
R40$_{16}$	Memory Write Control 0	t c b 0 0 0 0 0
R44$_{16}$	Blink Time Control	$blink$

The first 4 registers set the active window's bottom-right corner to $(256.X_1 + X_0, 256.Y_1 + Y_0)$ – its top-left corner is $(0, 0)$ by default (coordinates are measured as shown in Figure 10.1). $t = 1$ enables text mode, while $t = 0$ enables graphics mode. In text mode, $c = 1$ displays a cursor where the next character will be drawn, and $b = 1$ makes it blink every $blink + 1$ frame. In this book we use only text mode in full screen, so we can use $X_0 = 31$, $X_1 = 3$, $Y_0 = 223$, $Y_1 = 1$, $t = c = b = 1$, and, for instance, $blink = 30$. In summary we can set R34, R35, R36, R37, R40, and R44 to $1F_{16}$, 3, DF_{16}, 1, $E0_{16}$ and $1E_{16}$, respectively.

The active window can be cleared by setting $s = 1$ and $a = 1$ in the Memory Clear Control register. Using $s = 1$ and $a = 0$ clears the full screen, whatever the size of the active window.

10.2.2 Text drawing registers

Drawing text on the screen must be done one character at a time, by writing each character's ASCII code (see Appendix B) in the R02 register. Doing this draws the corresponding character at the current cursor position on the screen, with the current background and foreground colors. All characters use 8×16 pixels. The cursor then automatically moves to the right to draw the next character (at the 100^{th} character of a line it goes to the beginning of the next, and at the bottom-right corner it goes back to the top-left corner). Initially the cursor is at the top-left corner, and the foreground and background colors are black. This can be changed with the following registers:

[1] RC7 and R8A control output pins of the RA8875 chip which are connected to the "driver circuit on/off" and "backlight on/off" pins in the 40-pin connector [1].

R2A$_{16}$	Font Write Cursor Horizontal Position 0	x_0
R2B$_{16}$	Font Write Cursor Horizontal Position 1	0 0 0 0 0 0 x_1
R2C$_{16}$	Font Write Cursor Vertical Position 0	y_0
R2D$_{16}$	Font Write Cursor Vertical Position 1	0 0 0 0 0 0 0 y_1
R63$_{16}$	Foreground Color 0	0 0 0 0 0 fc_r
R64$_{16}$	Foreground Color 1	0 0 0 0 0 fc_g
R65$_{16}$	Foreground Color 2	0 0 0 0 0 0 0 fc_b

These registers set the cursor position to $(256.x_1 + x_0, 256.y_1 + y0)$ pixels, and the foreground color's red, green and blue components to fc_r, fc_g, fc_b, respectively. The maximum value of each component corresponds to full intensity. For instance $(fc_r, fc_g, fc_b) = (7, 7, 3)$ corresponds to white.

10.2.3 Communication protocol

In order to read and write values in the above registers, the graphics card accepts 3 types of commands as input: Select Register, Read Data and Write Data. The Select Register command takes as argument a byte containing a register ID (*e.g.*, $2A_{16}$ to select R2A). The Read Data command returns the current value of the last selected register. It does not have any argument. Finally, the Write Data command writes its 8-bit argument in the last selected register. These commands are encoded on 16 bits, with the two most significant containing the command type, and the 8 least significant its argument. The command type is 10_2 for Select Register, 01_2 for Read Data and 00_2 for Write Data. For instance, in order to set R14 to 63_{16}, one must use a Select Register 14_{16} command, followed by a Write Data 63_{16} command (whose encodings are 8014_{16} and 0063_{16}, respectively).

These commands can be sent to the graphics card with the Chip Select (CS), Clock (SCK), and Master Out Slave In (MOSI) pins (see Figure 10.3), as follows (see Figure 10.4):

- set the CS pin to 0 to start a new command (the 3 pins must be 1 by default).
- send the 16 command bits on the MOSI pin, starting with the most significant and ending with the least significant. One bit must be sent each time the SCK pin is rising, *i.e.*, goes from 0 to 1. The SCK frequency must be lower than the graphics card system clock frequency.
- set the CS pin back to 1.

Conversely, the graphics card sends the 8-bit result of Read Data commands on the Master In Slave Out (MISO) pin. It sends these bits during the last 8 clock cycles, from the most to the least significant (see Figure 10.4).

Before sending any command, however, the board must be powered on and reset. It is powered with the VIN and GND pins (see Figure 10.3 – VIN can be connected to 3.3V or 5V). And it can be reset with the Reset (RST) pin. This pin should be 1 by

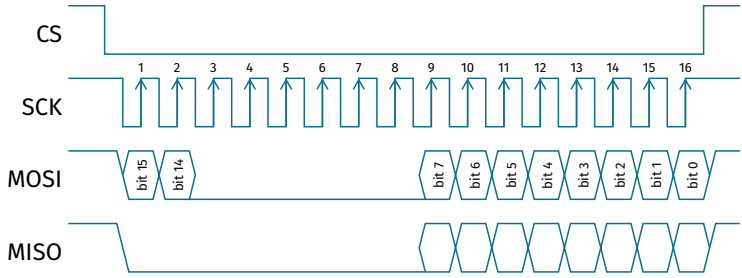


FIGURE 10.4 The signals used between the RA8875 driver board and the Arduino. The Arduino must send 16-bit commands on the MOSI pin while setting the CS pin to 0, one bit at each rising edge of the SCK clock signal. It can optionally receive an 8-bit result on the MISO pin while sending the 8 least significant command bits.

default. Setting it to 0 during at least 1ms starts resetting the graphics card, which takes less than 1ms after RST is set back to 1 (for safety 10ms delays are better).

10.3 Serial Peripheral Interface component

At this stage, "all we have left to do" is to generate appropriate signals on the graphics card pins, in order to set appropriate values in its registers, in turn to configure the card and display text on the screen. In theory, the Arduino could generate these signals with a similar method as our LED blinking programs. Indeed, the signal we generated to blink a LED is in fact a clock signal. With more effort, we could also generate the CS and MOSI signals too. But there is a simpler method. Indeed, the signals shown in Figure 10.4 are a special case of a more general communication protocol called the Serial Peripheral Interface (SPI). And the Arduino microcontroller has a dedicated SPI component to interact with devices using this protocol. This section gives an overview of this component and explains how to use it. A complete description can be found in Chapter 32 of [8].

The SPI component is a hardware circuit which can output CS, SCK, and MOSI signals, and read input MISO signals, using the PA28, 27, 26 and 25 pins, respectively (see Figures 6.2 and 10.5). It does so with 3 main registers (see Table 10.2):

- The Transmit Data Register. Writing a value (up to 16 bits) in this register sends it on the MOSI pin as described above, while setting the CS pin to 0.
- The Receive Data Register. The value received on the MISO pin while sending a value on the MOSI pin is stored in this register.
- The Status Register. This read-only register has the following binary format (we show only the bits that we use):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	t	r

where t is 0 while a value is being sent (and 1 when this is done), and r is 1 iff a new value has been received since the last read of the Receive Data register.

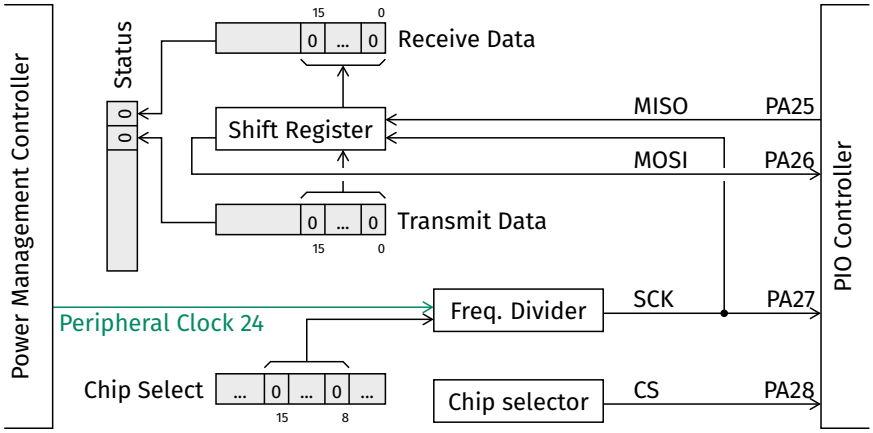


FIGURE 10.5 A simplified representation of the Serial Peripheral Interface (SPI) circuit and registers (in gray). A shift register is used to transmit data in most to least significant bit order on MOSI, while storing bits received in the same order on MISO. It is left shifted after each transmitted / received bit. See Figure 32-5 in [8].

Name	Type	Address
Control Register	Write-Only	40008000_{16}
Mode Register	Read-Write	40008004_{16}
Receive Data Register	Read-Only	40008008_{16}
Transmit Data Register	Write-Only	$4000800C_{16}$
Status Register	Read-Only	40008010_{16}
Chip Select Register	Read-Write	40008030_{16}

TABLE 10.2 The Serial Peripheral Interface controller registers used in this book.

The SPI protocol has many variants. For instance, the number of bits transmitted at a time can vary between 8 and 16. These bits can be transmitted when SCK is rising from 0 to 1, or when it is falling from 1 to 0. The clock frequency can vary, etc. The SPI component supports many such variants in order to support many SPI devices. It can also be used either in master mode, to control a SPI device with the Arduino, or in slave mode, to control the Arduino with such as device. The drawback is that it needs to be configured before being used. This can be done with the following registers (we show only the bits that we use):

- The Mode Register. Its binary format is

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	m

  where $m = 1$ selects the master mode, and $m = 0$ the slave mode.
- The Chip Select Register. Its binary format is

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	*divider*	*bits*	0 0	c	0

where $c = 1$ means that data is sent when SCK is rising, $bits$ means that $bits + 8$ bits are sent at a time, and $divider$ sets the SCK frequency to the Peripheral Clock frequency divided by $divider$.

In our case we need $m = 1$, $c = 1$, and $bits = 8$. The SCK frequency can be set to 4MHz, which is not too large (after reset the graphics card runs at 20MHz). We can thus use $divider = 21 = 15_{16}$ $(84/21 = 4)$. In summary, we can set these two registers to 1 and 1582_{16}, respectively.

Finally, in order to be used, the SPI component must be enabled, which is done by writing the value 1 in the Control Register. Its input clock signal, the Peripheral Clock 24, must also be enabled (see Figure 10.5). This can be done by writing the value 2^{24} in the PMC Peripheral Clock Enable Register (see Section 9.1). Last but not least, the PIO controller must be configured to give control of the PA25, 26, 27, and 28 pins to the SPI component. This can be done by writing the value $2^{25} + 2^{26} + 2^{27} + 2^{28}$ in the PIO A Disable Register (see Section 6.6).

10.4 Graphics card driver

We now have everything we need to write functions to reset and configure the graphics card (or GPU, for Graphics Processing Unit), and to draw text on the screen. We can store them just after the clock driver functions, *i.e.*, at address $C02D8_{16}$.

10.4.1 Low-level functions

Lets start with a `gpu_reset()` function, by using the RST pin. We assume here that RST is connected to Arduino's pin 20, corresponding PB12 (see Figure 6.1). Then, as explained above, the reset function should set PB12 to 0, wait for 10ms, set it to 1, and wait 10ms again. This is very similar to the LED blinking function, with PB27 replaced with PB12 and no loop (see Section 9.7):

```
gpu_reset()
fn      00                  19  C02D8
cst     400E1000            03  +002
cst     00001000            03  +007
store                       14  +00C
cst     400E1010            03  +00D
cst     00001000            03  +012
store                       14  +017
cst     400E1060            03  +018
cst     00001000            03  +01D
store                       14  +022
cst     400E1034            03  +023
cst     00001000            03  +028
store                       14  +02D
cst8    0A                  02  +02E
call    028B delay          1A  +030
cst     400E1030            03  +033
cst     00001000            03  +038
store                       14  +03D
cst8    0A                  02  +03E
call    028B delay          1A  +040
ret                         1D  +043
```

We then need a function to configure and enable the SPI component:

`spi_init()`

Give control of the PA25, 26, 27, and 28 pins to the SPI component with the PIO A Disable Register.

Enable the SPI component clock with the PMC Peripheral Clock Enable Register.

Set the SPI component in master mode with the SPI Mode Register.

Enable the SPI component with the SPI Control Register.

Configure the SPI with a 4MHz clock and 16 bits per transfer, sent when SCK is rising (with the SPI Chip Select Register).

```
fn     00                19 C031C
cst    400E0E04          03  +002
cst    1E000000          03  +007
store                    14  +00C
cst    400E0610          03  +00D
cst    01000000          03  +012
store                    14  +017
cst    40008004          03  +018
cst_1                    01  +01D
store                    14  +01E
cst    40008000          03  +01F
cst_1                    01  +024
store                    14  +025
cst    40008030          03  +026
cst    00001582          03  +02B
store                    14  +030
ret                      1D  +031
```

After that we can provide a `spi_transfer` function to send an arbitrary 16-bit value on MOSI, passed as argument, and returning the received value on MISO:

`spi_transfer`($value$) $\rightarrow$ $response$

Wait until the current transmission, if any, is done, *i.e.*, wait until the t bit of the SPI Status Register value x is 1 ($\Leftrightarrow$ $x \wedge 2 \neq 0$).

Send the data in the function's 0^{th} argument by writing it in the SPI Transmit Data Register.

Wait until the response is received on MISO, *i.e.*, wait until the r bit of the SPI Status Register value x is 1 ($\Leftrightarrow$ $x \wedge 1 \neq 0$).

Read the response in the SPI Receive Data Register and return it.

```
fn     01                19 C034E
cst    40008010          03  +002
load                     13  +007
cst8   02                02  +008
and                      08  +00A
cst_0                    00  +00B
ifeq   0002              0D  +00C
cst    4000800C          03  +00F
get    00    value       16  +014
store                    14  +016
cst    40008010          03  +017
load                     13  +01C
cst_1                    01  +01D
and                      08  +01E
cst_0                    00  +01F
ifeq   0017              0D  +020
cst    40008008          03  +023
load                     13  +028
retv                     1E  +029
```

With this we can provide a function to write a value in a graphics card register.

This function takes as arguments the register ID and the value to write into it:

`gpu_set_register`(id, $value$)

Call the `spi_transfer` function with a Select Register command (bitwise OR of the command type 8000_{16} and the register ID).

Call the `spi_transfer` function with a Write Data command, whose type is 00_{16} (for safety, we keep only the 8 least significant bits of the value), and return.

```
fn      02                  19  C0378
cst     00008000            03  +002
get     00  id              16  +007
or                          09  +009
call    034E spi_transfer   1A  +00A
get     01  value           16  +00D
cst8    FF                  02  +00F
and                         08  +011
call    034E spi_transfer   1A  +012
ret                         1D  +015
```

We can now use this function to configure the graphics card by writing the values described in Section 10.2.1 in its registers, in order (R88 ← 6, R89 ← 1, R04 ← 81_{16}, etc). To reduce code size, we can store these bytes one after the other in memory (`88`, `06`, `89`, `01`, `04`, `81`, etc). A function can then read them 2 by 2 and, for each pair, write the register value with the previous function. Note however that we need to wait at least 0.1ms after setting R88 and R89, and 100ms after writing in the Memory Clear Control register. For this we can use pairs `00`, x to mean "wait x ms" (there is no R00 register). Our list thus becomes `88`, `06`, `00`, `01`, `89`, `01`, `00`, `01`, `04`, `81`, etc. By doing this for all the register values computed in Section 10.2.1 we get the following list, stored just after the previous function, *i.e.*, in [$C0390_{16}$, $C03C0_{16}$[:

```
151B011A DF191917 03160415 63148104 01000189 01000688  C0390
1E44E040 0137DF36 03351F34 408A01C7 80016400 808E151D  C03A8
```

In order to read this list we first need a function to read a single byte in memory:

`load_byte`($address$) → $value$

Load the word x at the given address and return its 8 least significant bits $x \wedge FF_{16}$.

```
fn      01                  19  C03C0
get     00  address         16  +002
load                        13  +004
cst8    FF                  02  +005
and                         08  +007
retv                        1E  +008
```

A function to write a value in a graphics card register, or to wait for some time if the register ID is 0, is also useful:

`gpu_set_register_or_wait`(id, $value$)

If the register ID is equal to 0, jump to the last 3 instructions. Otherwise continue to the next instructions.

The register ID is not 0: write $value$ in register id with `gpu_set_register` and

```
fn      02                  19  C03C9
get     00  id              16  +002
cst_0                       00  +004
ifeq    0010                0D  +005
get     00  id              16  +008
get     01  value           16  +00A
```

return.

The register ID is 0: wait for $value$ ms with `delay` and return.

```
call  0378 …set_register  1A  +00C
ret                       1D  +00F
get   01   value          16  +010
call  028B delay          1A  +012
ret                       1D  +015
```

We can now implement a `gpu_init` function to reset and configure the graphics card, by reading and setting all the register values in the above list. For this we use a pointer p to the next pair to read in the list, stored in the 4^{th} stack frame slot:

`gpu_init()`

Reset the graphic card. Configure the SPI component to communicate with it.

Initialize p to the beginning of the list.

Call `load_byte` to push the byte at address p on the stack (a register ID).

Call it again to push the byte at address $p+1$ on the stack (the value to write in the register).

Call `gpu_set_register_or_wait` with the above values as arguments.

Increment p by 2 to prepare reading the next pair of values.

If p is not the end of the list, go back to offset D_{16}. Otherwise return.

```
fn    00                  19  C03DF
call  02D8 gpu_reset      1A  +002
call  031C spi_init       1A  +005
cst   000C0390 → p        03  +008
get   04   p              16  +00D
call  03C0 load_byte      1A  +00F
get   04   p              16  +012
cst_1                     01  +014
add                       04  +015
call  03C0 load_byte      1A  +016
call  03C9 …or_wait       1A  +019
cst8  02                  02  +01C
add                       04  +01E
get   04   p              16  +01F
cst   000C03C0            03  +021
iflt  000D                0C  +026
ret                       1D  +029
```

10.4.2 Drawing functions

To finish the graphics card driver we provide a function to clear the screen, and 3 functions for drawing text. The `gpu_clear_screen` function sets the s and a bits of the Memory Clear Control register to 1 to start clearing the active window. It then reads s back repeatedly, with Read Data commands (4000_{16}), until it is 0 (which indicates that the operation is done):

`gpu_clear_screen()`

```
fn    00                  19  C0409
cst8  8E                  02  +002
cst8  C0                  02  +004
call  0378 …set_register  1A  +006
cst   00004000            03  +009
call  034E spi_transfer   1A  +00E
cst8  80                  02  +011
and                       08  +013
cst_0                     00  +014
ifne  0009                10  +015
ret                       1D  +018
```

The `gpu_set_cursor` function takes a column c and a row r as arguments, expressed in *number of characters*, and writes $8c$ and $16r$ in the Font Write Cursor registers (see Section 10.2.2 – recall that each character is 8×16 pixels). More precisely, it writes $8c = c \ll 3$ and $16r = r \ll 4$ in R2A and R2B, and $8c \gg 8 = c \gg 5$ and $16r \gg 8 = r \gg 4$ in R2C and R2D (recall that `gpu_set_register` keeps only the 8 least significant bits of its argument):

```
gpu_set_cursor(c, r)
fn    02                      19 C0422
cst8  2A                      02 +002
get   00   c                  16 +004
cst8  03                      02 +006
lsl                           0A +008
call  0378 …set_register      1A +009
cst8  2B                      02 +00C
get   00   c                  16 +00E
cst8  05                      02 +010
lsr                           0B +012
call  0378 …set_register      1A +013
cst8  2C                      02 +016
get   01   r                  16 +018
cst8  04                      02 +01A
lsl                           0A +01C
call  0378 …set_register      1A +01D
cst8  2D                      02 +020
get   01   r                  16 +022
cst8  04                      02 +024
lsr                           0B +026
call  0378 …set_register      1A +027
ret                           1D +02A
```

The `gpu_set_color` function takes 3 arguments r, g and b, and writes them in the respective Foreground Color registers:

```
gpu_set_color(r, g, b)
fn    03                      19 C044D
cst8  63                      02 +002
get   00   r                  16 +004
call  0378 …set_register      1A +006
cst8  64                      02 +009
get   01   g                  16 +00B
call  0378 …set_register      1A +00D
cst8  65                      02 +010
get   02   b                  16 +012
call  0378 …set_register      1A +014
ret                           1D +017
```

Finally, the `gpu_draw_char` function draws the character given as argument by writing it in the R02 register with `gpu_set_register`:

```
gpu_draw_char(c)
fn    01                      19 C0465
cst8  02                      02 +002
get   00   c                  16 +004
call  0378 …set_register      1A +006
ret                           1D +009
```

10.4.3 Summary

In summary, our graphics card driver provides the functions shown in Table 10.3 and its full code is:

```
14000010 00003400E 10100314 00001000 03400E10 00030019  C02D8
0A021400 00100003 400E1034 03140000 10000340 0E106003  C02F0
04030019 1D028B1A 0A021400 00100003 400E1030 03028B1A  C0308
```

Function	Address
gpu_clear_screen()	C0409 ($C0000_{16}$+1033)
gpu_draw_char(c)	C0465 ($C0000_{16}$+1125)
gpu_init()	C03DF ($C0000_{16}$+991)
gpu_set_color(r, g, b)	C044D ($C0000_{16}$+1101)
gpu_set_cursor(c, r)	C0422 ($C0000_{16}$+1058)
gpu_set_register(id, $value$)	C0378 ($C0000_{16}$+888)
load_byte($address$) → $value$	C03C0 ($C0000_{16}$+960)
spi_transfer($value$) → $response$	C034E ($C0000_{16}$+846)

TABLE 10.3 The most important graphics card driver functions.

```
00800403 14010000 00003400E 06100314 1E000000 03400E0E  C0320
01191D14 00001582 03400080 30031401 40008000 03140140  C0338
80100314 00164000 800C0300 020D0008 02021340 00801003  C0350
16000080 00030219 1E134000 80080300 170D0008 01134000  C0368
01000189 01000688 00001D03 4E1A08FF 02011603 4E1A0900  C0380
80016400 808E151D 151B011A DF191917 03160415 63148104  C0398
08FF0213 00160119 1E44E040 0137DF36 03351F34 408A01C7  C03B0
191D028B 1A01161D 03781A01 16001600 100D0000 1602191E  C03C8
03C01A04 01041603 C01A0416 000C0390 03031C1A 02D81A00  C03E0
1AC0028E 0200191D 000D0C00 0C03C003 04160402 0203C91A  C03F8
00162A02 02191D00 09100008 8002034E 1A000040 00030378  C0410
1A0A0402 01162C02 03781A0B 05020016 2B020378 1A0A0302  C0428
64020378 1A001663 0203191D 03781A0B 04020116 2D020378  C0440
  1D0378 1A001602 0201191D 03781A02 16650203 781A0116  C0458
```

Lets store it in flash memory. To avoid you some typing we provide the necessary boot assistant commands in `part2/graphics_card_driver.txt`. Run them with:

```
user@host:~$ python3 flash_helper.py < part2/graphics_card_driver.txt
>Reading page 1026... Done.
Reading page 1027... Done.
Reading page 1028... Done.
Writing page 1026... Done.
Writing page 1027... Done.
Writing page 1028... Done.
>Done.
```

10.5 Experiments

In order to test our driver we can try to display the traditional “Hello, World!" message on the screen. First of all, we need to connect together the Arduino, the graphics card

and the LCD. For this the easiest way is to use a breadboard (see Table A.1). Still, this requires soldering header pins on the Adafruit RA8875 driver board. For this the easiest is to plug the header pins on the breadboard as shown in Figure 10.6, place the board on top of them, and solder the pins in place (see more detailed instructions at https://ebruneton.github.io/toypc/assembly.html). Then connect the LCD 40-pin flat cable to the board: slide out the black "ears" on each side of the board connector, insert the flat cable with the pins oriented as shown in Figure 10.6, and slide the ears back in. Finally, using jumper wires, connect:

- the VIN and GND pins to the Arduino's 3.3V and GND pins,
- the RST pin to the Arduino pin 20 (as assumed in the `gpu_reset` function),
- the MISO, MOSI and SCK pins to the Arduino's PA25, PA26 and PA27 pins, respectively (near the SAM3X8E chip, see Figure 6.1),
- the CS pin to the Arduino's PA28 pin, corresponding to pin 10 (see Figure 6.1).

We can then write a main function, at its expected address C2000_{16} (see Figure 9.2), to display "Hello, World!" on the screen. For this we need the ASCII code of these characters: 48_{16} for "H", 65_{16} for "e", etc (see Appendix B). We can store them in a list, starting a bit after the main function, for instance C2080_{16}:

```
                                    21 646C726F 57202C6F 6C6C6548  C2080
```

We start the main function with calls to `boot_mode_select_rom`, `clock_init`, and `gpu_init` (see Tables 9.3 and 10.3).

```
fn    00                  19 C2000  |  call  0200 clock_init     1A +005
call  02B4 …select_rom    1A +002   |  call  03DF gpu_init       1A +008
```

We then set the foreground color to green and the cursor position to $(20, 3)$:

```
cst_0                     00 +00B   |  cst8  14                  02 +012
cst8  07                  02 +00C   |  cst8  03                  02 +014
cst_0                     00 +00E   |  call  0422 …set_cursor    1A +016
call  044D gpu_set_color  1A +00F   |
```

Finally, we draw the characters by using a loop, with a pointer p to the next character to draw stored in the 4^{th} stack frame slot:

```
Initialize p to point to the 1st character.   cst   000C2080 → p   03 +019
Load the byte at address p and draw it        get   04   p         16 +01E
with gpu_draw_char.                           call  03C0 load_byte 1A +020
                                              call  0465 gpu_draw_char 1A +023
Increment p by 1 to prepare drawing           cst_1                01 +026
the next character.                           add                  04 +027
If p is not the end of the list of char-      get   04   p         16 +028
```

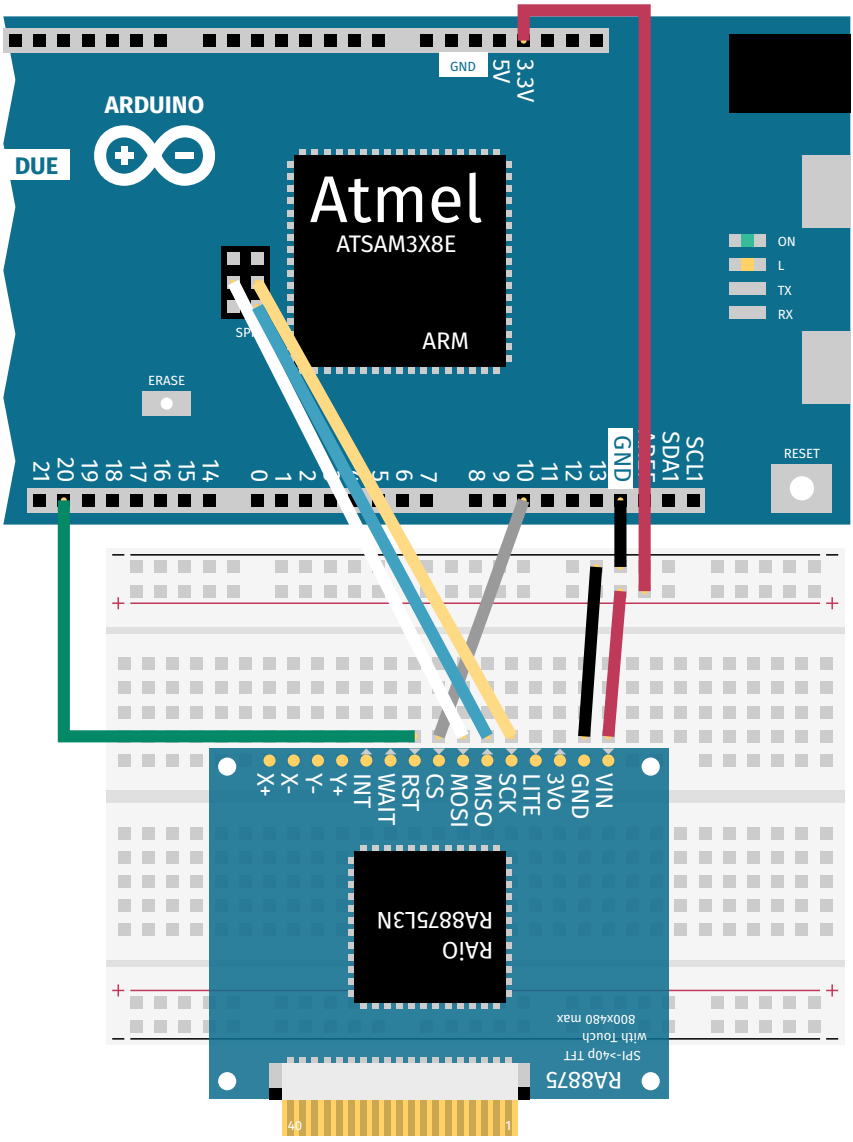


FIGURE 10.6 How to connect the Arduino Due, the Adafruit RA8875 driver board and the LCD with a breadboard.

acters, go back to offset $1E_{16}$. Otherwise loop forever doing nothing.

```
cst   000C208D   03   +02A
iflt  001E       0C   +02F
goto  0032       12   +032
```

To run this main function we need to store it in flash memory and to restart the Arduino with the boot mode selection set to boot from flash. We provide the necessary boot assistant commands in `part2/graphics_card_driver_test.txt`. Run them with:

```
user@host:~$ python3 flash_helper.py < part2/graphics_card_driver_test.txt
>Reading page 1056... Done.
Writing page 1056... Done.
>Done.
```

If all goes well, you should see a green "Hello, World!" message on the screen, followed by a blinking cursor! Note that if you press the RESET button on the Arduino, the message does not disappear, although the Arduino is now running the boot assistant. Indeed, the graphics card is running independently, and is *not* reset when the Arduino is. This is why we reset it explicitly with the `gpu_reset` function. You can now turn off the Arduino, which also turns off the graphics card and the LCD.

CHAPTER

11 Keyboard Driver

In this chapter we continue to assemble our toy computer by connecting a keyboard to the Arduino, and by writing a program to use it. We first present how keyboards work, how they communicate with computers, and how programs can use them. We then write such a program, called the keyboard driver. Finally, we test it with a small application displaying on the screen all the keys typed on the keyboard.

11.1 Keyboard

A computer needs to know which keys of a keyboard are pressed at any given time (several keys can be pressed simultaneously). If the keyboard had only one key, this would be very easy to do. Indeed, one could then use a push button switch connected on one end to VCC and on the other to an input pin of the microprocessor. A program running on the microprocessor could then read the value v of this input pin to know whether the key is pressed ($v = 1$) or not ($v = 0$). However, keyboards usually have about 100 keys. With this simple method, one would need about 100 input pins on the microprocessor to know which keys are pressed or not.

A solution to this problem is to still use one push button switch per key, but to connect them to a grid of wires as shown in Figure 11.1. In this design, each column wire is connected to an output pin, and each row wire is connected to an input pin. To know which keys are pressed, one can set each output pin to 1, *one by one*, and then read the values of the input pins. Indeed, the key at row y and column x is pressed if and only if row y is 1 when column x is 1. With this method, only 20 pins are needed for 100 keys (10 column output pins and 10 row input pins). The drawback is that columns must be *scanned* one by one. This scan could be done by the computer, but this would require a keyboard plug with at least 20 pins. To avoid this, the following method can be used:

- Continuously scan the grid as described above with a small chip *inside* the keyboard. This requires about 20 pins but this is fine, since they are internal.
- When the state of a key has changed since the last scan, send some data to the computer indicating which key is concerned, and whether it became pressed or released. This data can be sent one bit at a time over a *single* pin.

In fact this is what PS/2 (for Personal System/2) keyboards do. Since this method is

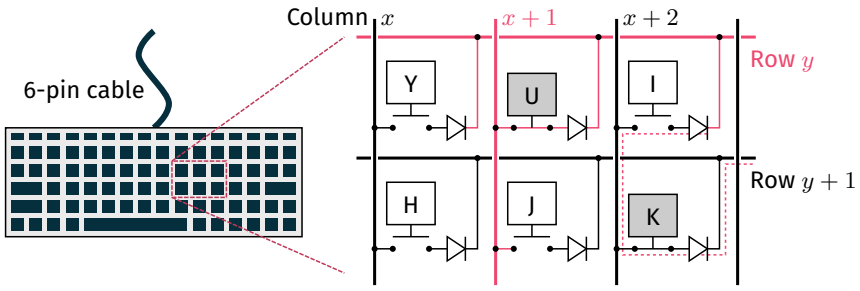


FIGURE 11.1 A schematic view of a possible keyboard circuit. Each key is a push button switch connected to a grid of wires. Setting exactly one column to VCC (in red) sets the rows of the pressed keys (in gray) in this column to VCC too. Diodes prevent *ghosting*: without them, if "I" was pressed too, "J" would incorrectly be considered as pressed because row $y+1$ would be at VCC too (via the dotted path).

much simpler than the one used by USB keyboards, we use such a keyboard for our toy computer, namely a MCSaite model (see Table A.1). *Any other keyboard might not work with our driver*[1].

11.1.1 Scancodes

When a key is pressed or released, a PS/2 keyboard sends to the computer one or more bytes, called *scancodes*. There are several standardized sets of scancodes. The one used by the MCSaite keyboard is IBM PC "set 2" (see Appendix C). With this standard, most keys emit a single scancode when pressed. For instance, pressing the "A" key emits the scancode $1C_{16}$. These keys emit the same scancode when they are released, preceded by $F0_{16}$. For instance, releasing the "A" key emits $F0_{16}$, $1C_{16}$.

A few other keys emit two scancodes when pressed, with the first one always equal to $E0_{16}$. For instance, pressing the "PageUp" key emits $E0_{16}$, $7D_{16}$. When released, these keys emit the same last scancode, preceded by $E0_{16}$, $F0_{16}$. For instance, releasing the "PageUp" key emits $E0_{16}$, $F0_{16}$, $7D_{16}$.

Finally, two keys are exceptions to these rules. "PrintScreen" emits 4 scancodes when pressed ($E0_{16}$, 12_{16}, $E0_{16}$, $7C_{16}$), and 6 when released ($E0_{16}$, $F0_{16}$, $7C_{16}$, $E0_{16}$, $F0_{16}$, 12_{16}). "Pause" emits 8 scancodes when pressed ($E1_{16}$, 14_{16}, 77_{16}, $E1_{16}$, $F0_{16}$, 14_{16}, $F0_{16}$, 77_{16}), and no scancodes when released.

Note that scancodes are different from ASCII codes. One reason is that some keys do not have any equivalent ASCII code, such as the "PageUp" key. Another reason is that some keys have two corresponding ASCII codes, such as the "A" key (one for "a", and one for "A"). However, most programs need ASCII codes instead of scancodes. For instance, an ASCII code is needed to draw a character on the screen. A small program is thus needed to convert scancodes to ASCII codes.

[1] Especially USB-only keyboards, even with the USB to PS/2 plug adapter provided with the MCSaite. This adapter works with the MCSaite because this keyboard supports both protocols.

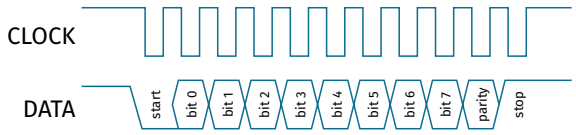


FIGURE 11.2 The signals output by a PS/2 keyboard. Each scancode is sent on the DATA pin, one bit per CLOCK cycle, starting with the least significant. It is preceded by a "start" bit (always 0) and followed by a "parity" bit and a "stop" bit (always 1).

11.1.2 Communication protocol

A PS/2 keyboard uses a 6-pin mini-DIN connector to send the above scancodes to the computer. Two pins are unused. Two other pins, GND and VCC, are inputs provided by the computer to power the keyboard. One pin, named CLOCK, is a clock signal generated by the keyboard. Finally, the last pin, named DATA, is used to actually send data. Each scancode is sent separately, one bit b_i at each clock cycle, as shown in Figure 11.2. These 8 bits are preceded by a "start" bit and followed by a "parity" bit and a "stop" bit. The parity bit p is computed so that $b_0 + b_1 + \ldots + b_7 + p$ is odd (this is called *odd parity*). It is used to detect transmission errors (if the previous sum, computed by the receiver, is even, then at least one received bit is incorrect).

This communication protocol is qualified as *serial* and *synchronous*. Serial means that one bit is transmitted at a time. The opposite, *i.e.*, transmitting several bits at a time, like a 40-pin LCD connector does, is called *parallel* transmission. Synchronous means that a separate clock signal is used, indicating when the data signal can be read (depending on the protocol, this can be when the clock signal is 0, when it is rising, when it is falling, etc). The opposite, *i.e.*, not using any clock signal, is called *asynchronous* transmission. In this case the receiver and the transmitter must agree on a bit rate beforehand.

11.2 Universal Synchronous Asynchronous Receiver Transmitter

At this stage, "all we have left to do" is to write a program to read the above signals, recover the corresponding scancodes, and then the corresponding ASCII codes. This could be done with the PIO controller, but there is a much simpler method. Indeed, the Arduino microcontroller has a dedicated component to interact with almost any device using a serial communication method, be it synchronous or asynchronous. It is called the Universal Synchronous Asynchronous Receiver Transmitter (USART) component. This section gives an overview of this component and explains how to use it. A complete description can be found in Chapter 35 of [8].

The core part of the USART component is similar to the SPI component (see Section 10.3). Indeed, this component is a hardware circuit which can receive serial data on an input pin called RXD, and transmit data on an output pin called TXD,

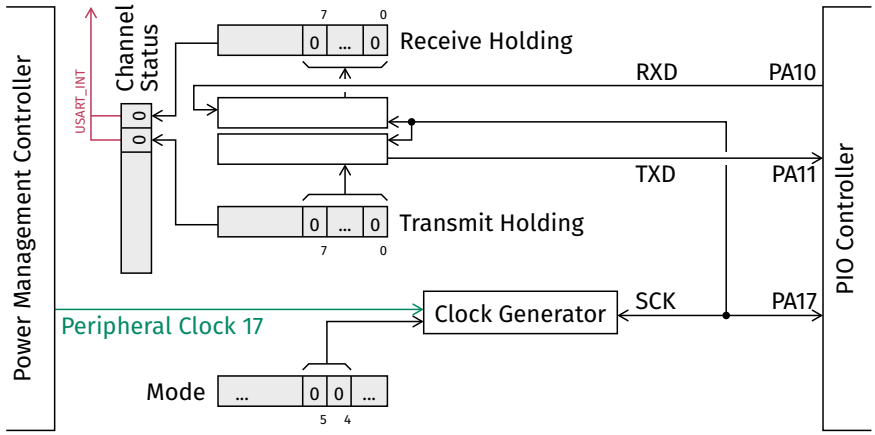


FIGURE 11.3 A simplified representation of the Universal Synchronous Asynchronous Receiver Transmitter (USART) circuit and registers (in gray).

Name	Type	Address
Control Register	Write-Only	40098000_{16}
Mode Register	Read-Write	40098004_{16}
Interrupt Enable Register	Write-Only	40098008_{16}
Channel Status Register	Read-Only	40098014_{16}
Receiver Holding Register	Read-Only	40098018_{16}

TABLE 11.1 The Universal Synchronous Asynchronous Receiver Transmitter registers used in this book.

synchronously or not with a clock signal on a SCK pin (see Figure 11.3). This is similar to the MISO, MOSI, and SCK pins of the SPI component, respectively. This component is also based on 3 main registers, similar to the SPI Transmit Data, SPI Receive Data, and SPI Channel Registers (see Table 11.1):

- The Transmit Holding Register. Writing a value in this register sends it on the TXD pin.
- The Receive Holding Register. Values received on the RXD pin are stored in this register.
- The Channel Status Register. This read-only register has the following binary format (we show only the bits that we use):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	s_t	s_r

where s_t is 0 while a value is being sent (and 1 when this is done), and s_r is 1 iff a new value has been received since the last read of the Receive Holding register.

The USART component must also be configured before being used, as the SPI

component. The main register which can be used to do this is the Mode Register, similar to the SPI Channel Select Register. Indeed this register has the following binary format (we show only the bits that we use):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	o	0	0	$stop$	$parity$	s	$bits$	clk	0	0	0	0

- The clk field indicates which clock signal to use for SCK. One option is to use the Peripheral Clock provided by the PMC. Another option, corresponding to $clk = 3$, is to use the signal provided by the external device (in our case the beyboard).
- The $bits$ field indicates that $bits + 5$ bits are sent or received at a time.
- The s field indicates if the component should operate in synchronous mode ($s = 1$) or in asynchronous mode ($s = 0$).
- The $parity$ field indicates how the parity bit p is computed (see Section 11.1.2). Odd parity corresponds to $parity = 1$.
- The $stop$ field indicates how many stop bits are used. $stop = 0$ corresponds to one stop bit, while $stop = 2$ corresponds to two stop bits.
- The o field indicates if least significant bit is sent first ($o = 0$), or if the most significant bit is sent first ($o = 1$).

In our case we need $clk = 3$, $bits = 3$, $s = 1$, $parity = 1$, $stop = 0$ and $o = 0$ (see Figure 11.3). We thus need to set the Mode Register to $3F0_{16}$. Another configuration register is the Interrupt Enable Register. This register has the following binary format (we show only the bits that we use):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	i_t	i_r

Setting $i_r = 1$ (resp. $i_t = 1$) means that an interrupt (see Sections 7.1 and 7.4) should be triggered when the Channel Status s_r bit (resp. s_t bit) is 1 (setting i_r or i_t to 0 has no effect). This process is explained in more details in the next section.

Finally, in order to be used, the USART component must be enabled. This can be done with the Control Register, which has the following binary format (we show only the bits that we use):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	e_t	0	e_r	0	0	0	0

Setting $e_r = 1$ enables the receiver part, while setting $e_t = 1$ enables the transmitter part (in our case we just need the receiver part). The USART internal clock, the Peripheral Clock 17 (see Figure 11.3), must also be enabled (even if an external SCK is used). This can be done by writing the value 2^{17} in the PMC Peripheral Clock Enable Register (see Section Section 9.1). Last but not least, the RXD and SCK pins must be configured as input pins (we don't need to configure the TXD pin since we don't use it). This can be done by setting the corresponding bits to 1 in the PIO Status Register and to 0 in the PIO Output Status Register (see Section 6.6). In fact the default value of the PIO Output Status Register is 0, so we just need to configure the PIO Status Register. Since the RXD and SCK pins correspond to PA10 and PA17 (see Figure 11.3), this can be done by writing the value $2^{10} + 2^{17}$ in the PIO A Enable Register.

11.3 Nested Vector Interrupt Controller

Thanks to the USART component, a program could periodically read the Channel Status register to check if a new scancode has been received from the keyboard. If so, it could then read it in the Receive Holding register. However, this method has an important drawback: if two or more scancodes are received since the last check, only the last one is available in the Receive Holding register (new values override previous ones). Some keys typed during this time could thus be missed. Even worse, some keys could be interpreted incorrectly. For instance, if the "PageUp" key is pressed, which emits $E0_{16}$, $7D_{16}$, and if $E0_{16}$ is missed, the above program would just get $7D_{16}$, which is the scancode emitted by the "9" key on the numeric pad. It would thus consider that this key was pressed, which is wrong.

In order to solve this problem we must make sure to not miss any scancode. This can be done with an interrupt, triggered each time a new value is received by the USART component. Recall that when an interrupt is triggered, execution jumps to an *interrupt handler*, at an address given by the Vector Table (see Sections 7.1 and 7.4). In our case, this handler could read the scancode, store it in an unused memory location, and return to the main program. This takes much less time than the reception of a scancode (the CLOCK frequency is at most 16.7kHz). Thus, provided there is enough unused memory to store received scancodes that have not yet been read by the main program, no scancode would be missed.

11.3.1 Interrupt status

In order to use interrupts, one must enable them first. The USART component can trigger an interrupt when a new value is received, when a value has been transmitted, etc. The first step is to choose which event(s) should trigger an interrupt. This is done with the USART Interrupt Enable Register presented in the previous section. The second step is to enable these interrupts in the Nested Vector Interrupt Controller (NVIC). This component provides two main registers to enable or disable the interrupts triggered by each peripheral (see Table 11.2):

- The Interrupt Set-Enable Register. Setting the bit i of this register to 1 enables the interrupts from peripheral number i (recall that peripherals are numbered, *e.g.*, 17 for USART, 24 for SPI, etc – see Figure 6.2). Setting a bit to 0 has no effect.
- The Interrupt Clear-Enable Register. Setting the bit i of this register to 1 disables the interrupts from peripheral number i. Setting a bit to 0 has no effect.

Note that if a scancode is received while the USART interrupt is disabled in the NVIC (but not at the USART level), its handler is not executed, but the interrupt becomes *pending*. When a pending interrupt is enabled, it becomes *active*, *i.e.*, its handler is executed. It is thus possible to temporarily disable the USART interrupt, without loosing any scancode, provided that the interrupt is not disabled for too long (*i.e.*, such that at most one scancode can be received during this time).

Name	Type	Address
Interrupt Set-Enable Register	Read-Write	$E000E100_{16}$
Interrupt Clear-Enable Register	Read-Write	$E000E180_{16}$

TABLE 11.2 The Nested Vector Interrupt Controller registers used in this chapter.

11.3.2 Interrupt handler entry and return

When an interrupt from peripheral number i becomes active, execution jumps to the interworking address stored in the $(16+i)^{th}$ entry of the Vector Table, *i.e.*, at offset $4(16+i)$ from the beginning of this table. Thus, for instance, when the USART interrupt becomes active, execution jumps to the interworking address at offset 84_{16}.

Before this, however, the microprocessor pushes on the stack the address of the instruction to return to when the handler terminates. It also pushes on the stack the current value of some registers, including R0, R1, R2, R3, and the Link Register (LR). Finally, the microprocessor sets the LR to a special value, called EXC_RETURN, and starts executing the interrupt handler by moving its interworking address into the Program Counter (Chapter 23 gives more details about this).

The interrupt handler can return and resume execution in the interrupted program by copying this EXC_RETURN value into the PC. When this happens, the microprocessor pops the values pushed above to restore the register values as they were before the interrupt. An interrupt handler can thus simply push the LR on the stack when it starts, and finally pop it into the PC to resume execution of the interrupted program.

11.4 Keyboard driver

We now have everything we need to write our keyboard driver. We start with a presentation of its goal and of the method used to achieve it. We then provide the corresponding implementation.

11.4.1 Design

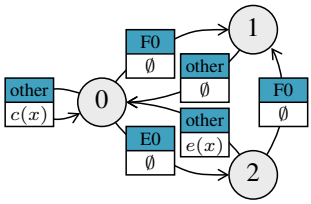
The main goal of the keyboard, for our basic input output system, is to input data without needing an external computer. Any data can be represented with 0s and 1s and thus, in theory, we just need two keys, or even just one. But this would not be very practical. On the other hand, we don't need to handle all keys, nor all combinations of keys (such as Ctrl+C, Ctrl+V, etc). In order to get a practical but simple driver we thus make the following choices. First, the driver should not output anything when a key is released. Second, when a key is pressed, the driver should output the corresponding ASCII code if there is one. If there are two, it should output one or the other, depending on whether the Shift key is pressed or not. If there are none, it should output *some* code between 128 and 255 (ASCII codes are between 0 and 127 included).

Finite State Machine

In the following we call *character* a byte value which is either an ASCII code or, if larger than 128, some value representing a key without ASCII code. The main goal of our driver is thus to transform the sequence of scancodes into a sequence of characters. As explained in Section 11.1.1, besides two exceptions, this sequence is made of subsequences of the form (x), $(\text{F0}_{16}, x)$, $(\text{E0}_{16}, x)$, or $(\text{E0}_{16}, \text{F0}_{16}, x)$. The driver should thus react differently when a scancode x_t is received, depending on the previous scancode x_{t-1} (we ignore the exceptions and the Shift key for now):

- if $x_{t-1} = F0_{16}$, do nothing since $(\ldots, \text{F0}_{16}, x_t)$ corresponds to a key release,
- if $x_{t-1} = E0_{16}$ and $x_t \neq F0_{16}$, output the character corresponding to $(\text{E0}_{16}, x_t)$,
- if $x_{t-1} = E0_{16}$ and $x_t = F0_{16}$, do nothing,
- otherwise, if $x_t \notin \{F0_{16}, E0_{16}\}$, output the character corresponding to (x_t).

Said otherwise, at any time t, the driver can be in 3 different *states* S_t, which can be noted 1 for $x_{t-1} = F0_{16}$, 2 for $x_{t-1} = E0_{16}$, and 0 for all other cases. Each input x_t triggers an *action* A (output a character or do nothing) which depends on S_t and x_t. It also makes the driver *transition* to a new state S_{t+1}, depending on x_t. This can be summarized in a *transition table*, or represented in a diagram, as follows:



$x_t = F0_{16}$		$x_t = E0_{16}$		otherwise		
S_{t+1}	A	S_{t+1}	A	S_{t+1}	A	S_t
1	$\emptyset$	2	$\emptyset$	0	$c(x_t)$	0
–	–	–	–	0	$\emptyset$	1
1	$\emptyset$	–	–	0	$e(x_t)$	2

where $\emptyset$ means "do nothing", $c(x)$ and $e(x)$ mean "output the character corresponding to (x) and $(\text{E0}_{16}, x)$", respectively, and "–" are cases which cannot happen. Such a system is called a *finite state automaton* or *finite state machine*.

To handle the Shift key[2] we can use a bit s set to 1 when this key is pressed (which emits 12_{16}) and reset to 0 when it is released (which emits F0_{16}, 12_{16}). This adds a new column and two actions in the transition table (an alternative is to duplicate each state, but this gives a larger transition table):

$x_t = 12_{16}$		$x_t = F0_{16}$		$x_t = E0_{16}$		otherwise		
0	$s \leftarrow 1$	1	$\emptyset$	2	$\emptyset$	0	$c(x_t, s)$	$S_t = 0$
0	$s \leftarrow 0$	–	–	–	–	0	$\emptyset$	$S_t = 1$
–	–	1	$\emptyset$	–	–	0	$e(x_t, s)$	$S_t = 2$

where $c(x_t, s)$ and $e(x_t, s)$ are now extended to output one character or another, depending on the Shift key state s.

[2]The MCSaite keyboard has only a left Shift key. We thus ignore the right Shift key (which emits 59_{16}).

The exception sequence ($E1_{16}$, 14_{16}, 77_{16}, $E1_{16}$, $F0_{16}$, 14_{16}, $F0_{16}$, 77_{16}) for the "Pause" key can be seen as 3 sequences ($E1_{16}$, 14_{16}, 77_{16}), ($E1_{16}$, $F0_{16}$, 14_{16}), and ($F0_{16}$, 77_{16}). The last one is already handled. The first two can be managed with a new state $S = 3$, a transition from $S = 0$ to $S = 3$ when $x_t = E1_{16}$, and from $S = 3$ to $S = 1$ for any scancode. Indeed, with the next transition from $S = 1$ to $S = 0$, this reads 3 scancodes. To simplify we use empty actions for these transitions, *i.e.*, we do nothing when "Pause" is pressed.

Finally, the exception sequence ($E0_{16}$, 12_{16}, $E0_{16}$, $7C_{16}$) emitted by "PrintScreen" can be seen as two normal sequences. We can then do nothing for the first, and associate the second with "PrintScreen". One issue is that releasing this key emits ($\ldots$, $E0_{16}$, $F0_{16}$, 12_{16}). With the above transition table, this is interpreted as releasing the Shift key, which is wrong. To detect this case we introduce a fifth state, $S = 4$. This leads to our final transition table:

$x_t = 12_{16}$		$x_t = F0_{16}$		$x_t = E0_{16}$		$x_t = E1_{16}$		otherwise		
0	$s \leftarrow 1$	1	$\emptyset$	2	$\emptyset$	3	$\emptyset$	0	$c(x_t, s)$	$S_t = 0$
0	$s \leftarrow 0$	–	–	–	–	–	–	0	$\emptyset$	$S_t = 1$
0	$\emptyset$	4	$\emptyset$	–	–	–	–	0	$e(x_t, s)$	$S_t = 2$
–	–	1	$\emptyset$	–	–	–	–	1	$\emptyset$	$S_t = 3$
0	$\emptyset$	–	–	–	–	–	–	0	$\emptyset$	$S_t = 4$

Character queue

Some actions of the above finite state machine "output a character", but what does this mean exactly? For the reasons explained in Section 11.3, we want to implement this machine in an interrupt handler. The output characters should thus be stored somewhere in memory, where the main program can later read them. To simplify, we use a single byte storage b, at a fixed memory address, and use $b = 0$ to represent an empty storage (this value cannot be output by any key). The action "output a character c" can then be specified as follows:

put_char(c): if $b = 0$ store c in b, otherwise do nothing (*i.e.*, drop c).

The main program could then read a character with the following function, which returns a character c if there is one in storage, or 0 otherwise:

get_char(): copy b in c, then store 0 in b, then return c.

In fact this function has a bug. Indeed, it can be interrupted at any time. The put_char function might therefore be called at any time, for instance between the first and second steps of get_char. The following scenario could then happen: b is currently 0, the first step stores 0 into c, put_char(d) runs and stores d in b, get_char resumes, stores 0 in b and returns 0. The end result is that d is lost! To avoid this, a solution is to disable the USART interrupt at the beginning of get_char, and to re-enable it at the

end (before the return). This ensures a *mutual exclusion* of the instructions using the shared storage b, via their *atomicity* (*i.e.*, the fact that they can't be interrupted).

Note that characters could still be dropped, in put_char, if they are typed faster than they are read. To avoid this more memory could be used to store them in a *queue*, until they are read. A queue is a "First In, First Out" (FIFO) data structure. It is a bit more complex to implement than a stack – a "Last In, First Out" (LIFO) data structure, which is why we use a single byte storage instead.

Character tables

To complete our design we need to specify the above $c(x, s)$ and $e(x, s)$ functions. The latter corresponds to keys which don't have ASCII codes (except the numeric pad "/" and "Enter" keys). We can thus choose $e(x, s)$ as we want in $[128, 255]$. Note also that $x < 128$ in ($E0_{16}$,x) sequences (see Appendix C). Hence, and since Shift has no effect on these keys, we can simply use $e(x, s) = x + 128$.

The $c(x, s)$ function can be obtained by combining the tables in Appendices B and C. For instance, $c(1C_{16}, s)$ is obtained by looking up the key corresponding to ($1C_{16}$), namely "a / A", and then by looking up the ASCII codes for "a" and "A", namely 61_{16} and 41_{16}. This gives $c(1C_{16}, 0) = 61_{16}$ and $c(1C_{16}, 1) = 41_{16}$. Some (x) sequences do not have any associated ASCII code, such as 01_{16}. In such cases we can set $c(x, s)$ to any value in $[128, 255]$ not already used. Finally, we can use $c(x, s) = 0$ for cases which cannot happen, such as $x = 00_{16}$. By doing this for all scancodes, and with the choices listed in Table 11.3, we get the following character tables for $s = 0$ and $s = 1$, each with 132 elements (listed from right to left):

```
00 31 71 8C 00 00 8D 00 00 60 09 83 85 87 89 00 8B 81 80 82 84 00 88 00
00 35 72 74 66 76 20 00 00 33 34 65 64 78 63 00 00 32 77 61 73 7A 00 00
00 39 30 6F 69 6B 2C 00 00 38 37 75 6A 6D 00 00 00 36 79 67 68 62 6E 00
00 00 5C 00 5D 0A 00 8F 00 00 3D 5B 00 27 00 00 00 2D 70 3B 6C 2F 2E 00
8E 1B 38 36 35 32 2E 30 00 00 00 37 34 00 31 00 00 08 00 00 00 00 00 00
                                    86 00 00 00 00 90 39 2A 2D 33 2B 8A
00 21 51 8C 00 00 8D 00 00 7E 09 83 85 87 89 00 8B 81 80 82 84 00 88 00
00 25 52 54 46 56 20 00 00 23 24 45 44 58 43 00 00 40 57 41 53 5A 00 00
00 28 29 4F 49 4B 3C 00 00 2A 26 55 4A 4D 00 00 00 5E 59 47 48 42 4E 00
00 00 7C 00 7D 0A 00 8F 00 00 2B 7B 00 22 00 00 00 5F 50 3A 4C 3F 3E 00
8E 1B 38 36 35 32 2E 30 00 00 00 37 34 00 31 00 00 08 00 00 00 00 00 00
                                    86 00 00 00 00 90 39 2A 2D 33 2B 8A
```

11.4.2 Implementation

We can finally implement our keyboard driver. We start by storing the above tables just after the graphics card driver, *i.e.*, at address $C0470_{16}$. The value $c(x, s)$ is then given by the byte at offset $x + 132s$ from this address. We continue with a function to initialize the driver state (S, s, and b) and the USART component:

Key	Char	Key	Char	Key	Char	Key	Char
F1	80	F6	85	F11	8A	CapsLock	8F
F2	81	F7	86	F12	8B	ScrollLock	90
F3	82	F8	87	Ctrl (left)	8C		
F4	83	F9	88	Alt (left)	8D		
F5	84	F10	89	NumLock	8E		

TABLE 11.3 The characters chosen for keys without ASCII code.

`keyboard_init()`

Initialize S, s, and b to 0. Each could be stored in one byte but, in order to simplify the code, we use one word for each, at addresses $400E1A90_{16}$, $400E1A94_{16}$, and $400E1A98_{16}$, respectively. These are some "General Purpose Backup Registers" in the "Controllers" memory region (see Figure 6.3 and Chapter 17 in [8]).

Configure the PA10 and PA17 pins as inputs with the PIO A Enable Register.

Enable the USART component clock (Peripheral clock 17) with the Peripheral Clock Enable Register.

Configure the USART component for the PS/2 signals with the USART Mode Register (see Section 11.2).

Configure the USART component to trigger an interrupt when a scancode is received, with the USART Interrupt Enable Register.

Enable the USART interrupt with the NVIC Interrupt Set-Enable Register.

Enable the USART receiver with the USART Control Register and return.

```
fn      00          19  C0578
cst     400E1A90    03  +002
cst_0               00  +007
store               14  +008
cst     400E1A94    03  +009
cst_0               00  +00E
store               14  +00F
cst     400E1A98    03  +010
cst_0               00  +015
store               14  +016
cst     400E0E00    03  +017
cst     00020400    03  +01C
store               14  +021
cst     400E0610    03  +022
cst     00020000    03  +027
store               14  +02C
cst     40098004    03  +02D
cst     000003F0    03  +032
store               14  +037
cst     40098008    03  +038
cst_1               01  +03D
store               14  +03E
cst     E000E100    03  +03F
cst     00020000    03  +044
store               14  +049
cst     40098000    03  +04A
cst8    10          02  +04F
store               14  +051
ret                 1D  +052
```

We then implement the put_char and get_char functions, as well as a wait_char function which repeatedly calls get_char until it returns a non-zero value:

`keyboard_put_char(`c`)`	**fn**	`01`	**19**	`C05CB`
If $b \neq 0$ go to the end of the function (the character queue is full, drop c).	**cst**	`400E1A98`	**03**	`+002`
	load		**13**	`+007`
	cst_0		**00**	`+008`
	ifne	`0014`	**10**	`+009`
Store c (the function's 0^{th} argument) in b.	**cst**	`400E1A98`	**03**	`+00C`
	get	`00` c	**16**	`+011`
	store		**14**	`+013`
Return.	**ret**		**1D**	`+014`
`keyboard_get_char()` $\rightarrow c$	**fn**	`00`	**19**	`C05E0`
Disable the USART interrupt with the NVIC Interrupt Clear-Enable Register.	**cst**	`E000E180`	**03**	`+002`
	cst	`00020000`	**03**	`+007`
	store		**14**	`+00C`
Push b on the stack. This is the value returned by the `retv` instruction below.	**cst**	`400E1A98`	**03**	`+00D`
	load		**13**	`+012`
Store 0 in b.	**cst**	`400E1A98`	**03**	`+013`
	cst_0		**00**	`+018`
	store		**14**	`+019`
Re-enable the USART interrupt with the NVIC Interrupt Set-Enable Register.	**cst**	`E000E100`	**03**	`+01A`
	cst	`00020000`	**03**	`+01F`
	store		**14**	`+024`
Return the top stack value.	**retv**		**1E**	`+025`
`keyboard_wait_char()` $\rightarrow c$	**fn**	`00`	**19**	`C0606`
Initialize c to 0.	**cst_0**	$\rightarrow c$	**00**	`+002`
Call `keyboard_get_char` and store the result in c.	**call**	`05E0` `…get_char`	**1A**	`+003`
	set	`04` c	**17**	`+006`
If c is 0 go back above to try again.	**get**	`04` c	**16**	`+008`
	cst_0		**00**	`+00A`
	ifeq	`0003`	**0D**	`+00B`
Otherwise return c, the top stack value.	**retv**		**1E**	`+00E`

The put_char function allows us to implement the actions of our Finite State Machine. We use one function per action, for the "do nothing", "s $\leftarrow$ 0", "s $\leftarrow$ 1", "output $e(x, s)$" and "output $c(x, s)$" actions, respectively:

`keyboard_skip_code(`$scancode$`)`	**fn**	`01`	**19**	`C0615`
Do nothing.	**ret**		**1D**	`+002`
`keyboard_release_shift(`$scancode$`)`	**fn**	`01`	**19**	`C0618`
Set s to 0.	**cst**	`400E1A94`	**03**	`+002`
	cst_0		**00**	`+007`
	store		**14**	`+008`
	ret		**1D**	`+009`

`keyboard_press_shift(`*scancode*`)`
Set s to 132. We premultiply s by 132 to simplify the `keyboard_put_code` function below.

```
fn    01                  19 C0622
cst   400E1A94            03  +002
cst8  84                  02  +007
store                     14  +009
ret                       1D  +00A
```

`keyboard_put_extended_code(`*scancode*`)`
Output $e(scancode, s) = scancode + 128$ with the `keyboard_put_char` function.

```
fn    01                  19 C062D
get   00   scancode       16  +002
cst8  80                  02  +004
add                       04  +006
call  05CB …put_char      1A  +007
ret                       1D  +00A
```

`keyboard_put_code(`*scancode*`)`
Compute the address of the byte containing $c(scancode, s)$, namely $C0470_{16}$ + *scancode* + s (s is premultiplied by 132, see above).

```
fn    01                  19 C0638
cst   000C0470            03  +002
get   00   scancode       16  +007
add                       04  +009
cst   400E1A94            03  +00A
load                      13  +00F
add                       04  +010
```

Call `load_byte` to load the byte at this address, output it with `keyboard_put_char`.

```
call  03C0 load_byte      1A  +011
call  05CB …put_char      1A  +014
ret                       1D  +017
```

In turn, these functions allow us to implement the transition table of the Finite State Machine. We represent it with one byte per cell, from right to left and top to bottom. We store in each "action" cell the address of the corresponding function (minus the address of `keyboard_skip_code` so that the result fits in one byte). We also premultiply the "next state" cell values by 10, the number of columns, so that a state value directly gives the offset of the beginning of its row in the transition table. The end result, with 0s for the "cannot happen" cases, is the following data, stored just after the above functions:

```
00000018 00030000 00000000 0000000D 0A001400 1E000023  C0650
00000000 00000000 00000A00 00000000 0A000000 28000000  C0668
                                               0000    C0680
```

In order to read the action and next state corresponding to a scancode in this table, it is useful to have a function returning its "action column" index (numbered from right to left):

`keyboard_action_column(`*scancode*`)` $\rightarrow c$
If $scancode \neq 12_{16}$, skip the next two instructions.
Otherwise ($scancode = 12_{16}$) return 8.

```
fn    01                  19 C0682
get   00   scancode       16  +002
cst8  12                  02  +004
ifne  000C                10  +006
cst8  08                  02  +009
retv                      1E  +00B
```

If $scancode \geq 132$, skip the next two instructions.

Otherwise ($scancode < 132$) return 0.

If $scancode \neq E0_{16}$, skip the next two instructions.

Otherwise ($scancode = E0_{16}$) return 4.

If $scancode \neq F0_{16}$, skip the next two instructions.

Otherwise ($scancode = F0_{16}$) return 6.

In any other case return 2 ($scancode$ is necessarily equal to $E1_{16}$).

```
get    00   scancode      16   +00C
cst8   84                 02   +00E
ifge   0015               11   +010
cst_0                     00   +013
retv                      1E   +014
get    00   scancode      16   +015
cst8   E0                 02   +017
ifne   001F               10   +019
cst8   04                 02   +01C
retv                      1E   +01E
get    00   scancode      16   +01F
cst8   F0                 02   +021
ifne   0029               10   +023
cst8   06                 02   +026
retv                      1E   +028
cst8   02                 02   +029
retv                      1E   +02B
```

With this we can finally implement the interrupt handler. This function reads a scancode and executes the corresponding Finite State Machine transition:

`keyboard_handler()`

Read the newly received scancode x in the USART Receive Holding Register.

Compute the address a of the transition table row corresponding to the current state ($C0650_{16}$ + S, since state values are premultiplied by 10).

Add the action column index of x (available in the 4^{th} stack frame slot) to a, with the help of `keyboard_action_column`.

Push x on the stack.

Compute the address of the action function $C0615_{16}$ + `load_byte`(a) (a is in the 5^{th} stack frame slot).

Call this function on x (pushed above).

Update S to the next state value, given by `load_byte`($a + 1$), and return.

```
fn     00                 19   C06AE
cst    40098018           03   +002
load          → x         13   +007
cst    000C0650           03   +008
cst    400E1A90           03   +00D
load                      13   +012
add           → a         04   +013
get    04     x           16   +014
call   0682   …action_column 1A  +016
add                       04   +019
get    04     x           16   +01A
cst    000C0615           03   +01C
get    05     a           16   +021
call   03C0   load_byte   1A   +023
add                       04   +026
calld                     1C   +027
cst    400E1A90           03   +028
get    05     a           16   +02D
cst_1                     01   +02F
add                       04   +030
call   03C0   load_byte   1A   +031
store                     14   +034
ret                       1D   +035
```

Function	Address
keyboard_get_char() $\rightarrow c$	C05E0 (C0000$_{16}$+1504)
keyboard_handler()	C06AE (C0000$_{16}$+1710)
keyboard_init()	C0578 (C0000$_{16}$+1400)
keyboard_wait_char() $\rightarrow c$	C0606 (C0000$_{16}$+1542)

TABLE 11.4 The most important keyboard driver functions.

In summary, the most important functions provided by our keyboard driver are those in Table 11.4 and its full code and data (besides the two 132 characters tables) is:

```
03140040 0E1A9803 1400400E 1A940314 00400E1A 90030019  C0578
80040314 00020000 03400E06 10031400 02040003 400E0E00  C0590
02000003 E000E100 03140140 09800803 14000003 F0034009  C05A8
03001410 0013400E 1A980301 191D1410 02400980 00031400  C05C0
1A980314 00020000 03E000E1 80030019 1D140016 400E1A98  C05D8
00191E14 00020000 03E000E1 00031400 400E1A98 0313400E  C05F0
00400E1A 94030119 1D01191E 0003D00 04160417 05E01A00  C0608
1D05CB1A 04800200 1601191D 14840240 0E1A9403 01191D14  C0620
1D05CB1A 03C01A04 13400E1A 94030400 16000C04 70030119  C0638
00000018 00030000 00000000 0000000D 0A001400 1E000023  C0650
00000000 00000000 00000A00 00000000 0A000000 28000000  C0668
161E0000 15118402 00161E08 02000C10 12020016 01190000  C0680
00191E02 021E0602 002910F0 0200161E 04020001F 10E00200  C0698
04006821A 04160413 400E1A90 03000C06 50031340 09801803  C06B0
1A040105 16400E1A 90031C04 03C01A05 16000C06 15030416  C06C8
                                                1D1403C0  C06E0
```

The last piece that we need is to configure the Vector Table entry for the USART interrupt to call our `keyboard_handler`. Since this handler is using bytecode instructions, we actually need to call our bytecode interpreter, with the `keyboard_handler` address as initial Instruction Counter. We also need to save and restore the [R0-R6] registers, used by the interpreter, before and after this call. In fact we can save only [R4-R6] since the microprocessor already saves and restores [R0-R3] on interrupt entry and return. This gives the following Cortex M3 instructions, which we put after the Hard Fault handler (*i.e.*, starting at C01C4$_{16}$):

LDR	R**2** $\leftarrow$ mem32[$\lfloor$PC$\rfloor_4 + 4 * 2$]	01001**010***00000010*	4A02	000
LDR	R**0** $\leftarrow$ mem32[$\lfloor$PC$\rfloor_4 + 4 * 3$]	01001**000***00000011*	4803	002
PUSH	R4 R**5** R**6** $LR \rightarrow$ stack	1011010*1***01110000**	B570	004
BLX	PC $\leftarrow$ R**0** $- 1$, LR $\leftarrow a + 3$	010001111**0000**000	4780	006
POP	R4 R**5** R**6** $PC \leftarrow$ stack	1011110*1***01110000**	BD70	008
data	(padding, unused)		0000	00A

```
data    (address of keyboard_handler function)          000C06AE   00C
data    (interworking address of interpreter)           000C0001   010
```

where POP moves the LR saved by PUSH into the PC, in order to resume the execution of the interrupted program. Finally, we need to store the interworking address of this USART handler at offset 84_{16} in the Vector Table. The boot assistant commands to do this, and to flash the keyboard driver and the above instructions are provided in `part2/keyboard_driver.txt`. Run them with:

```
user@host:~$ python3 flash_helper.py < part2/keyboard_driver.txt
>Reading page 1028... Done.
Reading page 1029... Done.
Reading page 1030... Done.
Reading page 1025... Done.
Reading page 0... Done.
Writing page 0... Done.
Writing page 1025... Done.
Writing page 1028... Done.
Writing page 1029... Done.
Writing page 1030... Done.
>Done.
```

11.5 Experiments

In order to test our driver we can try to display on the screen each key typed on the keyboard. First of all, we need to connect the keyboard to the Arduino. PS/2 keyboards need a 5V power source and generate 5V CLOCK and DATA signals. The Arduino Due has a 5V output pin, but only supports 3.3V inputs. To connect the two we thus need an adapter, called a *level converter*. This component is essentially a switch on a 3.3V circuit, controlled by a 5V input (see Figure 11.5). The 4 channel Logic Level Converter listed in Table A.1 provides 4 such switches. To use this small board you need to solder header pins on it. For this the easiest is to plug the header pins on the breadboard as shown in Figure 11.4, place the board on top of them, and solder the pins in place (see more detailed instructions at https://ebruneton.github.io/toypc/assembly.html). Then, using jumper wires, connect:

- the GND, LV, and HV pins to the Arduino's GND, 3.3V, and 5V pins,
- the GND and HV pins to the GND and VCC pins of the mini-DIN connector,
- the CLOCK pin of the mini-DIN connector to the Arduino pin SDA1 (corresponding to PA17, see Figure 6.1), via the LV4 / HV4 pins of the Level Converter,
- the DATA pin of the mini-DIN connector to the Arduino pin 19 (corresponding to PA10, see Figure 6.1), via the LV3 / HV3 pins of the Level Converter.

We can then write a main function, at its expected address $C2000_{16}$ (see Figure 9.2), to display on the screen each key typed on the keyboard. We start the main function with calls to `boot_mode_select_rom`, `clock_init`, `gpu_init`, and `keyboard_init`

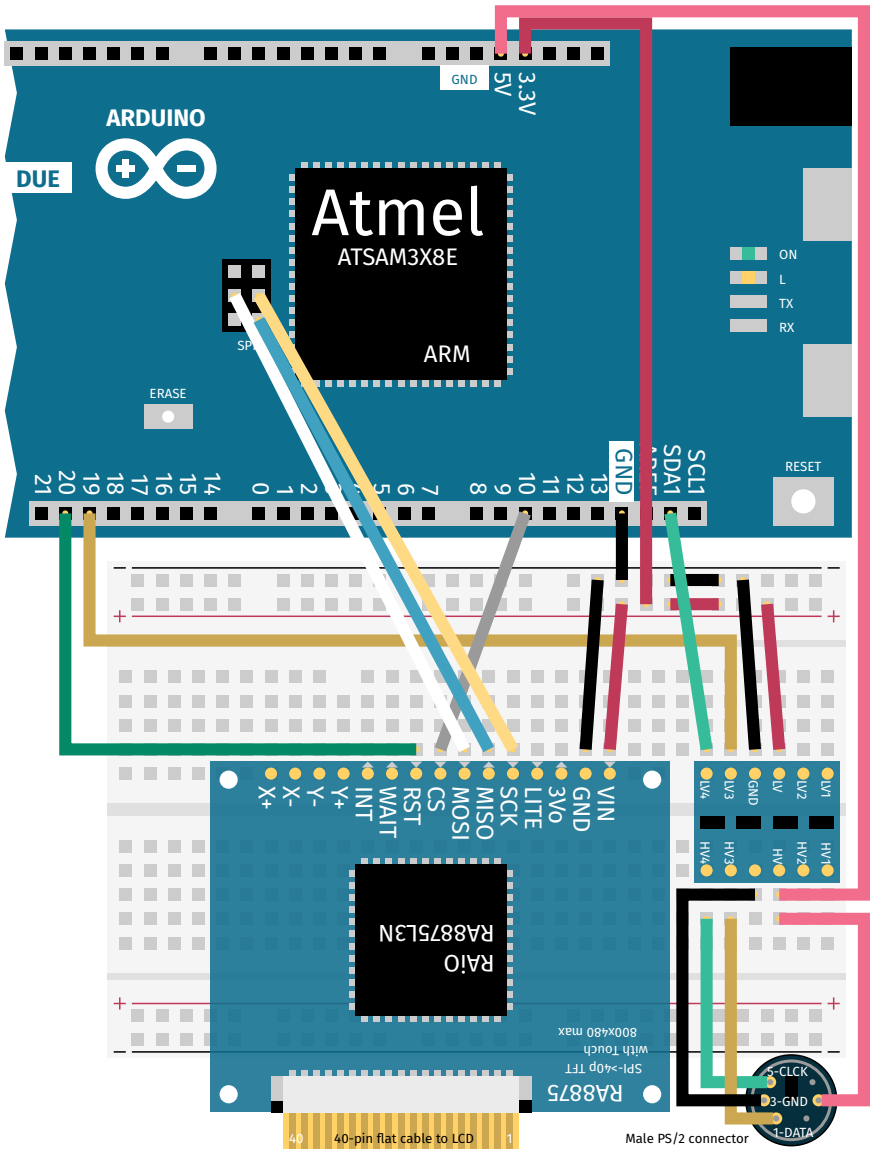


FIGURE 11.4 How to connect the Arduino Due and the keyboard, via a level converter.

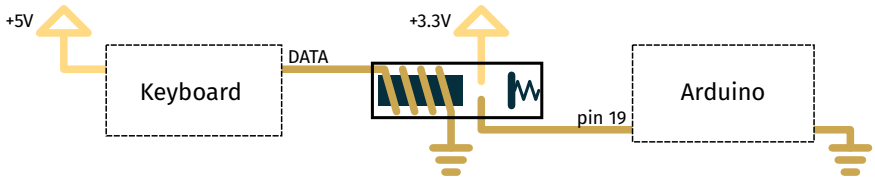


FIGURE 11.5 A schematic diagram of the level converter between the keyboard and the Arduino (actually built with transistors instead of relays).

(see Tables 9.3, 10.3 and 11.4). We then set the foreground color to green $(0, 7, 0)$:

```
fn      00                  19  C2000
call    02B4 …select_rom    1A  +002
call    0200 clock_init     1A  +005
call    03DF gpu_init       1A  +008
call    0578 keyboard_init  1A  +00B
```

```
cst_0                       00  +00E
cst8    07                  02  +00F
cst_0                       00  +011
call    044D gpu_set_color  1A  +012
```

Finally, we draw each character read from the keyboard, in an endless loop:

```
Read a character.
Draw it on the screen.
Repeat.
```

```
call    0606 …wait_char     1A  +015
call    0465 gpu_draw_char  1A  +018
goto    0015                12  +01B
```

To run this main function we need to store it in flash memory and to restart the Arduino with the boot mode selection set to boot from flash. We provide the necessary boot assistant commands in `part2/keyboard_driver_test.txt`. Run them with:

```
user@host:~$ python3 flash_helper.py < part2/keyboard_driver_test.txt
>Reading page 1056... Done.
Writing page 1056... Done.
>Done.
```

If all goes well, you should see a black screen with a blinking cursor on the top left corner. More importantly, if you type something on the keyboard, you should see it on the screen. You can test that the Shift key works as expected, that "Pause" does nothing, etc. Note that non character keys, such as the arrow keys, F1, F2, etc, still draw characters on the screen. This is because the graphics card font has characters for all values in [0-255], including those of the Latin-1 character set (ISO 8859-1). Thus, for instance, pressing the "Delete" key, which emits the scancodes ($E0_{16}$,71_{16}), draws ñ because $F1_{16} = 71_{16} + 128$ corresponds to ñ in Latin-1. When you are done testing the keyboard, turn off the Arduino.

CHAPTER

12 Memory Editor

Our toy computer is now fully assembled, and we have drivers to input and output data with its own keyboard and screen. The last remaining step to make it completely autonomous is to provide a way to enter programs and to run them, with the keyboard. The most basic way to do this is to write a program similar to the boot assistant, but using the keyboard instead of an external computer. This chapter implements such a program, hereafter called the "memory editor". We test it at the end to read the memory, control a LED with the keyboard, and run other programs.

12.1 User interface

A possible user interface for our memory editor would be to use the same interface as the boot assistant. We would thus type commands such as "`wC0000,#`", "`WC0000,1234#`", or "`GC0000#`", to read words, write words, or call functions. And the screen would display the commands and their results. This could lead to a very small program, but would not be very practical. Instead, we propose to make better use of the screen, by displaying a "memory page" at a time (256 bytes). In this way, we can read the memory like a book, instead of one word at a time. We also get more context when editing it, or before calling a function. More precisely, we propose the following user interface:

```
00000003 00000002 00000001 00000000 20000100
00000007 00000006 12345678 00000004 20000110
... 13 other rows ...
000000FF 000000FE 000000FD 000000FC 200001F0

                                2345678A 20000114
```

- A *page view* shows 16 rows of 4 words each, followed by the row's start address. As we did so far in this book, addresses increase *from right to left* and from top to bottom (because this is more adapted to little-endian order than left to right).
- Two *fields* below the page view show a value V and an address A, that the user can edit. When editing V, the current value at address A is highlighted in the page view. This value is not changed until Enter is pressed. When this happens V is written at address A and A is incremented by 4 to edit the next word.

- When editing A nothing else changes until Enter is pressed. When this happens, V is updated to show the value at this address. The page view is updated as well to show the page containing this address.

To simplify the implementation, editing a field can only be done by typing an hexadecimal digit ([0-9] or [a-f] in lowercase). Doing so shifts the current value to the left by one hex digit, and inserts the new hex digit on the right. For instance, if the current value is `12345678`, typing `A` gives `2345678A`.

Only one field can be edited at a time. The currently "selected" field is indicated with a blinking cursor under its least significant digit. Typing "w" selects the address field. Typing Enter when editing the address selects the value field.

Pressing "r" calls the function at the *highlighted address*. This is the address which is highlighted in the page view. It might differ from the value in the address field (if this value has been edited and Enter has not been pressed yet).

Finally, for convenience – this is not strictly necessary – the arrow keys can be used to go to the next or previous word or row. The left (resp. right) arrow increases (resp. decreases) the highlighted address by 4. The up (resp. down) arrow decreases (resp. increases) the highlighted address by 16.

12.2 State variables

The state of the above user interface can be described with only 3 variables:

- $address$: the address of the highlighted word in the page view. This variable is sufficient to determine which page contains the highlighted word, and thus to draw the page view (see the next section).
- $mode$: which field is currently being edited. We use $mode = 0$ if it is the value field, and $mode = 1$ if it is the address field.
- $input$: the current value input with the keyboard. This value is shown either in the value field (if $mode = 0$), or in the address field (if $mode = 1$). When Enter is pressed, it is either written in memory at $address$ (if $mode = 0$), or replaces $address$ (if $mode = 1$).

To simplify the implementation we use 3 more of the 8 General Purpose Backup Registers in the "Controllers" memory region to store these variables (we already used 3 of them for the keyboard driver, see Section 11.4.2). More precisely we use the registers at addresses $400E1A9C_{16}$, $400E1AA0_{16}$, and $400E1AA4_{16}$, respectively (see Figure 6.3 and in Chapter 17 in [8]). For convenience, we provide the following functions to get and set their values, called *getters* and *setters*. We store them after the keyboard driver, at address $C06E4_{16}$:

`med_get_address()` → *address*			
fn	`00`	**19**	`C06E4`
cst	`400E1A9C`	**03**	`+002`
load		**13**	`+007`
retv		**1E**	`+008`
`med_set_address(`*address*`)`			

```
fn      01                  19 C06ED
cst     400E1A9C            03 +002
get     00   address         16 +007
store                       14 +009
ret                         1D +00A
med_get_mode() → mode
fn      00                  19 C06F8
cst     400E1AA0            03 +002
load                        13 +007
retv                        1E +008
med_set_mode(mode)
fn      01                  19 C0701
cst     400E1AA0            03 +002
get     00   mode            16 +007
store                       14 +009
ret                         1D +00A
med_get_input() → input
fn      00                  19 C070C
cst     400E1AA4            03 +002
load                        13 +007
retv                        1E +008
med_set_input(input)
fn      01                  19 C0715
cst     400E1AA4            03 +002
get     00   input           16 +007
store                       14 +009
ret                         1D +00A
```

12.3 Drawing functions

12.3.1 Page view

To draw the page view we first need a function to draw its most basic element, an hexadecimal digit. The following function draws the hexadecimal digit d corresponding to the 4 least significant bits of its argument x ($d = x \wedge F_{16}$). If $d \leq 10$ it should draw a character between "0" and "9", which have contiguous ASCII codes in $[30_{16},39_{16}]$. It thus draws the character $d + 30_{16}$. Otherwise, if $d \geq 10$, it should draw a character between "A" and "F", which have contiguous ASCII codes in $[41_{16},46_{16}]$. It thus draws the character $d - 10 + 41_{16} = y + 37_{16}$:

`gpu_draw_hex_digit(`x`)`

Compute $d = x \wedge F_{16}$. The result is in the 5^{th} stack frame slot, after the function argument and 4 saved registers.

If $d \geq 10$, skip the next two instructions.

Otherwise, push 30_{16} on the stack and skip the next instruction.

Push 37_{16} on the stack.

Add d to the last pushed value (either 30_{16} or 37_{16}) and draw the resulting character with `gpu_draw_char`.

```
fn      01                  19 C0720
get     00   x               16 +002
cst8    0F                  02 +004
and          → d            08 +006
get     05   d               16 +007
cst8    0A                  02 +009
ifge    0013                11 +00B
cst8    30                  02 +00E
goto    0015                12 +010
cst8    37                  02 +013
get     05   d               16 +015
add                         04 +017
call    0465 gpu_draw_char  1A +018
ret                         1D +01B
```

With this we can write a function to draw the two digits of a byte b ($b \gg 4$ and b):

```
gpu_draw_hex_byte(b)
fn    01                  19 C073C
get   00   b              16 +002
cst8  04                  02 +004
lsr                       0B +006
call  0720 …hex_digit     1A +007
get   00   b              16 +00A
call  0720 …hex_digit     1A +00C
ret                       1D +00F
```

which can itself be used to write a function drawing the 4 bytes of a word w ($w \gg 24$, $w \gg 16$, $w \gg 8$, and w):

```
gpu_draw_hex_word(w)
fn    01                  19 C074C
get   00   w              16 +002
cst8  18                  02 +004
lsr                       0B +006
call  073C …draw_hex_byte 1A +007
get   00   w              16 +00A
cst8  10                  02 +00C
lsr                       0B +00E
call  073C …draw_hex_byte 1A +00F
get   00   w              16 +012
cst8  08                  02 +014
lsr                       0B +016
call  073C …draw_hex_byte 1A +017
get   00   w              16 +01A
call  073C …draw_hex_byte 1A +01C
ret                       1D +01F
```

We continue with a function to draw the word at a given address, followed by a space (whose ASCII code is 20_{16}, rendered with `gpu_draw_char`):

```
med_draw_page_word_at(address)
fn    01                  19 C076C
get   00   address        16 +002
load                      13 +004
call  074C …draw_hex_word 1A +005
cst8  20                  02 +008
call  0465 gpu_draw_char  1A +00A
ret                       1D +00D
```

In turn, this can be used to write a function drawing a line of the page view, given its start $address$. This function sets the color to green $(0, 7, 0)$, draws the 4 words at addresses $address + 12$, $address + 8$, $address + 4$, and $address$, sets the color to white $(7, 7, 3)$, and finally draws the start $address$ of the row:

```
med_draw_page_row(address)
fn    01                  19 C077A
cst_0                     00 +002
cst8  07                  02 +003
cst_0                     00 +005
call  044D gpu_set_color  1A +006
get   00   address        16 +009
cst8  0C                  02 +00B
add                       04 +00D
call  076C …page_word_at  1A +00E
get   00   address        16 +011
cst8  08                  02 +013
add                       04 +015
call  076C …page_word_at  1A +016
get   00   address        16 +019
cst8  04                  02 +01B
add                       04 +01D
call  076C …page_word_at  1A +01E
get   00   address        16 +021
call  076C …page_word_at  1A +023
cst8  07                  02 +026
cst8  07                  02 +028
cst8  03                  02 +02A
call  044D gpu_set_color  1A +02C
```

```
get   00   address        16   +02F
call  074C …draw_hex_word  1A   +031
ret                        1D   +034
```

We can finally implement a function to draw the page view. This function takes an $address$ and draws the page containing it, with the word at $address$ highlighted in yellow. The i^{th} page corresponds to addresses in $[256i, 256(i+1)[$. The page containing $address$ is thus such that $i = \lfloor address/256 \rfloor = address \gg 8$, and starts at $256i = i \ll 8 = address \wedge \text{FFFFFF00}_{16}$. In fact, to simplify the highlighting, and to make sure that the word at $address$ is fully contained in the page, we actually use $256i + (address \bmod 4)$ as the page's $base$ address. This gives $base = 256i + (address \wedge 3) = address \wedge \text{FFFFFF03}_{16}$. Note that since each row contains 16 bytes, the j^{th} row in a page starts at address $base + 16j$.

The word to highlight is h words after $base$, with $h = (address - base)/4$. This corresponds to the k^{th} word in the l^{th} row (starting from the right), with $k = h \bmod 4$ and $l = \lfloor h/4 \rfloor$. This, in turn, corresponds to column $c_h = 9(3-k)$ and row $r_h = l$, since each word uses 9 characters (counting the space). Substituting k and l with their values finally gives $(c_h, r_h) = (27 - 9(h \wedge 3), h \gg 2)$.

The `med_draw_page` function follows from the above computations. It first computes $base$, then uses a loop to draw the 16 rows, and finally draws the highlighted word on top, at the above coordinates:

`med_draw_page(`$address$`)`

Compute $base$. The result is in the 5^{th} stack frame slot.

Initialize j to 0 (in the 6^{th} slot).

Set the cursor to $(0, j)$, the top-left corner of the next row to draw.

Draw the next row, starting at $base + 16j = base + (j \ll 4)$.

Increment j by 1 to prepare drawing the next row.

If $j < 16$, go back above to draw the next row.

Compute $h = (address - base)/4$. The result is in the 7^{th} stack frame slot.

```
fn    01                      19   C07AF
get   00   address            16   +002
cst   FFFFFF03                03   +004
and        → base             08   +009
cst_0      → j                00   +00A
cst_0                         00   +00B
get   06   j                  16   +00C
call  0422 …set_cursor        1A   +00E
get   05   base               16   +011
get   06   j                  16   +013
cst8  04                      02   +015
lsl                           0A   +017
add                           04   +018
call  077A …draw_page_row     1A   +019
cst_1                         01   +01C
add                           04   +01D
get   06   j                  16   +01E
cst8  10                      02   +020
iflt  000B                    0C   +022
get   00   address            16   +025
get   05   base               16   +027
sub                           05   +029
cst8  02                      02   +02A
```

	lsr		$\rightarrow h$	**0B**	+02C
Compute $c_h = 27 - 9(h \wedge 3)$.	**cst8**	1B		**02**	+02D
	cst8	09		**02**	+02F
	get	07	h	**16**	+031
	cst8	03		**02**	+033
	and			**08**	+035
	mul			**06**	+036
	sub			**05**	+037
Compute $r_h = \lfloor h/4 \rfloor = h \gg 2$.	**get**	07	h	**16**	+038
	cst8	02		**02**	+03A
	lsr			**0B**	+03C
Set the cursor to these coordinates.	**call**	0422	…set_cursor	**1A**	+03D
Set the color to yellow $(7, 7, 0)$.	**cst8**	07		**02**	+040
	cst8	07		**02**	+042
	cst_0			**00**	+044
	call	044D	gpu_set_color	**1A**	+045
Draw the word at $address$ and return.	**get**	00	$address$	**16**	+048
	call	076C	…page_word_at	**1A**	+04A
	ret			**1D**	+04D

12.3.2 Fields

To draw the fields we first provide two functions to draw arbitrary values in each field. These functions just draw their argument x at the correct column and row coordinates – namely $(27, 18)$ and $(36, 18)$ – and with the correct color (green and white):

med_draw_value(x)

fn	01		**19**	C07FD
cst8	1B		**02**	+002
cst8	12		**02**	+004
call	0422	…set_cursor	**1A**	+006
cst_0			**00**	+009
cst8	07		**02**	+00A
cst_0			**00**	+00C
call	044D	gpu_set_color	**1A**	+00D
get	00	x	**16**	+010
call	074C	…draw_hex_word	**1A**	+012
ret			**1D**	+015

med_draw_address(x)

fn	01		**19**	C0813
cst8	24		**02**	+002
cst8	12		**02**	+004
call	0422	…set_cursor	**1A**	+006
cst8	07		**02**	+009
cst8	07		**02**	+00B
cst8	03		**02**	+00D
call	044D	gpu_set_color	**1A**	+00F
get	00	x	**16**	+012
call	074C	…draw_hex_word	**1A**	+014
ret			**1D**	+017

We use them in the following function to draw the two fields, depending on the current mode. If $mode = 0$, the first half of the function draws $address$ in the address field, and $input$ in the value field. It then sets the cursor under the least significant digit of the value field. Otherwise, if $mode \neq 0$, the second half draws the value stored in memory at $address$ in the value field, and draws $input$ in the address field.

It finally sets the cursor under the least significant digit of the address field:

```
med_draw_fields()
fn      00                  19  C082B
call    06F8 …get_mode      1A  +002
cst_0                       00  +005
ifne    001D                10  +006
call    06E4 …get_address   1A  +009
call    0813 …draw_address  1A  +00C
call    070C …get_input     1A  +00F
call    07FD …draw_value    1A  +012
cst8    22                  02  +015
cst8    12                  02  +017
call    0422 …set_cursor    1A  +019
ret                         1D  +01C
call    06E4 …get_address   1A  +01D
load                        13  +020
call    07FD …draw_value    1A  +021
call    070C …get_input     1A  +024
call    0813 …draw_address  1A  +027
cst8    2B                  02  +02A
cst8    12                  02  +02C
call    0422 …set_cursor    1A  +02E
ret                         1D  +031
```

12.4 Editing functions

We can now implement functions to react to keyboard inputs. These functions update the state variables and redraw the page view and/or the fields, depending on the key typed. We start with a function to enter a new hexadecimal digit x. As specified above, this should shift $input$ to the left by one hex digit, and insert x on the right. This can be done with $input \leftarrow (input \ll 4) + x$. After that the fields must be redrawn (but not the page view). Characters have an opaque black background, so we don't need to erase the previous values before drawing new ones:

```
med_new_digit(x)
fn      01                  19  C085D
call    070C …get_input     1A  +002
cst8    04                  02  +005
lsl                         0A  +007
get     00  x               16  +008
add                         04  +00A
call    0715 …set_input     1A  +00B
call    082B …draw_fields   1A  +00E
ret                         1D  +011
```

A function to enter a new highlighted address x is also useful, since several keys can change this address (the Enter key and the arrow keys). The following function sets $address$ to x, sets $input$ to the current value at this address, and finally redraws the page view and the fields:

```
med_new_address(x)
fn      01                  19  C086F
get     00  x               16  +002
call    06ED …set_address   1A  +004
get     00  x               16  +007
load                        13  +009
call    0715 …set_input     1A  +00A
get     00  x               16  +00D
call    07AF …draw_page     1A  +00F
call    082B …draw_fields   1A  +012
ret                         1D  +015
```

With this we can write a function to handle the Enter key. Pressing this key has two different effects, depending on the current $mode$. If $mode = 0$ the value field

is being edited. In this case we want to store $input$ in memory at $address$, and set $address + 4$ as the new highlighted address. This is what the first part of the function does, after a test of the $mode$ value. The second part, after the first `ret`, handles the case $mode \neq 0$: it sets $mode$ to 0 and the new address to $input$:

```
med_handle_enter()
fn      00                  19  C0885
call    06F8 …get_mode      1A  +002
cst_0                       00  +005
ifne    001A                10  +006
call    06E4 …get_address   1A  +009
call    070C …get_input     1A  +00C
store                       14  +00F
call    06E4 …get_address   1A  +010
cst8    04                  02  +013
add                         04  +015
call    086F …new_address   1A  +016
ret                         1D  +019
cst_0                       00  +01A
call    0701 …set_mode      1A  +01B
call    070C …get_input     1A  +01E
call    086F …new_address   1A  +021
ret                         1D  +024
```

We finally provide a function to handle any character c typed on the keyboard. This function is quite long because there are "many" characters to support. But its overall structure is regular. It is made of several sequences of instructions S_i, one for each supported character (*e.g.*, "w" or "r") or range of characters (*e.g.*, ["0"-"9"] or ["a"-"f"]). Each sequence S_i starts with one or two conditional jumps to go to the next sequence S_{i+1}, if c is not a character handled by S_i. The sequence continues with instructions to handle this character, and ends with a `ret`:

`med_handle_char(`c`)`

S_0: decimal digits [0-9]. If c < "0" (ASCII code 30_{16}), go to S_1.

If c > "9" (ASCII code 39_{16}), go to S_1.

Otherwise, enter the new hex digit $d = c - 30_{16}$ and return.

S_1: hexadecimal digits [a-f]. If c < "a" (ASCII code 61_{16}), go to S_2.

If c > "f" (ASCII code 66_{16}), go to S_2.

Otherwise, enter the new hex digit $d = c - 61_{16} + 10 = c - 57_{16}$ and return.

```
fn      01                  19  C08AA
get     00   c              16  +002
cst8    30                  02  +004
iflt    0019                0C  +006
get     00   c              16  +009
cst8    39                  02  +00B
ifgt    0019                0E  +00D
get     00   c              16  +010
cst8    30                  02  +012
sub                         05  +014
call    085D …new_digit     1A  +015
ret                         1D  +018
get     00   c              16  +019
cst8    61                  02  +01B
iflt    0030                0C  +01D
get     00   c              16  +020
cst8    66                  02  +022
ifgt    0030                0E  +024
get     00   c              16  +027
cst8    57                  02  +029
sub                         05  +02B
```

S_2: Enter key. If c is not Enter (ASCII code $0A_{16}$), go to S_3.

Otherwise, handle this key and return.

S_3: "w" character. If c is not "w" (ASCII code 77_{16}), go to S_4.

Otherwise, handle this key: set $address$ as the new $input$, set $mode$ to 1, redraw the fields, and return.

S_4: "r" character. If c is not "r" (ASCII code 72_{16}), go to S_5.

Otherwise, call the function at $address$, clear and redraw the screen (the called function might have changed it), and return.

S_5: ArrowLeft character. If c is not ArrowLeft (character $6B_{16}$+128=EB_{16}), go to S_6.

Otherwise, set $address + 4$ as the new address and return.

S_5: ArrowRight character. If c is not ArrowRight (character 74_{16}+128=$F4_{16}$), go to S_6.

Otherwise, set $address - 4$ as the new address and return.

S_6: ArrowUp character. If c is not ArrowUp (character 75_{16}+128=$F5_{16}$), go to S_7.

```
call  085D …new_digit      1A  +02C
ret                        1D  +02F
get   00    c              16  +030
cst8  0A                   02  +032
ifne  003B                 10  +034
call  0885 …handle_enter   1A  +037
ret                        1D  +03A
get   00    c              16  +03B
cst8  77                   02  +03D
ifne  0050                 10  +03F
call  06E4 …get_address    1A  +042
call  0715 …set_input      1A  +045
cst_1                      01  +048
call  0701 …set_mode       1A  +049
call  082B …draw_fields    1A  +04C
ret                        1D  +04F
get   00    c              16  +050
cst8  72                   02  +052
ifne  0065                 10  +054
call  06E4 …get_address    1A  +057
calld                      1C  +05A
call  0409 …clear_screen   1A  +05B
call  06E4 …get_address    1A  +05E
call  086F …new_address    1A  +061
ret                        1D  +064
get   00    c              16  +065
cst8  EB                   02  +067
ifne  0076                 10  +069
call  06E4 …get_address    1A  +06C
cst8  04                   02  +06F
add                        04  +071
call  086F …new_address    1A  +072
ret                        1D  +075
get   00    c              16  +076
cst8  F4                   02  +078
ifne  0087                 10  +07A
call  06E4 …get_address    1A  +07D
cst8  04                   02  +080
sub                        05  +082
call  086F …new_address    1A  +083
ret                        1D  +086
get   00    c              16  +087
cst8  F5                   02  +089
ifne  0098                 10  +08B
```

Otherwise, set $address - 16$ as the new address and return.

```
call  06E4 …get_address   1A  +08E
cst8  10                  02  +091
sub                       05  +093
call  086F …new_address   1A  +094
ret                       1D  +097
```

S_7: ArrowDown character. If c is not ArrowDown (character 72_{16}+128=F2$_{16}$), go to S_8.

Otherwise, set $address + 16$ as the new address and return.

```
get   00    c             16  +098
cst8  F2                  02  +09A
ifne  00A8                10  +09C
call  06E4 …get_address   1A  +09F
cst8  10                  02  +0A2
add                       04  +0A4
call  086F …new_address   1A  +0A5
ret                       1D  +0A8
```

S_8: any other character. Return.

12.5 Main function

We can finally implement the last function of the memory editor, and of our basic input output system! This function initializes the drivers, sets the initial $mode$ and $address$ to 0, and finally calls the previous function for each new character typed on the keyboard, in an endless loop:

```
memory_editor()
fn    00                  19  C0953
call  0200 clock_init     1A  +002
call  03DF gpu_init       1A  +005
call  0578 keyboard_init  1A  +008
cst_0                     00  +00B
call  0701 …set_mode      1A  +00C
cst_0                     00  +00F
call  086F …new_address   1A  +010
call  0606 …wait_char     1A  +013
call  08AA …handle_char   1A  +016
goto  0013                12  +019
```

Note that this function does *not* change the boot mode selection to boot from ROM at the next reset. Indeed, with the memory editor, our toy computer is now completely autonomous, and we no longer need the boot assistant nor an external computer to use it. By putting together the code of all the functions defined in this chapter we get the memory editor's full code:

```
A0030019 1D140016 400E1A9C 03011910 13400E1A 9C030019  C06E4
13400E1A A4030019 1D140016 400E1AA0 03011910 13400E1A  C06FC
110A0205 16080F02 00160119 1D140016 400E1AA4 03011910  C0714
1A0B0402 00160119 1D04651A 04051637 02001512 30020013  C072C
1A0B1002 0016073C 1A0B1802 00160119 1D07201A 00160720  C0744
074C1A13 00160119 1D073C1A 0016073C 1A0B0802 0016073C  C075C
16076C1A 040C0200 16044D1A 00070200 01191D04 651A2002  C0774
07020702 076C1A00 16076C1A 04040200 16076C1A 04080200  C078C
16000008 FFFFFF03 03001601 191D074C 1A001604 4D1A0302  C07A4
000B0C10 02061604 01077A1A 040A0402 06160516 04221A06  C07BC
```

```
0B020207 16050608 03020716 09021B02 0B020205 05160016  C07D4
1A12021B 02011191D 076C1A00 16044D1A 00070207 0204221A  C07EC
04221A12 02240201 191D074C 1A001604 4D1A0007 02000422  C0804
001D1000 06F81A00 191D074C 1A001604 4D1A0302 07020702  C081C
1306E41A 1D04221A 12022202 07FD1A07 0C1A0813 1A06E41A  C0834
04020700 1A01191D 04221A12 022B0208 131A070C 1A07FD1A  C084C
07151A13 001606ED 1A001601 191D082B 1A07151A 04001600  C0864
070C1A06 E41A001A 100006F8 1A00191D 082B1A07 AF1A0016  C087C
01191D08 6F1A070C 1A07011A 001D086F 1A040402 06E41A14  C0894
161D085D 1A053002 00160019 0E390200 16001900 30020016  C08AC
00161D08 5D1A0557 02001600 300E6602 00160030 0C610200  C08C4
1A010715 1A06E41A 00501077 02001610 08851A00 3B100A02  C08DC
1A06E41A 04091A1C 06E41A00 65107202 00161D08 2B1A0701  C08F4
F4020016 1D086F1A 04040206 E41A0076 10EB0200 161D086F  C090C
0206E41A 009810F5 02001610 086F1A05 040206E4 1A008710  C0924
191D086F 1A041002 06E41A00 A810F202 00161D08 6F1A0510  C093C
08AA1A06 061A086F 1A000701 1A000578 1A03DF1A 02001A00  C0954
                                                 001312  C096C
```

Lets store it in flash memory. The boot assistant commands to do this are provided in `part2/memory_editor.txt`. Run them with:

```
user@host:~$ python3 flash_helper.py < part2/memory_editor.txt
>Reading page 1030... Done.
Reading page 1031... Done.
Reading page 1032... Done.
Reading page 1033... Done.
Writing page 1030... Done.
Writing page 1031... Done.
Writing page 1032... Done.
Writing page 1033... Done.
>Done.
```

Note that the `memory_editor` function is not stored at the $C2000_{16}$ address used so far for main functions (see Figure 9.2). To run it on reset we need to change the $C2000_{16}$ value, at address $C0188_{16}$ in the Reset handler (see Section 9.6.1), to the `memory_editor` function address, namely $C0953_{16}$. Do this as follows:

```
user@host:~$ python3 flash_helper.py
>WC0188,C0953#
Reading page 1025... Done.
>flash#
Writing page 1025... Done.
```

The final layout of our basic input output system is shown in Figure 12.1, and its most important functions are listed in Table 12.1. In total, this system consists of 669 bytecode instructions, plus 364 bytes of data, for a total size of 1900 bytes. To which we must add 199 Cortex M3 instructions for the virtual machine interpreter and the

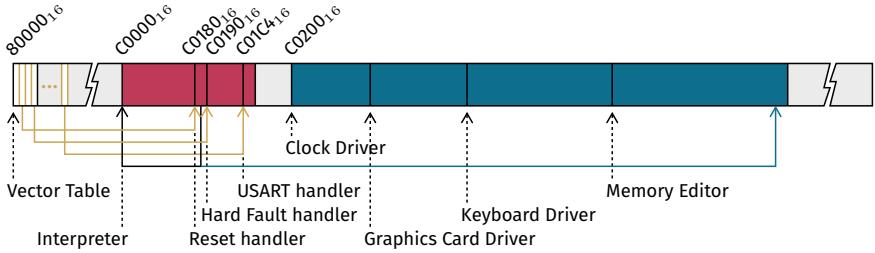


FIGURE 12.1 The final layout of our basic input output system in flash memory. Red, blue and gray areas represent machine code, bytecode and unused memory, respectively (not to scale).

Reset, Hard Fault and USART handlers, plus 64 bytes of data, for a total of 472 bytes.

12.6 Experiments

You can now set the Arduino to boot from flash and restart it:

```
>reset#
```

If all goes well, you should see the memory editor's user interface on the screen, showing the first memory page, starting at address 0. Since we removed the function call to change the boot mode selection, this memory page is mapped to flash memory. More precisely, it is mapped to the page starting 80000_{16}, which contains the Vector Table. You should recognize, on the first line, the Vector Table entries given at the end of Section 9.6.1:

```
000C01A5 00000000 000C0181 20088000 00000000
```

They are followed by empty entries ($FFFFFF_{16}$), except the one for the USART handler. Lets try another memory location. On the Arduino's keyboard, type "w000c0000" followed by Enter. You should now see a new memory page, starting with

```
F000E8DF DE00BF88 281F3201 78102600 000C0000
```

This corresponds to the beginning of our virtual machine interpreter, which is indeed stored at this address (see Section 8.3.9). Similarly, you can type "w000c0200" followed by Enter to look at the clock driver code. You should see a new memory page starting with the code shown at the end of Section 9.5. In the same way, you can display the graphics card and keyboard driver code, and even the memory editor code.

Lets now try to store some values in memory. Type "w20070000"+Enter to go the RAM region. Then type "12345678"+Enter to store this value at this address. You should see the new value in the page view. We can use this method to control the Arduino's LED, as we did in Section 6.6.2, but with our memory editor instead of the boot assistant. Type "w400e1000"+Enter to show the PIO B registers. You should

Function	Address
boot_mode_select_rom()	C02B4 ($C0000_{16}$+692)
clock_init()	C0200 ($C0000_{16}$+512)
delay(n)	C028B ($C0000_{16}$+651)
gpu_clear_screen()	C0409 ($C0000_{16}$+1033)
gpu_draw_char(c)	C0465 ($C0000_{16}$+1125)
gpu_draw_hex_byte(b)	C073C ($C0000_{16}$+1852)
gpu_draw_hex_digit(x)	C0720 ($C0000_{16}$+1824)
gpu_draw_hex_word(w)	C074C ($C0000_{16}$+1868)
gpu_init()	C03DF ($C0000_{16}$+991)
gpu_set_color(r, g, b)	C044D ($C0000_{16}$+1101)
gpu_set_cursor(c, r)	C0422 ($C0000_{16}$+1058)
gpu_set_register(id, $value$)	C0378 ($C0000_{16}$+888)
keyboard_get_char() $\rightarrow c$	C05E0 ($C0000_{16}$+1504)
keyboard_handler()	C06AE ($C0000_{16}$+1710)
keyboard_init()	C0578 ($C0000_{16}$+1400)
keyboard_wait_char() $\rightarrow c$	C0606 ($C0000_{16}$+1542)
load_byte($address$) $\rightarrow value$	C03C0 ($C0000_{16}$+960)
memory_editor()	C0953 ($C0000_{16}$+2387)
spi_transfer($value$) $\rightarrow response$	C034E ($C0000_{16}$+846)

TABLE 12.1 The most important functions of the basic input output system.

note that all registers are 0, except the PIO Status Register ($400E1008_{16}$), as well as the Status, Output Status, Output Data Status, and Pull-up Status Registers. The bit 12 of these registers is 1, indicating that the PB12 pin is an output pin, controlled by the microprocessor, currently set to 1, and with its pull-up resistor disabled. Indeed, this is the pin used to reset the graphics card, configured in `gpu_reset` (see Section 10.4). We can now redo the experiments of Section 6.6.2: press the ArrowDown key to select the $400E1010_{16}$ address and then type "08000000"+Enter: you should see the LED turning off. Using the arrow keys, select the $400E1030_{16}$ address and type "08000000"+Enter: you should see the LED turning on again.

We can also try to display a reserved memory region. Reading the memory in these regions causes an exception which should be handled by our Hard Fault handler, which blinks the LED very fast. To test this type "w00200000"+Enter (see Figure 6.3). You should see the LED blinking. Moreover, typing any key should have no effect, since the memory editor effectively crashed. We could restart it by pressing the RESET button. Instead, to check that our toy computer is completely autonomous, unplug it from your computer and plug it to a power outlet with a phone charger. The memory editor should be running again.

As a last experiment, lets try to run a function. We can use `boot_mode_select_rom`

for this. Type "w000c02b4"+Enter to go to this function (see Table 12.1). Then type "r". Nothing changes on the screen, but the Arduino is now configured to boot from ROM, *i.e.*, to run the boot assistant, at the next reset. To verify this, unplug the Arduino from the power outlet and plug it again to your computer. The screen should stay off. We can verify that the boot assistant is running as follows:

```
user@host:~$ python3 flash_helper.py
>V#
v1.1 Dec 15 2010 19:25:04
```

We can then change again the boot mode selection to run from flash, and reset the Arduino, with:

```
>reset#
```

At this point the memory editor should be running again. The above method can be used to temporarily connect our toy computer to an external computer. This can be useful, for instance, to do a backup of its flash memory. Such a backup can be done by reading all the flash memory with boot assistant "w" commands, and storing the result in a file on the host computer. This file can then be used to restore the flash memory, with boot assistant "W" commands.

Conclusion

A microcontroller contains in a single chip a microprocessor, various memories, and specialized circuits to communicate and interact with the external world. In this part we connected an Arduino Due, based on an Atmel microcontroller, to a Liquid Crystal Display (LCD), and to a keyboard. We used its boot program in read-only memory, via an external computer, to store in its flash memory a basic memory editor program. Like the Atmel's boot program, this editor allows us to input programs and to run them. However, it does this by using the keyboard and the LCD connected to the Arduino, *i.e.*, without needing any external computer. Our toy computer is thus fully autonomous. The rest of this book illustrates this by progressively improving it, to make it more and more usable, without using any external computer.

Further readings

This part barely scratched the surface of what microcontrollers can do. And it only presented a very low level method to program them (because the goal of this book it to program one from scratch). To know more about what microcontrollers can do, how they work, and how to use them in more convenient ways, you can read the following books:

- "Arduino Workshop, 2nd Edition: A Hands-on Introduction with 65 Projects" [4]. This book gives a very practical introduction to the Arduino microcontrollers. It explains how they can be programmed with the Arduino Integrated Development Environment, and shows how they can be used with many external components (including 7-segments displays, micro-SD cards, keypads, touchpads, motors, infrared sensors, GPS modules, external memories, etc).
- "Embedded System Design with ARM Cortex-M Microcontrollers: Applications with C, C++ and MicroPython" [22]. This book explains how microcontrollers work in general, and gives practical examples illustrating each aspect. For instance, it presents digital and analog signals, conversions between digital and analog signals, interrupts, clocks and timers, communication protocols, etc.
- "Embedded Systems Fundamentals with Arm Cortex-M Based Microcontrollers: A Practical Approach" [9]. This book also gives "theoretical" and practical information about microcontrollers, with an emphasis on how to use them "properly" (*i.e.*, to get efficient and responsive programs with low power requirements).

These books use programs written in textual form, which is much more practical than hexadecimal numbers representing machine code or bytecode instructions. Some of the above books introduce the *programming language* that they use for this, at the

same time as they present microcontrollers. But none of them explains in detail how a program written in textual form can run on a microprocessor, which can only execute machine code instructions. This process is introduced in the next part, based on a toy programming language for our toy computer.

PART

3 A Toy Compiler

Introduction

Our toy computer is now fully assembled and autonomous. However, it is still very hard to use. Indeed, even if our virtual machine instructions are much simpler than ARM instructions, they remain difficult to use. The goal of this part is thus to provide an easier way to program our toy computer.

To illustrate how this can be done, consider the task of writing a program to compute the factorial of a number n, defined by $factorial(n) = 1 * 2 * \ldots * n$ if $n > 0$, and 1 otherwise. One way to write such a program is to use the property that, for $n > 0$, $factorial(n) = factorial(n-1) * n$. This leads to the following bytecode instructions:

factorial(n)				get	00	n		16	+00A
fn	01		**19** C1000	**cst_1**				**01**	+00C
get	00	n	**16** +002	**sub**				**05**	+00D
cst_0			**00** +004	**call**	1000	factorial		**1A**	+00E
ifne	000A		**10** +005	**get**	00	n		**16**	+011
cst_1			**01** +008	**mul**				**06**	+013
retv			**1E** +009	**retv**				**1E**	+014

The first part, in the left column, compares n, the 0^{th} word in the function's stack frame, with 0. If it is not equal to 0, it jumps to the second part, in the right column. Otherwise it returns 1. The second part, in the right column, subtracts 1 from n, calls the factorial function with this argument (we assume here that it is stored at $C1000_{16}$), and returns the result multiplied by n. In order to write this program we need to type on the keyboard the following *numbers*:

```
1E 06001610 001A0501 00161E01 000A1000 00160119  C1000
```

Typing numbers is error prone, and understanding their meaning requires a lot of effort. It would be much easier if we could type the following *text* instead:

```
fn 1
  get 0 cst_0 ifne 10 cst_1 retv
  get 0 cst_1 sub call 4096 get 0 mul retv
```

In fact this text still contains some numbers, such as 0, 10=A_{16} and 4096=1000_{16}. The last two are quite tedious to compute. For instance, one must sum the size of all the instructions up to the first `retv` (included) to get the value 10. Function addresses such as 4096 also require to compute the bytecode size of each function to keep track

of their addresses. To avoid having to do this, it would be simpler if we could use *labels* to designate instructions, and *identifiers* to designate functions:

```
fn factorial 1
  get 0 cst_0 ifne not_zero cst_1 retv
  :not_zero get 0 cst_1 sub call factorial get 0 mul retv
```

Similarly, to get rid of the last numbers, it would be useful to be able to give names to the function parameters. We could then replace 1, the number of parameters of the factorial function, with a list of parameter names. And we could replace 0 with n:

```
fn factorial(n)
  get n cst_0 ifne not_zero cst_1 retv
  :not_zero get n cst_1 sub call factorial get n mul retv
```

This text would already be much easier to type and to understand than the above numbers. However, it still contains long sequences of bytecode instructions which are more complex than the mathematical *expressions* they compute. For instance, `get n cst_1 sub call factorial get n mul` computes $factorial(n-1) * n$. It would be much easier if we could use these expressions directly (this example also adds some "punctuation" signs, namely curly braces, commas, and semi-colons):

```
fn factorial(n) {
  n, 0 ifne not_zero; 1 retv;
  :not_zero factorial(n - 1) * n retv;
}
```

It would also be more natural if we could write the last remaining bytecode instructions in a different order, closer to the order of words in English. And instead of writing "if n is not 0, jump to :not_zero to not return 1", it would be simpler to write "if n is 0, return 1". We could even get rid of the label by putting the instructions to execute when n is 0 inside curly braces:

```
fn factorial(n) {
  if n == 0 { return 1; }
  return factorial(n - 1) * n;
}
```

Finally, to make it clear that this function returns a value (some do not), and takes a number as parameter, it would be practical to have some *type declarations*, such as the following (where `u32` means an "unsigned 32 bit" value):

```
fn factorial(n: u32) -> u32 {
  if n == 0 { return 1; }
  return factorial(n - 1) * n;
}
```

In fact the goal of this part is to be able to type programs in this form (inspired from Rust [15]), and to *automatically* get the corresponding bytecode instructions, in numerical form (also called *binary* form). For this we write a program, called a *compiler*, which transforms the program text, called the *source code*, into *compiled code*, *i.e.*, bytecode instructions in binary form. This compiler is a large program, and we can't write it in source code because we don't have a compiler yet! On the other hand, writing it in binary form would be very hard. To solve this problem we write the compiler in several steps:

1. Write a small compiler to compile textual bytecode instructions (*e.g.*, `fn 1 get 0` ...). Write this *opcodes compiler* in binary form.
2. Write a compiler for programs using textual bytecode instructions with function names and instruction labels. Write this *labels compiler*, also called an *assembler*, with pure textual bytecode instructions. Compile it with the opcodes compiler.
3. Rewrite the labels compiler with function names and labels, and improve it so that it can compile programs using expressions such as `factorial(n - 1) * n`. Compile this *expressions compiler* with the labels compiler.
4. Rewrite the expressions compiler with expressions, and improve it in order to support programs using *statements* such as `if` and `return`. Compile this *statements compiler* with the expressions compiler.
5. Rewrite the statements compiler with statements, and improve it to accept programs with type declarations such as `factorial(n: u32) -> u32 { ... }`. Compile this *types compiler* with the statements compiler.

With this method, only the first compiler needs to be written in binary form. And this compiler is small because its task is quite simple. Moreover, each step is easier to do than the previous one because it can use simplifying features introduced in the previous steps. However, in order to implement this method, we need a way to type text, *i.e.*, a *text editor*. Indeed, all we have for now is the memory editor, and entering text in memory with it would require typing the ASCII code *numbers* corresponding to each character of the text! Obviously, this would be even worse than entering programs directly in binary form, since a program in text form is usually longer than in binary form. Unfortunately, we can't write a text editor program in any textual form, since we don't have a text editor yet! We thus start by writing a simple text editor, directly in binary form.

Before all this, however, we also need a way to store programs in source or binary form in flash memory. Otherwise, we would loose everything we typed if we ever do a mistake causing a crash. For this we implement one more driver, called the flash memory driver (we didn't need this in the previous part because flashing programs was done from the external computer). This driver also needs to be implemented in binary form. The rest of this part presents the above steps in detail. It is organized as follows:

- Chapter 13 presents the flash memory driver, used to read and write data in flash memory. We use it at the end to save itself in flash memory.

- Chapter 14 explains how our text editor works, and presents its implementation in binary form.
- Chapter 15 explains how the opcodes compiler works, and presents its implementation in binary form.
- Chapter 16 explains how the labels compiler works, and presents its implementation in textual form.
- Chapter 17 explains what expressions are, how they can be compiled, and presents the ones supported by our toy compiler. It then gives the corresponding implementation.
- Chapter 18 does the same with statements.
- Chapter 19 does the same with type declarations.
- Chapter 20 finally provides a new version of our toy compiler which produces ARM instructions instead of bytecode instructions. We use it in the next part to eventually get rid of our virtual machine.

CHAPTER

13 Flash Memory Driver

Before writing our toy compiler, as described in introduction, we need a way to save it in flash memory. In theory, our memory editor provides everything we need to do this. Indeed, as we have seen in the previous part, we can save a 64 words page in flash memory by writing these words at their final address, and then by writing an appropriate value in the Enhanced Embedded Flash Controller (EEFC) Command Register (which has a well defined address in memory). All these steps can be done manually with the memory editor, but doing so would not be very practical. To make this is easier we provide in this chapter a few helper functions, called the flash memory driver. We use them at the end to store themselves in flash memory.

13.1 Overview

Due to the flash memory usage constraints, it is not practical to directly edit data here. Instead, what we can do is edit data in RAM, and then save it in flash memory. The latter step can be done automatically (see Figure 13.1):

- copy the first 64 words of data from RAM to flash memory (we assume in this chapter that data is always saved at the beginning of a flash memory page),
- save them by writing the appropriate command in the EEFC Command Register,
- ...
- copy the last n words of data from RAM to flash memory, and the $64 - n$ remaining words from flash memory to itself (recall that we cannot save a page without writing all its 64 words first),
- save them by writing the appropriate command in the EEFC Command Register.

Conversely, to edit data which is already in flash memory, we can copy it in RAM, edit it here, and then save it back. Note that these algorithms use two kinds of steps: copying memory from one address to another, and saving a page of flash memory. The rest of this section presents them in more detail.

13.1.1 Memory copy

As explained above, our flash memory driver needs a function to copy some data from one address to another. Our text editor, presented in the next chapter, also needs

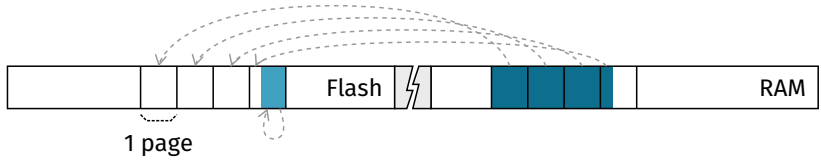


FIGURE 13.1 Saving data (in dark blue) from RAM into flash memory must be done page by page. Each 64 words page must be copied (dashed arrows) and then saved. The unused words of the last page (light blue) must be copied in place so that all the page words are written before it is saved (as required by the EEFC component).

ALGORITHM 13.1 Copying n bytes from src to dst.

1. if $dst < src$
2. initialize i to 0
3. while $i + 4 \leq n$, copy the word at $src + i$ to $dst + i$ and then increment i by 4
4. while $i < n$, copy the byte at $src + i$ to $dst + i$ and then increment i by 1
5. otherwise
6. initialize i to n
7. while $i \geq 4$, decrement i by 4 and then copy the word at $src + i$ to $dst + i$
8. while $i > 0$, decrement i by 1 and then copy the byte at $src + i$ to $dst + i$

such a function. The former only needs to copy words, between two distinct memory regions. But the latter needs to copy bytes, between two regions which can overlap. To support both use cases, we present here a general memory copy algorithm.

The basic algorithm to copy n bytes starting at address src to address dst is very simple. We just need to store the byte loaded from address $src + i$ to address $dst + i$, for all $i \in [0, n[$. However, the order in which these operations are done is important if the source and destination regions overlap. Consider for instance the task of copying $n = 10$ bytes from $src = 4$ to $dst = 7$ (see Figure 13.2). Starting by copying the byte at $src + 0$ to $dst + 0$ would override the byte at $src + 3$, leading to an incorrect result. The solution is to copy the bytes in decreasing order, from $i = n - 1$ to $i = 0$. Conversely, copying $n = 10$ bytes from $src = 7$ to $dst = 4$ must be done in increasing order (starting by copying the byte at $src + 9$ to $dst + 9$ would override the byte $src + 6$). In summary, bytes must be copied in decreasing order if $dst \geq src$, and in increasing order otherwise. Note also that we can start by copying words, and use byte copies only for the last 1 to 3 remaining bytes, leading to Algorithm 13.1.

13.1.2 Page flash

Once the 64 words of a page have been copied from RAM to flash memory, they can be saved by writing the appropriate value in the EEFC Command Register. One must then wait until the EEFC Status Register value is 1, indicating that the operation is done. During this time, the flash memory bank must not be used (see Section 6.5).

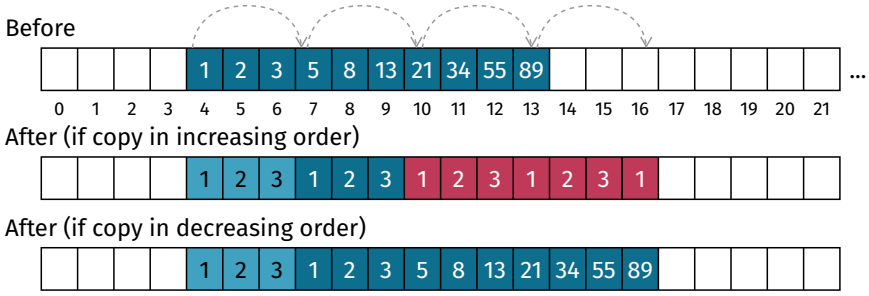


FIGURE 13.2 Copying 10 bytes (dark blue) from address 4 to address 7 in increasing order (*i.e.*, from byte 4 to byte 13) leads to incorrect results (in red). Copying them in decreasing order, from byte 13 to byte 4, solves the problem.

Unfortunately, our virtual machine *is* in flash memory. Therefore, we can't use a bytecode function to read the EEFC Status Register (running it would require the microprocessor to read the ARM instructions of the virtual machine, *i.e.*, would use the flash memory). A solution is to use a small subroutine made of ARM instructions, stored in RAM, to save a page in flash memory without using it. This subroutine can be implemented as follows:

PUSH	$\mathrm{R0}\ \mathrm{R1}\ LR \rightarrow \mathrm{stack}$	1011010**1**00000011	B503	000
LDR	$\mathrm{R0} \leftarrow \mathrm{mem32}[\lfloor \mathrm{PC} \rfloor_4 + 4 * 4]$	01001**000**00000100	4804	002
LDR	$\mathrm{R1} \leftarrow \mathrm{mem32}[\lfloor \mathrm{PC} \rfloor_4 + 4 * 4]$	01001**001**00000100	4904	004
STR	$\mathrm{R1} \rightarrow \mathrm{mem32}[\mathrm{R0} + 4 * 0]$	01100000000**001**	6001	006
LDR	$\mathrm{R1} \leftarrow \mathrm{mem32}[\mathrm{R0} + 4 * 1]$	01101000010**001**	6841	008
CMP	$compare(\mathrm{R1}, 1)$	00101**001**00000001	2901	00A
IT	if $\neq$ then	10111111**0001**000	BF18	00C
B	$\mathrm{PC} \leftarrow \mathrm{PC} + 2 * \mathbf{2043} - 4096$	11100**11111111011**	E7FB	00E
POP	$\mathrm{R0}\ \mathrm{R1}\ PC \leftarrow \mathrm{stack}$	1011110**1**00000011	BD03	010
data	(padding, unused)		0000	012
data	(EEFC1 Command Register)		400E0C04	014
data	(EEFC1 Command)		5A000003	018

It starts with a PUSH to save the R0 and R1 registers, as well as the Link Register (LR). It then loads the address of the EEFC Command Register in R0, and the value to write into it in R1, with two LDR instructions (this data is stored just after the function itself). The next instruction actually stores this value in the Command Register, thereby starting the flashing process. The next LDR instruction loads the value of the EEFC Status Register (whose address is 4 bytes after the Command Register address, *i.e.*, R0 + 4). The following CMP instruction compares this value with 1. If it is not equal to 1, the B instruction jumps back to the LDR instruction to read the Status Register again. Otherwise this instruction is skipped and the final POP

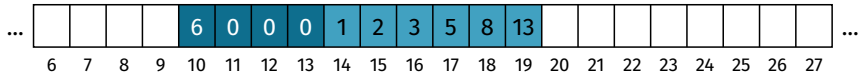


FIGURE 13.3 A data buffer containing 6 bytes of data (light blue), starting at address 10, begins with a 4 bytes header (dark blue) containing the size of the following data.

restores the R0 and R1 registers, and returns to the caller by moving the saved LR into the Program Counter (PC). The first data word after that contains the address of the EEFC1 Command Register (we want to store our compiler in the same flash memory bank as our basic input output system). The final word is the command to write at this address in order to flash the 0^{th} page. To flash the p^{th} page instead, p should be put in bytes 1 and 2 of this word (*e.g.*, to flash the 4^{th} page, use $5A000403_{16}$ – see Section 6.5). In summary, the complete code of this subroutine is:

```
400E0C04 0000BD03 E7FBBF18 29016841 60014904 4804B503   000
                                                5A000003   018
```

13.1.3 Data buffers

In order to copy or save data we must know their address, but we must also know their size in bytes. To avoid having to manually keep track of this size, we can store it in memory too. One way to do this is to store it in a word located just before the data themselves (see Figure 13.3). This word is called a *header* or a *metadata* (because it is some data about other data). In the following we call this header and its associated data a *data buffer*. And we provide functions to copy and save data buffers.

13.2 Implementation

We can now implement the above algorithms. We do this in a data buffer, so that our flash memory driver can save itself. This buffer must be saved at the start of a page, as we assumed at the beginning. Lets use the next page after our memory editor, page 10, starting at $C0A00_{16}$. The code will thus start at $C0A04_{16}$, after the header.

For Algorithm 13.1 we need functions to load and store a single byte. We already have a `load_byte` function (see Table 12.1), but we don't have a `store_byte` one. We thus provide one, as follows:

```
store_byte(ptr, value)
fn      02                  19  C0A04
get     00   ptr             16  +002
get     00   ptr             16  +004
load                        13  +006
cst     FFFFFF00            03  +007
and                         08  +00C
get     01   value           16  +00D
or                          09  +00F
store                       14  +010
ret                         1D  +011
```

This function loads the word at ptr, discards its 8 least significant bits (while keeping the others unchanged) by computing the bitwise AND of this word with $\texttt{FFFFFF00}_{16}$, replaces them with $value$ (supposed to be strictly less than 256) with a bitwise OR, and finally stores the result back at ptr. We then implement Algorithm 13.1 in the following `mem_copy` function:

`mem_copy`(src, dst, n) $\rightarrow$ $dst + n$	**fn**	03		**19**	C0A16
Step 1. If $dst \geq src$, go the second half of this function (see below).	**get**	01	dst	**16**	+002
	get	00	src	**16**	+004
	ifge	0042		**11**	+006
Step 2. Initialize i to 0.	**cst_0**		$\rightarrow i$	**00**	+009
Step 3. If $i + 4 > n$, go to step 4 (i is stored in the 7^{th} stack frame slot).	**get**	07	i	**16**	+00A
	cst8	04		**02**	+00C
	add			**04**	+00E
	get	02	n	**16**	+00F
	ifgt	0026		**0E**	+011
Otherwise, load the word at $src + i$ and store it at $dst + i$, ...	**get**	01	dst	**16**	+014
	get	07	i	**16**	+016
	add			**04**	+018
	get	00	src	**16**	+019
	get	07	i	**16**	+01B
	add			**04**	+01D
	load			**13**	+01E
	store			**14**	+01F
... increment i by 4, and go back above to check again if $i + 4 < n$.	**cst8**	04		**02**	+020
	add			**04**	+022
	goto	000A		**12**	+023
Step 4. If $i \geq n$, go the end of the function (see below).	**get**	07	i	**16**	+026
	get	02	n	**16**	+028
	ifge	0078		**11**	+02A
Otherwise, load the byte at $src + i$ and store it at $dst + i$, ...	**get**	01	dst	**16**	+02D
	get	07	i	**16**	+02F
	add			**04**	+031
	get	00	src	**16**	+032
	get	07	i	**16**	+034
	add			**04**	+036
	call	03C0	`load_byte`	**1A**	+037
	call	0A04	`store_byte`	**1A**	+03A
... increment i by 1, and go back above to check again if $i < n$.	**cst_1**			**01**	+03D
	add			**04**	+03E
	goto	0026		**12**	+03F

The second half of the function is similar, and handles the case $dst \geq src$ by copying data in decreasing order, as described in Algorithm 13.1:

```
get   02  n          16  +042
get   07  i          16  +044
cst8  04             02  +046
iflt  005D           0C  +048
cst8  04             02  +04B
sub                  05  +04D
get   01  dst        16  +04E
get   07  i          16  +050
add                  04  +052
get   00  src        16  +053
get   07  i          16  +055
add                  04  +057
load                 13  +058
store                14  +059
goto  0044           12  +05A
```

```
get   07  i          16  +05D
cst_0                00  +05F
ifle  0078           0F  +060
cst_1                01  +063
sub                  05  +064
get   01  dst        16  +065
get   07  i          16  +067
add                  04  +069
get   00  src        16  +06A
get   07  i          16  +06C
add                  04  +06E
call  03C0 load_byte 1A  +06F
call  0A04 store_byte 1A +072
goto  005D           12  +075
```

Both parts jump to the following final instructions when the copy is done. These instructions simply return $dst + n$:

```
get   01  dst        16  +078
get   02  n          16  +07A
add                  04  +07C
retv                 1E  +07D
```

Using this memory copy function, it is easy to write a function to copy a data buffer from src to dst. Indeed, we simply need to call `mem_copy` with src, dst, and $n = \text{mem32}[src] + 4$, the total size of the data buffer (recall that $\text{mem32}[x]$ means "the 32 bit value at address x"):

```
buffer_copy(src, dst)
fn    02             19  C0A94
get   00  src        16  +002
get   01  dst        16  +004
get   00  src        16  +006
load                 13  +008
cst8  04             02  +009
add                  04  +00B
call  0A16 mem_copy  1A  +00C
ret                  1D  +00F
```

We can now implement a function to copy and save a single page in flash memory. As described above, to save $n \leq 256$ bytes, we must first copy them, then copy the remaining $256 - n$ bytes of the page in place, and finally save the page by calling the subroutine defined in Section 13.1.2. For this, the subroutine must be stored somewhere in RAM first. The easiest solution is to store it on the stack. The following function uses this method to save n bytes starting from src in a page of the Flash1 memory bank specified by its $page$ index:

`page_flash`(src, $page$, n)

If $n = 0$ there is nothing to do, return right away. Otherwise execute the

```
fn    03             19  C0AA4
get   02  n          16  +002
cst_0                00  +004
```

following instructions.

Copy n bytes from src to $C0000_{16} + 256.page$, the address of the $page^{th}$ page of the Flash1 memory bank. The `mem_copy` call returns $dst = C0000_{16} + 256.page + n$, in the 7^{th} stack frame slot.

Copy the remaining $256 - n$ bytes of the page in place, from dst to dst, and discard the result returned by `mem_copy`.

Disable the USART interrupts with the Nested Vector Interrupt Controller (see Section 11.3 and below).

Push the value to store in the EEFC1 Command Register in order to save the $page^{th}$ page: $5A000003_{16} \mid (page \ll 8)$.

Push the remaining words of the subroutine to save this page. These words must be pushed in reverse order, because each word is pushed 4 bytes *before* the previous one.

Call the subroutine, which starts in the 14^{th} stack frame slot. Its *interworking address* is the address of this slot (given by the `ptr` instruction), plus 1.

Re-enable the USART interrupts and return.

```
ifne  0009                10  +005
ret                       1D  +008
get   00   src            16  +009
cst   000C0000            03  +00B
get   01   page           16  +010
cst8  08                  02  +012
lsl                       0A  +014
add                       04  +015
get   02   n              16  +016
callr 00A6 mem_copy       1B  +018
get   07   dst            16  +01B
get   07   dst            16  +01D
cst   00000100            03  +01F
get   02   n              16  +024
sub                       05  +026
callr 00B5 mem_copy       1B  +027
pop                       18  +02A
cst   E000E180            03  +02B
cst   00020000            03  +030
store                     14  +035
cst   5A000003            03  +036
get   01   page           16  +03B
cst8  08                  02  +03D
lsl                       0A  +03F
or                        09  +040
cst   400E0C04            03  +041
cst   0000BD03            03  +046
cst   E7FBBF18            03  +04B
cst   29016841            03  +050
cst   60014904            03  +055
cst   4804B503            03  +05A
ptr   0E                  15  +05F
cst_1                     01  +061
add                       04  +062
blx                       1F  +063
cst   E000E100            03  +064
cst   00020000            03  +069
store                     14  +06E
ret                       1D  +06F
```

A few things should be noted:

- USART interrupts are temporarily disabled while the page is being saved. Without this, a key press or release during this time would run the `keyboard_handler`,

which would make use of the flash memory. In turn, this would cause a Hard Fault because flash memory must not be used while a page is being saved. Unfortunately, flashing a page takes a few milliseconds, during which several interrupts could occur. In this case they are lost, except the last one, which can confuse the keyboard driver. For instance, releasing the "r" key causes two interrupts, for the $F0_{16}$ and $2D_{16}$ scancodes. If the first is lost, this appears as a key press (see Appendix C). This problem disappears in the next part.

- we use `callr` instead of `call` instructions to call `mem_copy`. The next section explains why (these instructions use an offset from their own address, instead of an offset from $C0000_{16}$ – see Section 8.2.4).
- n must be a multiple of 4, so that `mem_copy` does not call `store_byte`. Indeed, `store_byte` does not work in flash memory, because loads do not "see" the effect of stores until the page is saved (see Section 6.5.1). If it was called several times to store the bytes of a word, only the last call would have any effect.

We can finally implement the last function of our flash memory driver, which copies a data buffer starting at src and saves it in the Flash1 memory bank, starting at the $page^{th}$ page. This function simply calls `page_flash` for each page.

`buffer_flash`(src, $page$)

Compute the number of bytes n to copy. This is $\text{mem32}[src] + 4$, rounded upwards to a multiple of 4 (as required by `page_flash`), *i.e.*, $(\text{mem32}[src] + 7) \wedge \text{FFFFFFFC}_{16}$.

If n, in the 6^{th} stack frame slot, is greater than 255, jump to the next instructions. Otherwise, call `page_flash` to copy and save n bytes from src into $page$, and return.

Call `page_flash` to copy and save 256 bytes from src into $page$.

Increment src by 256.

Increment $page$ by 1.

```
fn    02                    19  C0B14
get   00   src              16  +002
load                        13  +004
cst8  07                    02  +005
add                         04  +007
cst   FFFFFFFC              03  +008
and        → n              08  +00D
get   06   n                16  +00E
cst8  FF                    02  +010
ifgt  001F                  0E  +012
get   00   src              16  +015
get   01   page             16  +017
get   06   n                16  +019
callr 008B page_flash       1B  +01B
ret                         1D  +01E
get   00   src              16  +01F
get   01   page             16  +021
cst   00000100              03  +023
callr 0098 page_flash       1B  +028
get   00   src              16  +02B
cst   00000100              03  +02D
add                         04  +032
set   00   src              17  +033
get   01   page             16  +035
cst_1                       01  +037
```

Function	Address
buffer_copy(*src*, *dst*)	C0A94 (C0000$_{16}$+2708)
buffer_flash(*src*, *page*)	C0B14 (C0000$_{16}$+2836)
mem_copy(*src*, *dst*, *n*) → *dst* + *n*	C0A16 (C0000$_{16}$+2582)
page_flash(*src*, *page*, *n*)	C0AA4 (C0000$_{16}$+2724)
store_byte(*ptr*, *value*)	C0A04 (C0000$_{16}$+2564)

TABLE 13.1 The most important functions of the flash memory driver.

Decrement n by 256 and go back above to copy the rest of the data buffer.

```
add                          04   +038
set   01   page              17   +039
cst   00000100               03   +03B
sub                          05   +040
goto  000E                   12   +041
```

In summary, the main functions of our flash memory driver are those listed in Table 13.1, and its full code is the following:

```
00160116 03191D14 09011608 FFFFFF00 03130016 00160219  C0A04
04071600 16040716 01160026 0E021604 04020716 00004211  C0A1C
07160016 04071601 16007811 02160716 000A1204 04021413  C0A34
05040200 5D0C0402 07160216 00261204 010A041A 03C01A04  C0A4C
16050100 780F0007 16004412 14130407 16001604 07160116  C0A64
1E040216 0116005D 120A041A 03C01A04 07160016 04071601  C0A7C
00091000 02160319 1D0A161A 04040213 00160116 00160219  C0A94
03071607 1600A61B 0216040A 08020116 000C0000 0300161D  C0AAC
03031400 02000003 E000E180 03180B5 1B050216 00000100  C0AC4
E7FBBF18 03000BD 03034000E 0C040309 0A080201 165A0000  C0ADC
00E10003 1F04010E 154804B5 03036001 49040329 01684103  C0AF4
061608FF FFFFFC03 04070213 00160219 1D140002 000003E0  C0B0C
00000100 03011600 161D008B 1B061601 16001600 1F0EFF02  C0B24
00000100 03011704 01011600 17040000 01000300 1600981B  C0B3C
                                     000E1205          C0B54
```

13.3 Storage

Lets store our driver in flash memory. For this we must first enter it in RAM, say at address 20070000$_{16}$, and then save it by calling the `buffer_flash` function. In the memory editor, type "w20070000"+Enter, and then store the size of our driver at this address by typing "00000154"+Enter. Continue by entering each word of the driver code, listed above, by typing its value followed by Enter.

Our driver is now in RAM. To save it in flash memory we must call `buffer_flash`, at address $C0B14_{16} - C0A00_{16} + 20070000_{16}$, with $src = 20070000_{16}$ and $page = 10$. This can be done with the following function:

```
save_driver()
fn      00              19  00000
cst     20070000        03  +002
cst8    0A              02  +007
cst     20070114        03  +009
calld                   1C  +00E
ret                     1D  +00F
```

Note that the call to `buffer_flash` causes indirect calls to `page_flash` and `mem_copy` which, for now, are in RAM. Hence, the instructions calling these functions cannot use their final address in flash memory, since they are not stored there yet! This is why we used `callr` instructions instead of `call` instructions in the above code. Indeed, by specifying the callee with an offset from the caller, the code works wherever it is stored, in RAM or in flash memory. Such code is called *position independent code*. The full code of the above function is the following:

```
1D1C2007 01140300A 02200700 00030019  00000
```

With the memory editor, enter these values in an unused RAM region, for instance starting at address 20080000_{16}. Then type "w20080000"+Enter, followed by "r", to run this function. The driver should now be saved in flash memory. To check this, type "w000c0a00"+Enter. You should see the following screen, displaying the same words as those listed above, after the data buffer header:

```
FFFFFF00 03130016 00160219 00000154 000C0A00
00004211 00160116 03191D14 09011608 000C0A10
...
```

Alternatively, if something went wrong or if you don't want to enter all the driver code with the keyboard, you can "cheat" by saving it via an external computer, as follows. First run the `boot_mode_select_rom` function by typing "w000c02b4"+Enter, followed by "r". Then reset the Arduino and, on the host computer, run the following commands to flash the driver code and reset the Arduino again:

```
user@host:~$ python3 flash_helper.py < part3/flash_memory_driver.txt
>Reading page 1034... Done.
Reading page 1035... Done.
Writing page 1034... Done.
Writing page 1035... Done.
>Done.
```

Finally, on the Arduino, type "w000c0a00"+Enter to check that the driver is indeed in flash memory: you should see the same screen as above.

CHAPTER

14 Text Editor

In order to write source code, including the source code of our toy compiler, we need a text editor. Since we don't have a compiler yet, we need to write this editor directly in binary form. It should thus be as short as possible, and therefore as simple as possible. An editor for very short texts, a few lines, could be extremely simple. However, it would be completely impractical to use for longer texts. In theory, we could start with a tiny editor E_0 written in binary form, write with E_0 a small compiler C_1 and a larger editor E_1 (in source code compiled with C_1), write with E_1 a larger compiler C_2 and a better editor E_2, etc. In this chapter, to save space, we directly implement an editor capable of editing "large" texts (tens of thousands of characters). As a result, its size is not minimal, but is still manageable.

14.1 User interface

For very short texts, an editor could simply put any character typed at the end of the current text. But for longer texts, one needs to insert or delete characters at any position. For this we need a *cursor*, indicating where the next character typed will be inserted, or which character will be deleted. We also need a way to move this cursor, one character, one line, or even several lines at a time (it would be impractical to type 10,000 times the same key to move the cursor by 10,000 characters).

A text with more than 30 lines cannot fit on the screen of our toy computer. We thus need a way to *scroll* the text, *i.e.*, to select which part should be displayed at a given time. A simple method is to use an automatic scroll, ensuring that the cursor stays in the middle of the screen. This avoids the need of some "scroll keys", and of the related code. Based on this, we define the following requirements for our text editor:

- the screen should show 15 lines before the cursor line or, if there are less than 15 lines before, all these lines. It should also show all the lines after the cursor, until the end of the text or the bottom of the screen.
- pressing the Arrow Left (resp. Right) key should move the cursor to the previous (resp. next) character.
- pressing the Arrow Up (resp. Down) key should move the cursor to the previous (resp. next) line.

- pressing the Page Up (resp. Down) key should move the cursor to the 30^{th} previous (resp. next) line.
- pressing the BackSpace key should delete the character before the cursor.
- pressing a character key should insert this character at the cursor.
- pressing the Escape key should exit the editor.

14.2 Algorithms

This section explains the methods used in our text editor in order meet these requirements, for texts made of tens of thousands of characters. It explains how the text is stored and edited in memory, and how it is displayed and updated on the screen.

14.2.1 Gap buffer

The simplest way to store the edited text in memory is to store all its characters one after the other, using one byte per character. Then we only need 3 addresses to know where the text begins, where it ends, and where the cursor is. We call them $begin$, end, and $cursor$, respectively (see Figure 14.1). With this method, moving the cursor is very easy: we just need to change the $cursor$ value. However, inserting a character is more complex. Indeed, we need to copy each character after the cursor in the next memory byte, in order to make space for the new character. Similarly, in order to delete a character, we need to copy each character after the cursor in the previous memory byte (see Figure 14.1). Unfortunately, doing this with tens of thousands of characters after the cursor would be too slow (recall that our virtual machine is ten times slower than the microprocessor).

To solve this problem, a solution is to use a slightly more complex data structure to store the edited text, called a *gap buffer*. This structure adds some free memory after the cursor. It uses an additional gap variable, indicating how many free bytes are after the cursor (see Figure 14.2). With this structure, inserting a character is easy and fast: just store it at the cursor, increment $cursor$ by 1 and decrement gap by 1[1]. Deleting a character is also easy and fast: decrement $cursor$ by 1 and increment gap by 1. On the other hand, moving the cursor is now more complex. In order to move it by n characters to the left, one must copy the n characters before the gap, to move them after the gap (and vice-versa to move the cursor to the right – see Figure 14.3). Still, if n represents at most 30 lines of text, this operation is fast enough, even if the text has thousands of lines. We therefore use a gap buffer for our text editor.

Moving the cursor to a new address $cursor'$ in a gap buffer can be done as described in Algorithm 14.1 (as can be seen from Figure 14.3). In order to move the cursor to the l^{th} previous line, this new address can be computed as described in Algorithm 14.2. The basic idea is to scan the text from right to left, starting from the cursor, and to count the number of "new line" characters encountered until l such characters are found (or the beginning of the text is reached). Similarly, to move the

[1] If the gap is empty, the easiest solution is to simply drop any newly typed character.

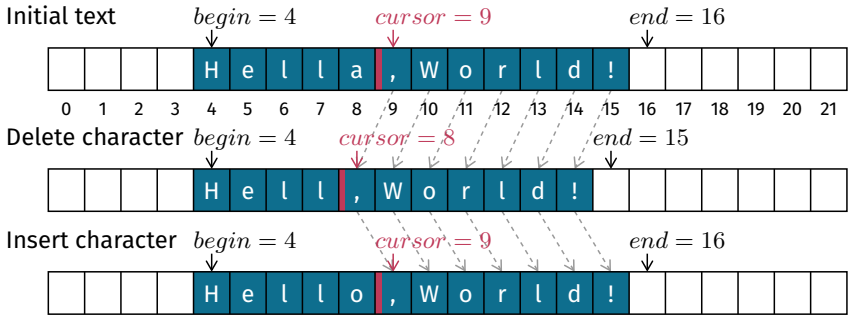


FIGURE 14.1 Inserting or deleting a character in a text stored in a single span requires copying (dashed arrows) all the characters after the cursor to move them one byte after or before their current position (we show characters for clarity, but cells actually contain ASCII code numbers). On the other hand, moving the cursor is trivial.

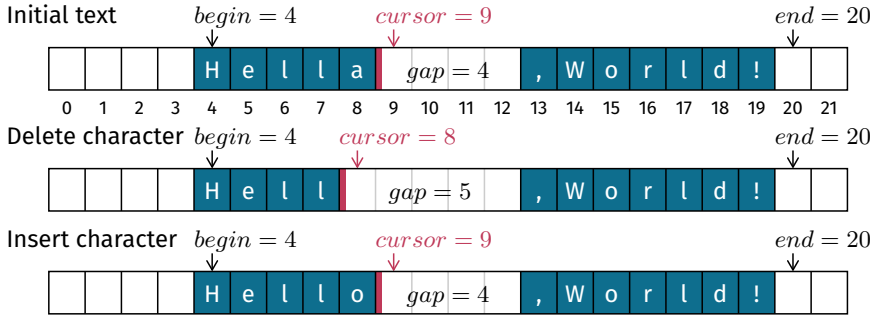


FIGURE 14.2 Inserting or deleting a character in a text stored in a gap buffer is very easy and fast, whatever the text length. No memory copy is needed.

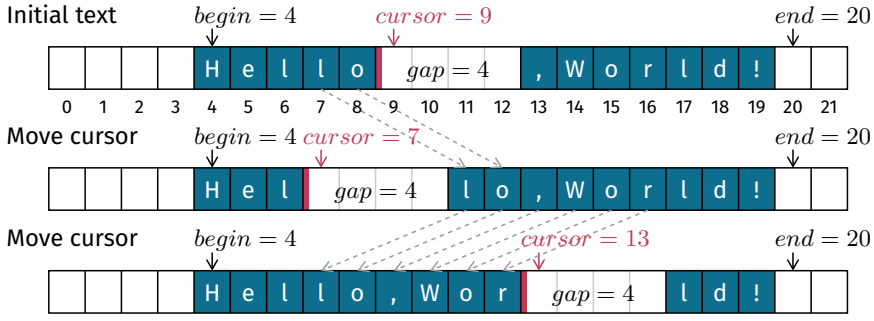


FIGURE 14.3 Moving the cursor by n characters in a gap buffer requires copying n bytes (dashed arrows) to move them from one side of the gap to the other. Still, if n is bounded by a "reasonable" value, this is fast enough even for long texts.

ALGORITHM 14.1 Setting the cursor to $cursor'$, where $begin \leq cursor' \leq end - gap$.

if $cursor' > cursor$
  copy $n = cursor' - cursor$ bytes from $src = cursor + gap$ to $dst = cursor$
otherwise
  copy $n = cursor - cursor'$ bytes from $src = cursor'$ to $dst = cursor' + gap$
set $cursor$ to $cursor'$

ALGORITHM 14.2 Computing the beginning address a of the l^{th} previous line, $l > 0$.

1. initialize a to $cursor$
2. if $a = begin$ return a
3. if the character at $a - 1$ is a "new line" character
4. if $l = 0$ return a, otherwise decrement l by 1
5. decrement a by 1 and go back to step 2

cursor to the l^{th} next line, the new address $cursor'$ can be computed as described in Algorithm 14.3. Note that two addresses can be defined for a character: its current address, depending on the position and size of the gap, and a "canonical address" defined as its current address if there was no gap. The two addresses are equal for characters before the cursor. For those after the cursor, the current address is equal to the canonical address plus gap. Algorithms 14.2 and 14.3 compute a canonical address a, but use current addresses to read characters.

14.2.2 Text drawing

In order to draw the text we must first compute the canonical address of the first character to draw, on the top left corner, so that we get 15 lines of text before the line containing the cursor (if possible). This could be done with Algorithm 14.2 but, with a slightly improved algorithm, we can compute at the same time the column and row where the cursor should appear on screen. Indeed, the row can be incremented from 0, each time we encounter a "new line" character while scanning the text backwards from the cursor. And the column can be incremented from 0 for each character found on the line containing the cursor (the 0^{th} row). Finally, by taking tabulation characters into account (we represent them with 2 spaces), we get Algorithm 14.4.

ALGORITHM 14.3 Computing the beginning address a of the l^{th} next line, $l > 0$.

1. initialize a to $cursor$
2. if $a = end - gap$ return a
3. if the character at $a + gap$ is a "new line" character
4. if $l = 1$ return $a + 1$, otherwise decrement l by 1
5. increment a by 1 and go back to step 2

ALGORITHM 14.4 Computing the beginning address a of the l^{th} previous line, $l > 0$, and the cursor's screen column and row, col and row.

1. initialize a to $cursor$, col to 0 and row to 0
2. if $a = begin$ return $\{a, col, row\}$
3. if the character c at $a - 1$ is a "new line" character
4. if $row = l$ return $\{a, col, row\}$, otherwise increment row by 1
5. otherwise, and if $row = 0$
6. increment col by 1, or by 2 if c is a tabulation character
7. decrement a by 1 and go back to step 2

ALGORITHM 14.5 Drawing the text on screen.

1. clear the screen and set the graphics cursor to the top-left corner, $(0, 0)$
2. initialize r to 0
3. compute $\{a, col, row\}$ with Algorithm 14.4
4. if $a = cursor$, increment a by gap
5. while $a < end$ and $r < 30$
6. if the character c at a is a "new line" character
7. increment r by 1 and set the graphics cursor to $(0, r)$
8. otherwise
9. draw c, or two spaces if c is a tabulation character
10. increment a by 1
11. if $a = cursor$, increment a by gap
12. set the graphics cursor to (col, row)

We can then draw the text one character at a time, until the end of the text or the bottom of the screen is reached (see Algorithm 14.5). Recall that the graphics cursor automatically moves to the right after a character is drawn[2]. However, it does this also for "new line" characters (drawn with some icon), instead of moving the graphics cursor to the beginning of a new line. Thus, "new line" characters must not be drawn. Instead, a "current row" variable r should be incremented, and the graphics cursor should be set to the beginning of this row. Likewise, two spaces must be drawn for each tabulation character (otherwise drawn with some icon). Finally, note that this algorithm requires the current address a of each character, in order to read them. This address must be incremented by gap when it reaches $cursor$.

14.2.3 Double buffering

Most of the time, when a character is typed, only the line containing it needs to be updated on the screen, which could be fast. However, sometimes the whole screen

[2]The graphics card also automatically wraps lines longer than 100 characters. To simplify our text editor we assume that lines are always shorter than this.

FIGURE 14.4 To avoid flickering, text is drawn in one layer while the other is displayed (left). When this is done the layer roles are exchanged (right).

must be redrawn (*e.g.*, when scrolling up or down). To simplify our editor we always redraw the whole screen after each key press. This takes some time, during which partially redrawn images could be seen, causing *flickering*. To avoid this, a solution is to draw the text in a video memory region which is not visible on screen and, when this is done, display this image. The graphics card we are using provides an easy way to use this method, called *double buffering*. It can divide its 768 KB of RAM in two images, called layers 0 and 1. By default text is drawn in layer 0, which is also displayed on the LCD. But it is possible to draw in layer 0 while layer 1 is displayed, or vice-versa (see Figure 14.4). This can be done with the following graphics card registers (we show only the bits that we use):

R20$_{16}$	Display Configuration	l 0 0 0 0 0 0 0
R41$_{16}$	Memory Write Control 1	0 0 0 0 0 0 0 l
R52$_{16}$	Layer Transparency 0	0 0 0 0 0 0 0 l

The l bit of the Display Configuration register enables the double buffering mode when it is 1, and disables it otherwise. When it is enabled, characters are drawn in the layer given by the l bit of the Memory Write Control 1 register, while the screen displays the layer given by the l bit of the Layer Transparency 0 register. When double buffering is disabled, characters are drawn in layer 0 and the screen displays layer 0 too. The layer used for drawing is called the *back buffer*, while the layer displayed on the screen is called the *front buffer*.

14.3 Implementation

We can now implement our text editor, in a new data buffer starting at the next page after the flash memory driver, at address C0C0$_{16}$. We start with a function implementing Algorithm 14.1, with the last step replaced with "return $cursor'$":

`ted_set_cursor`($begin$, $cursor$, gap, $cursor'$) $\rightarrow$ $cursor'$

fn	04		**19**	C0C04	**get**	01	$cursor$	**16**	+009
get	03	$cursor'$	**16**	+002	**get**	02	gap	**16**	+00B
get	01	$cursor$	**16**	+004	**add**			**04**	+00D
ifle	001B		**0F**	+006	**get**	01	$cursor$	**16**	+00E

```
get  03  cursor'      16  +010
get  01  cursor       16  +012
sub                   05  +014
call 0A16 mem_copy    1A  +015
get  03  cursor'      16  +018
retv                  1E  +01A
get  03  cursor'      16  +01B
get  03  cursor'      16  +01D
get  02  gap          16  +01F
add                   04  +021
get  01  cursor       16  +022
get  03  cursor'      16  +024
sub                   05  +026
call 0A16 mem_copy    1A  +027
get  03  cursor'      16  +02A
retv                  1E  +02C
```

Then, instead of implementing Algorithm 14.2, we implement Algorithm 14.4, which is more general. This algorithm is supposed to return 3 values a, col and row. Since a function can't return several values, the following function only returns a. To "return" col and row we use two *pointers* col^p and row^p where these values can be stored in memory. We suppose that these values are initialized to 0 by the caller:

`ted_move_backward`($begin$, $cursor$, l, col^p, row^p) $\rightarrow a$

Step 1. Initialize c to 0 and a to $cursor$.

Step 2. If a (in the 10^{th} stack frame slot) is equal to $begin$, go to step 8.

Step 3. Set c (in the 9^{th} stack frame slot) to the character at $a - 1$.

If c is not equal to Enter (A_{16}), go to step 5.

Step 4. If the value at row^p is equal to l, go to step 8.

Otherwise, increment the value at row^p by 1, and go to step 7.

Step 5. If the value at row^p is not 0, go to step 7.

```
fn    05                  19  C0C31
cst_0       → c           00  +002
get   01    cursor → a    16  +003
get   0A    a             16  +005
get   00    begin         16  +007
ifle  004C                0F  +009
get   0A    a             16  +00C
cst_1                     01  +00E
sub                       05  +00F
call  03C0 load_byte      1A  +010
set   09    c             17  +013
get   09    c             16  +015
cst8  0A                  02  +017
ifne  002F                10  +019
get   04    row^p         16  +01C
load                      13  +01E
get   02    l             16  +01F
ifeq  004C                0D  +021
get   04    row^p         16  +024
get   04    row^p         16  +026
load                      13  +028
cst_1                     01  +029
add                       04  +02A
store                     14  +02B
goto  0047                12  +02C
get   04    row^p         16  +02F
load                      13  +031
cst_0                     00  +032
```

	ifne	0047		**10**	+033
Step 6. Add 1 to the value at col^p, one	**get**	03	col^p	**16**	+036
or two times, depending on whether c is	**get**	03	col^p	**16**	+038
equal to Tab (9_{16}) or not.	**load**			**13**	+03A
	get	09	c	**16**	+03B
	cst8	09		**02**	+03D
	ifne	0044		**10**	+03F
	cst_1			**01**	+042
	add			**04**	+043
	cst_1			**01**	+044
	add			**04**	+045
	store			**14**	+046
Step 7. Decrement a by 1 and go back	**cst_1**			**01**	+047
to step 2.	**sub**			**05**	+048
	goto	0005		**12**	+049
Step 8. Return a.	**get**	0A	a	**16**	+04C
	retv			**1E**	+04E

We continue with the implementation of Algorithm 14.3:

`ted_move_forward`($cursor$, gap, end, l) $\rightarrow a$

	fn	04		**19**	C0C80
Step 1. Initialize a to $cursor$.	**get**	00	$cursor \rightarrow a$	**16**	+002
Step 2. If a (in the 8^{th} stack frame slot)	**get**	08	a	**16**	+004
is equal to $end - gap$ return a.	**get**	02	end	**16**	+006
	get	01	gap	**16**	+008
	sub			**05**	+00A
	ifne	0011		**10**	+00B
	get	08	a	**16**	+00E
	retv			**1E**	+010
Step 3. If the character at $a + gap$ is	**get**	08	a	**16**	+011
not Enter (A_{16}), go to step 5.	**get**	01	gap	**16**	+013
	add			**04**	+015
	call	03C0	`load_byte`	**1A**	+016
	cst8	0A		**02**	+019
	ifne	002F		**10**	+01B
Step 4. If l is equal to 1	**get**	03	l	**16**	+01E
	cst_1			**01**	+020
	ifne	0029		**10**	+021
return $a + 1$,	**get**	08	a	**16**	+024
	cst_1			**01**	+026
	add			**04**	+027
	retv			**1E**	+028
otherwise decrement l by 1.	**get**	03	l	**16**	+029

	cst_1			**01**	+02B
	sub			**05**	+02C
	set	03	l	**17**	+02D
Step 5. Increment a by 1 and go back to step 2.	**cst_1**			**01**	+02F
	add			**04**	+030
	goto	0004		**12**	+031

With this we can compute the new cursor position after an ArrowLeft, Right, Up or Down key, or a PageUp or Down key c has been pressed, as follows:

`ted_handle_key`($begin$, $cursor$, gap, end, c) $\rightarrow$ $cursor'$

	fn	05		**19**	C0CB4
Step 1. Initialize col and row to 0.	**cst_0**		$\rightarrow$ col	**00**	+002
	cst_0		$\rightarrow$ row	**00**	+003
Step 2. If c is not the ArrowLeft key, or if $cursor = begin$ (no way to go left), go to step 3.	**get**	04	c	**16**	+004
	cst8	EB		**02**	+006
	ifne	0017		**10**	+008
	get	01	$cursor$	**16**	+00B
	get	00	$begin$	**16**	+00D
	ifeq	0017		**0D**	+00F
Otherwise, return $cursor - 1$.	**get**	01	$cursor$	**16**	+012
	cst_1			**01**	+014
	sub			**05**	+015
	retv			**1E**	+016
Step 3. If c is not the ArrowRight key, or if $cursor = end - gap$ (no way to go right), go to step 4.	**get**	04	c	**16**	+017
	cst8	F4		**02**	+019
	ifne	002D		**10**	+01B
	get	01	$cursor$	**16**	+01E
	get	03	end	**16**	+020
	get	02	gap	**16**	+022
	sub			**05**	+024
	ifge	002D		**11**	+025
Otherwise, return $cursor + 1$.	**get**	01	$cursor$	**16**	+028
	cst_1			**01**	+02A
	add			**04**	+02B
	retv			**1E**	+02C
Step 4. If c is not the ArrowUp key, go to step 5.	**get**	04	c	**16**	+02D
	cst8	F5		**02**	+02F
	ifne	0041		**10**	+031
Otherwise, return the beginning address of the previous line by calling the `ted_move_backward` function with $l = 1$, and with pointers to col and row, in the 9^{th} and 10^{th} stack frame slots, for col^p and	**get**	00	$begin$	**16**	+034
	get	01	$cursor$	**16**	+036
	cst_1			**01**	+038
	ptr	09	col	**15**	+039
	ptr	0A	row	**15**	+03B

row^p.

Step 5. If c is not the ArrowDown key, go to step 6.

Otherwise, return the beginning address of the next line.

Step 6. If c is not the PageUp key, go to step 7.

Otherwise, return the beginning address of the 30^{th} previous line ($30 = 1E_{16}$).

Step 7. If c is not the PageDown key, go to step 8.

Otherwise, return the beginning address of the 30^{th} next line.

Step 8. Return $cursor$.

```
call   0C31 …move_backward  1A  +03D
retv                        1E  +040
get    04   c               16  +041
cst8   F2                   02  +043
ifne   0053                 10  +045
get    01   cursor          16  +048
get    02   gap             16  +04A
get    03   end             16  +04C
cst_1                       01  +04E
call   0C80 …move_forward   1A  +04F
retv                        1E  +052
get    04   c               16  +053
cst8   FD                   02  +055
ifne   0068                 10  +057
get    00   begin           16  +05A
get    01   cursor          16  +05C
cst8   1E                   02  +05E
ptr    09   col             15  +060
ptr    0A   row             15  +062
call   0C31 …move_backward  1A  +064
retv                        1E  +067
get    04   c               16  +068
cst8   FA                   02  +06A
ifne   007B                 10  +06C
get    01   cursor          16  +06F
get    02   gap             16  +071
get    03   end             16  +073
cst8   1E                   02  +075
call   0C80 …move_forward   1A  +077
retv                        1E  +07A
get    01   cursor          16  +07B
retv                        1E  +07D
```

In order to implement a function to draw the edited text, we first provide 3 simple functions to use double buffering. The first one enables this mode and sets the back buffer to layer 1:

```
gpu_set_double_buffer()
fn     00                   19  C0D32
cst8   20                   02  +002
cst8   80                   02  +004
call   0378 …set_register   1A  +006
cst8   41                   02  +009
cst_1                       01  +00B
call   0378 …set_register   1A  +00C
ret                         1D  +00F
```

The second one disables double buffering and sets the front and back buffers to

layer 0:

```
gpu_set_single_buffer()
fn      00                  19  C0D42
cst8    41                  02  +002
cst_0                       00  +004
call    0378 …set_register  1A  +005
cst8    52                  02  +008
cst_0                       00  +00A
call    0378 …set_register  1A  +00B
cst8    20                  02  +00E
cst_0                       00  +010
call    0378 …set_register  1A  +011
ret                         1D  +014
```

The third one swaps the layers used for the front and back buffers. For this, it reads the l bit of the Memory Write Control ("back buffer") register (in the 4^{th} stack frame slot), sets this register to $1 - l$, and sets the Layer Transparency 0 ("front buffer") register to l. The first step is done by selecting the register with a Select Register command, followed by a Read Data command (both sent with `spi_transfer` – see Section 10.2.3):

```
gpu_swap_buffer()
fn      00                  19  C0D57
cst     00008041            03  +002
call    034E spi_transfer   1A  +007
pop                         18  +00A
cst     00004000            03  +00B
call    034E spi_transfer   1A  +010
cst_1                       01  +013
get     04  layer           16  +014
sub                         05  +016
call    034E spi_transfer   1A  +017
pop                         18  +01A
cst8    52                  02  +01B
get     04  layer           16  +01D
call    0378 …set_register  1A  +01F
ret                         1D  +022
```

We can now implement a function to draw the edited text, with Algorithm 14.5:

`ted_draw`($begin$, $cursor$, gap, end)

Step 1. Clear the screen (actually the back buffer) and set the graphics cursor to $(0, 0)$.

Step 2. Initialize r to 0 and c to 0.

Step 3. Initialize col and row to 0, and compute a, col and row – in the 12^{th}, 10^{th} and 11^{th} stack frame slots, respectively – by calling `ted_move_backward` with $l = 15$ and with pointers to col and row for col^p and row^p.

Step 4. If $a = cursor$, increment a by gap.

```
fn      04                  19  C0D7A
call    0409 …clear_screen  1A  +002
cst_0                       00  +005
cst_0                       00  +006
call    0422 …set_cursor    1A  +007
cst_0       → r             00  +00A
cst_0       → c             00  +00B
cst_0       → col           00  +00C
cst_0       → row           00  +00D
get     00  begin           16  +00E
get     01  cursor          16  +010
cst8    0F                  02  +012
ptr     0A  col             15  +014
ptr     0B  row             15  +016
call    0C31 …move_backward 1A  +018
get     0C  a               16  +01B
get     01  cursor          16  +01D
```

Step 5. If $a \geq end$, go to step 12.

If r (in the 8^{th} stack frame slot) is ≥ 30, go to step 12 (30 = $1E_{16}$).

Step 6. Set c (in the 9^{th} stack frame slot) to the character at a.

If c is not equal to Enter (A_{16}), go to step 9.

Step 7. Increment r by 1,

set the graphics cursor to $(0, r)$, and go to step 10.

Step 9. If c is equal to Tab (9_{16}), draw two spaces (20_{16}) and go to step 10.

Otherwise draw c.

Step 10. Increment a by 1.

Step 11. If a is not equal $cursor$, go back to step 5.

Otherwise increment a by gap and go back to step 5.

Step 12. Switch the back and front buffers, set the graphics cursor to (col, row) and return.

```
ifne  0025              10  +01F
get   02    gap         16  +022
add                     04  +024
get   0C    a           16  +025
get   03    end         16  +027
ifge  0078              11  +029
get   08    r           16  +02C
cst8  1E                02  +02E
ifge  0078              11  +030
get   0C    a           16  +033
call  03C0  load_byte   1A  +035
set   09    c           17  +038
get   09    c           16  +03A
cst8  0A                02  +03C
ifne  0050              10  +03E
get   08    r           16  +041
cst_1                   01  +043
add                     04  +044
set   08    r           17  +045
cst_0                   00  +047
get   08    r           16  +048
call  0422  …set_cursor 1A  +04A
goto  0069              12  +04D
get   09    c           16  +050
cst8  09                02  +052
ifne  0064              10  +054
cst8  20                02  +057
call  0465  gpu_draw_char 1A +059
cst8  20                02  +05C
call  0465  gpu_draw_char 1A +05E
goto  0069              12  +061
get   09    c           16  +064
call  0465  gpu_draw_char 1A +066
cst_1                   01  +069
add                     04  +06A
get   0C    a           16  +06B
get   01    cursor      16  +06D
ifne  0025              10  +06F
get   02    gap         16  +072
add                     04  +074
goto  0025              12  +075
call  0D57  …swap_buffer 1A +078
get   0A    col         16  +07B
get   0B    row         16  +07D
```

```
call   0422 …set_cursor   1A   +07F
ret                       1D   +082
```

We can finally implement the main function of our text editor. The following function takes as parameter the address of a *text buffer*, *i.e.*, a data buffer containing the text to edit (see Section 13.1.3). It also takes as parameter an initial cursor *offset* from the beginning of the text, and a maximum text length. It starts by storing in the 7^{th} stack frame slot the current text *length* (contained the text buffer header), and returns immediately if it is larger than *max_length*:

`text_editor`(*buffer*, *offset*, *max_length*)

```
fn     03                 19   C0DFD
get    00   buffer         16   +002
load        → length      13   +004
get    07   length         16   +005

get    02   max_length     16   +007
ifle   000D               0F   +009
ret                       1D   +00C
```

It continues by initializing *begin* to *buffer* + 4, *cursor* to *begin* + *length*, *end* to *begin* + *max_length*, *gap* to *end* − *cursor*, and *c* to 0, in the 8^{th}, 9^{th}, 10^{th}, 11^{th}, and 12^{th} stack frame slots, respectively:

```
get    00   buffer         16   +00D
cst8   04                 02   +00F
add         → begin       04   +011
get    08   begin          16   +012
get    07   length         16   +014
add         → cursor      04   +016
get    08   begin          16   +017

get    02   max_length     16   +019
add         → end         04   +01B
get    0A   end            16   +01C
get    09   cursor         16   +01E
sub         → gap         05   +020
cst_0       → c           00   +021
```

The cursor is then changed to *begin* + *offset* with a call to `ted_set_cursor`, after setting *offset* to *length* if it is larger than that. The initialization phase ends by enabling the double buffering mode, setting the color to green $(0, 7, 0)$, and drawing the text:

```
get    01   offset         16   +022
get    07   length         16   +024
ifle   002D               0F   +026
get    07   length         16   +029
set    01   offset         17   +02B
get    08   begin          16   +02D
get    09   cursor         16   +02F
get    0B   gap            16   +031
get    08   begin          16   +033
get    01   offset         16   +035

add                       04   +037
call   0C04 …set_cursor   1A   +038
set    09   cursor         17   +03B
call   0D32 …double_buffer 1A  +03D
cst_0                     00   +040
cst8   07                 02   +041
cst_0                     00   +043
call   044D gpu_set_color 1A   +044
get    08   begin          16   +047
get    09   cursor         16   +049
```

```
get   0B  gap        16  +04B
get   0A  end        16  +04D
call  0D7A ted_draw  1A  +04F
```

The rest of the function is a loop which handles keys typed on the keyboard, until Escape is pressed.

Step 1. Read a character from the keyboard and store it in c.

Step 2. If c is not the Escape key, go to step 3.

Otherwise, set the cursor to the end of the text, *i.e.*, to $end - gap$, to remove any gap in the text itself. Then store the final text length in the buffer header, *i.e.*, store this new cursor value minus $begin$ at address $buffer$.

Then disable the double buffering mode and return.

Step 3. If c is not the BackSpace key, go to Step 4.

Otherwise, if $cursor = begin$ (no previous character to delete), go back to step 1.

Otherwise, delete the previous character by decrementing $cursor$ by 1,

and by incrementing gap by 1. Then go to step 6 to redraw the text.

Step 4. If c is not an ASCII character other than Delete ($7F_{16}$), go to step 5.

Otherwise, if $gap = 0$ (no way to insert a new character), go back to step 1.

```
call  0606 …wait_char      1A  +052
set   0C   c               17  +055
get   0C   c               16  +057
cst8  1B                   02  +059
ifne  0076                 10  +05B
get   00   buffer          16  +05E
get   08   begin           16  +060
get   09   cursor          16  +062
get   0B   gap             16  +064
get   0A   end             16  +066
get   0B   gap             16  +068
sub                        05  +06A
call  0C04 …set_cursor     1A  +06B
get   08   begin           16  +06E
sub                        05  +070
store                      14  +071
call  0D42 …single_buffer  1A  +072
ret                        1D  +075
get   0C   c               16  +076
cst8  08                   02  +078
ifne  0093                 10  +07A
get   09   cursor          16  +07D
get   08   begin           16  +07F
ifeq  0052                 0D  +081
get   09   cursor          16  +084
cst_1                      01  +086
sub                        05  +087
set   09   cursor          17  +088
get   0B   gap             16  +08A
cst_1                      01  +08C
add                        04  +08D
set   0B   gap             17  +08E
goto  00CE                 12  +090
get   0C   c               16  +093
cst8  7F                   02  +095
ifge  00B6                 11  +097
get   0B   gap             16  +09A
cst_0                      00  +09C
```

Otherwise, store c at $cursor$,

increment $cursor$ by 1,

and decrement gap by 1. Then go to step 6 to redraw the text.

Step 5. If c is any character not handled above, set the cursor to the new cursor value computed by `ted_handle_key`.

Step 6. Redraw the edited text and go back to step 1.

```
ifeq  0052                0D  +09D
get   09  cursor          16  +0A0
get   0C  c               16  +0A2
call  0A04 store_byte     1A  +0A4
get   09  cursor          16  +0A7
cst_1                     01  +0A9
add                       04  +0AA
set   09  cursor          17  +0AB
get   0B  gap             16  +0AD
cst_1                     01  +0AF
sub                       05  +0B0
set   0B  gap             17  +0B1
goto  00CE                12  +0B3
get   08  begin           16  +0B6
get   09  cursor          16  +0B8
get   0B  gap             16  +0BA
get   08  begin           16  +0BC
get   09  cursor          16  +0BE
get   0B  gap             16  +0C0
get   0A  end             16  +0C2
get   0C  c               16  +0C4
call  0CB4 …handle_key    1A  +0C6
call  0C04 …set_cursor    1A  +0C9
set   09  cursor          17  +0CC
get   08  begin           16  +0CE
get   09  cursor          16  +0D0
get   0B  gap             16  +0D2
get   0A  end             16  +0D4
call  0D7A ted_draw       1A  +0D6
goto  0052                12  +0D9
```

In summary, the main functions of our text editor are those listed in Table 14.1, and its full code is the following:

```
0A161A05 01160316 01160402 16011600 1B0F0116 03160419  C0C04
0005191E 03160A16 1A050316 01160402 16031603 161E0316  C0C1C
2F100A02 09160917 03C01A05 010A1600 4C0F0016 0A160116  C0C34
00130416 00471214 04011304 16041600 4C0D0216 13041600  C0C4C
05120501 14040104 01004410 09020916 13031603 16004710  C0C64
1608161E 08160011 10050116 02160816 00160419 1E0A1600  C0C7C
0103161E 04010816 00291001 0316002F 100A0203 C01A0401  C0C94
0D001601 16001710 EB020416 00000519 00041204 01031705  C0CAC
002D1105 02160316 0116002D 10F40204 161E0501 01160017  C0CC4
0C311A0A 15091501 01160016 004110F5 0204161E 04010116  C0CDC
```

Function	Address
ted_draw(*begin*, *cursor*, *gap*, *end*)	C0D7A ($C0000_{16}$+3450)
text_editor(*buffer*, *offset*, *max_length*)	C0DFD ($C0000_{16}$+3581)

TABLE 14.1 The most important functions of the text editor.

```
10FD0204 161E0C80 1A010316 02160116 005310F2 0204161E  C0CF4
16007B10 FA020416 1E0C311A 0A150915 1E020116 00160068  C0D0C
0203781A 80022002 00191E01 161E0C80 1A1E0203 16021601  C0D24
1A002002 03781A00 52020378 1A004102 00191D03 781A0141  C0D3C
1601034E 1A000040 00031803 4E1A0000 80410300 191D0378  C0D54
04221A00 00040091A 04191D03 781A0416 52021803 4E1A0504  C0D6C
00251001 160C160C 311A0B15 0A150F02 01160016 00000000  C0D84
091703C0 1A0C1600 78111E02 08160078 1103160C 16040216  C0D9C
09160069 1204221A 08160008 17040108 16005010 0A020916  C0DB4
0104651A 09160069 1204651A 20020465 1A200200 64100902  C0DCC
04221A0B 160A160D 571A0025 12040216 00251001 160C1604  C0DE4
04071608 16040402 00161D00 0D0F0216 07161300 1603191D  C0DFC
08160117 0716002D 0F071601 16000509 160A1604 02160816  C0E14
044D1A00 07020000 321A0917 0C041A04 01160816 0B160916  C0E2C
16007610 1B020C16 0C170606 1A0D7A1A 0A160B16 09160816  C0E44
161D0D42 1A140508 160C041A 050B160A 160B1609 16081600  C0E5C
1704010B 16091705 01091600 520D0816 09160093 10080200  C0E74
0A041A0C 16091600 520D000B 1600B611 7F020C16 00CE120B  C0E8C
1608160B 16091608 1600CE12 0B170501 0B160917 04010916  C0EA4
1A0A160B 16091608 1609170C 041A0CB4 1A0C160A 160B1609  C0EBC
                                    00 52120D7A        C0ED4
```

To store it in flash memory we must enter it in RAM first, lets say at address 20070000_{16}, and then save it in flash. In the memory editor, type "w20070000"+Enter, and then store the text editor size in bytes at this address by typing "w000002d5"+Enter. Continue by entering each word of the text editor code, listed above, by typing its value followed by Enter. Finally, save this code in flash memory (starting at $page = 12$) by running the following function:

```
save()
fn    00              19  00000
cst   20070000        03  +002
cst8  0C              02  +007
call  0B14 buffer_flash 1A  +009
ret                   1D  +00C
```

For this, type "w20080000"+Enter, followed by the full code of this function:

```
1D 0B141A0C 02200700 00030019  00000
```

Then type "w20080000" followed by "r" to run it. Alternatively, if you don't want to enter the full text editor code manually with the memory editor, which is a bit

tedious, you can "cheat" by saving it via an external computer, as follows. First run the `boot_mode_select_rom` function by typing "w000c02b4"+Enter, followed by "r". Then reset the Arduino and, on the host computer, run the following commands to flash the text editor code and reset the Arduino again:

```
user@host:~$ python3 flash_helper.py < part3/text_editor.txt
>Reading page 1036... Done.
Reading page 1037... Done.
Reading page 1038... Done.
Writing page 1036... Done.
Writing page 1037... Done.
Writing page 1038... Done.
>Done.
```

14.4 Experiments

Lets test our text editor. The following function edits a text buffer stored at address 20070000_{16}, with an initial cursor at the beginning of the text, and a maximum text length of 1000_{16} bytes:

```
test()
fn      00                19 00000
cst     20070000          03 +002
cst_0                     00 +007
cst     00001000          03 +008
call    0DFD text_editor  1A +00D
ret                       1D +010
```

Enter it in RAM at address 20080000_{16} by typing "w20080000"+Enter, followed by the full code of this function:

```
1D 0DFD1A00 00100003 00200700 00030019  00000
```

Then initialize an empty text buffer by typing "w20070000"+Enter, followed by "00000000"+Enter. Finally, run the text editor on this empty buffer by typing "20080000"+Enter, followed by "r". If all goes well you should be able to type some text. Try typing some characters, test the Tab and Enter keys, type several lines and then test the arrow keys, and the Page Up and Down keys. You can also exit the text editor with Escape, and then run it again by typing "r" (you should see the text you typed before, with the cursor reinitialized to the beginning of the text). When you are back in the memory editor, you should also see the length of your text, at address 20070000_{16}, followed by the ASCII codes of each character.

If something goes wrong, this is probably due to a typo when you entered the text editor code. In this case, with the memory editor, double check the code in flash memory by comparing it with the code shown at then end of the previous section. If you find an error, copy this code in RAM (with a small function using `buffer_copy`), fix the error with the memory editor, and save the code back in flash memory (with the `save` function from the previous section).

CHAPTER

15 Opcodes Compiler

We now have everything we need to implement our compiler. We start in this chapter with a very simple version, whose main role is to convert opcode names into their numerical value. Indeed, this initial compiler must be written in binary form, and should thus be as small as possible, in order to simplify our task. Hopefully, this is the last program we need to write in such form (besides a few small functions to launch programs with the memory editor). We use it at the end to write a command editor, namely a small program to make it easier to run other programs.

15.1 Requirements

The goal of our initial compiler is to convert opcode instructions from textual to binary form. For instance, given the text "`fn 1 get 0 cst_0 ifne 10` ...", it should produce, in increasing address order, 19_{16} 01_{16} 16_{16} 00_{16} 00_{16} 10_{16} $000A_{16}$ ... The programs that it should accept as input can be described as "zero or more instructions, one after the other", where each instruction is one of "`cst_0`", "`cst_1`", "`cst8`" followed by an 8-bit value, "`cst`" followed by a 32-bit value, and so on for the remaining opcodes. These rules define the *grammar* of a *programming language*, that valid programs must follow. They can be summarized with:

program: instruction*
instruction: "`cst_0`" | "`cst_1`" | "`cst8`" INTEGER | "`cst32`" INTEGER | ...

where "*" means "zero or more times" and "|" means "or". Text between quotes, as well as names in capital letters, refer to individual "words" or "punctuation signs" of the program, called *tokens*. As in English, tokens are generally separated by spaces. Here INTEGER designates an integer value, *i.e.*, a token made of one or more decimal digit characters. In this context, we define the precise requirements of our initial compiler as follows:

- The compiler should take as input a source code address, noted src_buffer, and a destination address where to store the compiled code, noted dst_buffer.
- The source code should be in a data buffer (see Section 13.1.3), and should follow the above grammar.

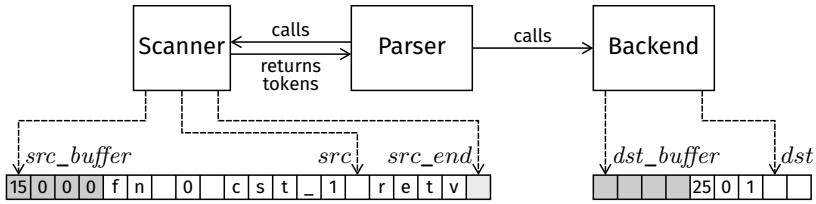


FIGURE 15.1 The 3 parts of our compiler (top) and its 2 data structures (bottom), here with 3 tokens of a 15 bytes program already read (left) and compiled (right).

- The compiled code must be produced in a data buffer. It must be the binary form of the bytecode instructions provided as input.
- The compiler should return 0 if the compilation was successful, and a non-zero value otherwise. In the latter case, the location of the error in the source code should be stored at the dst_buffer address.

Many errors could occur in the source code, such as an undefined opcode name ("cst_3"), an opcode without argument followed by integer value ("cst_0 10"), an opcode with argument not followed by an integer value ("cst8 cst8"), an opcode with an 8-bit argument followed by an integer greater than 255, a jump instruction opcode followed by an invalid instruction offset, etc. To simplify our task, in this chapter, we only require the detection of (most of the) undefined opcode names.

15.2 Algorithms

A compiler can generally be divided in at least 3 parts: a *scanner*, a *parser*, and a *backend* (see Figure 15.1). The scanner reads the source code and extracts its individual tokens. The parser calls the scanner to read tokens, and checks that they follow the programming language's grammar. The backend provides functions to generate the compiled code. In very simple compilers such as ours, the parser uses the backend to directly produce the compile code, while analyzing the source code.

Our compiler uses 5 main variables, shown in Figure 15.1. Besides src_buffer and dst_buffer, already defined, the most important ones are src and dst. src points to the next character to read. dst points to the next byte where compiled code must be written. Finally, src_end points to the next byte after the end of the source code. When src reaches src_end the whole program has been read and the compiler returns.

Scanner The scanner splits the source code in tokens, detects invalid tokens, and returns some data about each token. For instance, it should detect that "`cst_3`" is invalid, and it could return 42 for the token "42" ($34_{16}32_{16}$ in ASCII). To simplify, in this chapter, we move the error detection in the backend. A token is then any sequence of characters which does not contain a space, a tabulation, or a "new line". To compute the numerical value v of an integer token "$c_{n-1} \dots c_1 c_0$", we can initialize v to 0 and

ALGORITHM 15.1 Reading a token and returning its value v.

1. while $src < src\_end$ and the character at src is a space, tab or "new line"
2. increment src by 1 to skip this character
3. if $src = src\_end$ return nothing
4. initialize v to 0
5. while $src < src\_end$ and the character c at src is not a space, tab or "new line"
6. update v to $10v + (c - 30_{16})$
7. increment src by 1
8. return v

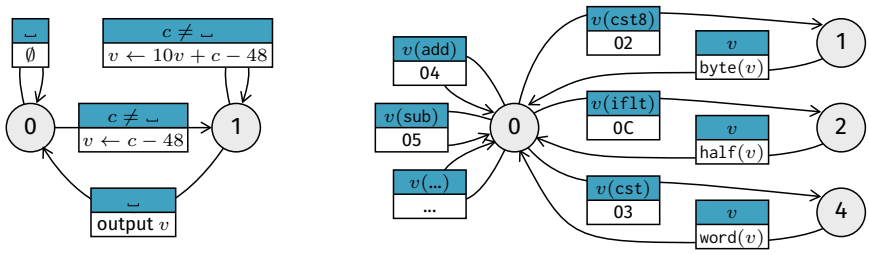


FIGURE 15.2 The scanner and parser can be modeled with Finite State Machines (see Section 11.4.1) reading characters c (left) and token values v (right), respectively. ␣ represents a space, tab or "new line". Many parser transitions are not shown.

update v to $10v+(c_i-30_{16})$, for each character c_i from left to right. In fact, to simplify the initial compiler, the scanner returns such a value for *all* tokens. For instance, for the "`fn`" token ($66_{16}6E_{16}$ in ASCII), it returns $10(66_{16} - 30_{16}) + (6E_{16} - 30_{16}) = 602$. In summary, a token is read as described in Algorithm 15.1, which also corresponds to the finite state machine in Figure 15.2.

Backend The backend provides functions to write opcodes and their arguments in the output buffer. Here we mostly need functions to write 8-bit and 16-bit values in memory, plus some code to detect invalid opcodes. Valid opcodes are between 0 and 31 included, but we also add here a pseudo opcode `d` (for "data", with value 32), with an 8-bit argument x. Once compiled, a `d` x instruction simply produces the byte x. It can be used to mix code and data (such as the transition table of our keyboard driver). In summary, a function to write an *opcode* (without its argument) should return an error if $opcode > 32$, do nothing if $opcode = 32$, or write the *opcode* byte otherwise.

Parser The parser calls the scanner to read the source code one token at a time, and generates the corresponding compiled code with the backend. For our very simple initial compiler, the parser can be modeled with a Finite State Machine, represented in Figure 15.2. There are 4 states, corresponding to the expected "type" of the next token returned by the scanner. State 0 corresponds to opcode tokens, such as `add`. States 1,

token	v	$opcode$	S
cst_0	8EA4E	00	0
cst_1	8EA4F	01	0
cst8	E414	02	1
cst	16CE	03	4
add	1560	04	0
sub	1D10	05	0
mul	1AC2	06	0
div	16D0	07	0
and	15C4	08	0
or	2B8	09	0
lsl	1A4A	0A	0
lsr	1A50	0B	0
iflt	F65C	0C	2
ifeq	F613	0D	2
ifgt	F62A	0E	2
ifle	F64D	0F	2
ifne	F661	10	2

token	v	$opcode$	S
ifge	F61B	11	2
goto	F25B	12	2
load	1051A	13	0
store	B5E35	14	0
ptr	1BEA	15	1
get	17D2	16	1
set	1C82	17	1
pop	1BB6	18	0
fn	25A	19	1
call	DCF0	1A	2
callr	8A1A2	1B	2
calld	8A194	1C	0
ret	1C1E	1D	0
retv	11972	1E	0
blx	1628	1F	0
d	34	20	1

TABLE 15.1 The token value v and the corresponding compiled $opcode$ and next state S for each valid opcode token.

2, and 4 correspond to 1, 2, and 4-byte opcode arguments, respectively (such as the argument of `cst8`, `iflt`, and `cst`, respectively). In these 3 states, any token value v should simply be written at dst in 1, 2, or 4 bytes, and the next state is state 0. In state 0, the opcode corresponding to the token value v, noted $opcode(v)$, should be written at dst. And the next state, noted $S(v)$, should be either 0, 1, 2, or 4, depending on this opcode. By listing the opcode names and computing their token values v with Algorithm 15.1, we get $opcode(v)$ and $S(v)$, shown in Table 15.1.

In order to implement this Finite State Machine we need functions to compute $opcode(v)$ and $S(v)$. For this, the easiest is to store Table 15.1 in memory. $opcode(v)$ can then be computed by finding the row corresponding to v, and then returning the value in its $opcode$ column – and similarly for $S(v)$. In fact, since the $opcode$ of the i^{th} row is i, we don't need to store this column. Notice also that the least significant byte $lsb(v)$ of the token values v are all unique. We can thus store only one byte per value in this column. In summary, $opcode(v)$ and $S(v)$ can be computed as described in Algorithm 15.2, where `LSB` and `S` are the $lsb(v)$ and $S(v)$ value lists.

Note that for the invalid token `cst_3`, $lsb(v)$ is equal to 51, which is not in `LSB`. In such cases, Algorithm 15.2 returns the invalid opcode 33. Hence, most invalid tokens can be detected by checking for invalid opcodes. However, some invalid tokens, such as `cst_2`, cannot be detected like this because the least significant byte of their token value *is* in `LSB`. We fix this in Chapter 16, at the price of a greater complexity.

ALGORITHM 15.2 Computing $\{opcode(v), S(v)\}$ for a token value v.

LSB = [4E, 4F, 14, CE, . . .], S = [0, 0, 1, 4, . . .]
1. initialize i to 0
2. while $i \leq 32$ and the i^{th} value in LSB is not equal to $(v \wedge 255)$
3. increment i by 1
4. return $\{i, i^{th}$ value in S$\}$

15.3 Implementation

We can now implement this initial compiler. We do this in a new data buffer, in the next flash memory page after the text editor (*i.e.*, at address $C0F00_{16}$). We start with the scanner, with a function returning 1 if a given character c is a space, a tabulation, or a "new line" (20_{16}, 09_{16}, and $0A_{16}$ in ASCII, respectively), and 0 otherwise:

```
tc_is_space(c) → bool
fn     01              19  C0F04
get    00   c          16  +002
cst8   20              02  +004
ifeq   0019            0D  +006
get    00   c          16  +009
cst8   09              02  +00B
ifeq   0019            0D  +00D
get    00   c          16  +010
cst8   0A              02  +012
ifeq   0019            0D  +014
cst_0                  00  +017
retv                   1E  +018
cst_1                  01  +019
retv                   1E  +01A
```

We then implement Algorithm 15.1 in two parts. Steps 1 and 2 are implemented in the following function, which returns the new src value:

tc_skip_spaces(src, src_end) → src'

Initialize src' to src.

Step 1. If src' (in the 6^{th} stack frame slot) is greater than or equal to src_end, go to the last instruction.

If the character at src' is not a spacing character, go to the last instruction.

Step 2. Increment src' (the top stack value) by 1 and go back to step 1.

Return the top stack value src'.

```
fn     02                    19  C0F1F
get    00   src → src'       16  +002
get    06   src'             16  +004
get    01   src_end          16  +006
ifge   001C                  11  +008
get    06   src'             16  +00B
call   03C0 load_byte        1A  +00D
call   0F04 tc_is_space      1A  +010
cst_1                        01  +013
ifne   001C                  10  +014
cst_1                        01  +017
add                          04  +018
goto   0004                  12  +019
retv                         1E  +01C
```

Steps 5 to 7 are implemented in the next function, which also returns the new src value (we assume that steps 3 and 4 are done by the caller). Since a function can't

return several values, it can't return v as described in Algorithm 15.1. Instead, it takes as parameter a *pointer* v^p to a memory word where v can be read and modified:

`tc_read_token`(src, src_end, v^p) $\rightarrow$ src'

Initialize src' to src.

Step 5. If src' (in the 7^{th} stack frame slot) is greater than or equal to src_end, go to the last instruction.

If the character at src' is a spacing character, go to the last instruction.

Step 6. Update v, at address v^p, to $10v + (c - 30_{16})$, where c is the character at src'. To simplify, we do not check if this new value actually fits in a word.

Step 7. Increment src' (the top stack value) by 1 and go back to step 5.

Return the top stack value src'.

fn	03			**19**	C0F3C
get	00	$src \rightarrow src'$		**16**	+002
get	07	src'		**16**	+004
get	01	src_end		**16**	+006
ifge	002E			**11**	+008
get	07	src'		**16**	+00B
call	03C0	load_byte		**1A**	+00D
call	0F04	tc_is_space		**1A**	+010
cst_1				**01**	+013
ifeq	002E			**0D**	+014
get	02	v^p		**16**	+017
get	02	v^p		**16**	+019
load				**13**	+01B
cst8	0A			**02**	+01C
mul				**06**	+01E
get	07	src'		**16**	+01F
call	03C0	load_byte		**1A**	+021
cst8	30			**02**	+024
sub				**05**	+026
add				**04**	+027
store				**14**	+028
cst_1				**01**	+029
add				**04**	+02A
goto	0004			**12**	+02B
retv				**1E**	+02E

This concludes the scanner part. We continue with the backend part. As said above, we mostly need here functions to store 8-bit and 16-bit values in memory. We already have a `store_byte` function (see Table 13.1), hence we only need a new `store_half` function (very similar to `store_byte`, already explained):

`store_half`(ptr, $value$)

fn	02		**19**	C0F6B
get	00	ptr	**16**	+002
get	00	ptr	**16**	+004
load			**13**	+006
cst	FFFF0000		**03**	+007
and			**08**	+00C
get	01	$value$	**16**	+00D
or			**09**	+00F
store			**14**	+010
ret			**1D**	+011

We finish the backend part with a function to write an *opcode* byte at dst, which returns the new dst value, dst'. As described above, this function returns an error (represented with $dst = 0$) if $opcode > 32$, and does nothing if $opcode = 32$:

`tc_write_opcode`(dst, $opcode$) $\rightarrow$ dst'

fn	02		**19**	C0F7D
get	01	*opcode*	**16**	+002
cst8	21		**02**	+004
ifne	000B		**10**	+006
cst_0			**00**	+009
retv			**1E**	+00A
get	01	*opcode*	**16**	+00B
cst8	20		**02**	+00D
ifne	0015		**10**	+00F
get	00	*dst*	**16**	+012
retv			**1E**	+014
get	00	*dst*	**16**	+015
get	01	*opcode*	**16**	+017
call	0A04	`store_byte`	**1A**	+019
get	00	*dst*	**16**	+01C
cst_1			**01**	+01E
add			**04**	+01F
retv			**1E**	+020

We continue the implementation with the parser part, starting with Algorithm 15.2. We first store the `LSB` and `S` tables, at addresses $C0F9E_{16}$ and $C0FC0_{16}$, respectively (note that we end each table with a 33^{rd} 0 value, since i can be equal to 33 at step 4 of Algorithm 15.2):

```
EA351A5B 1B614D2A 135C504A B8C4D0C2 1060CE14 4F4E....   C0F9C
00000000 00000000 04010000 00342872 1E94A2F0 5AB682D2   C0FB4
    0001 00000000 02020100 01010100 00020202 02020202   C0FCC
```

We then implement Algorithm 15.2 in the following function. Since a function can't return several values, it returns the *opcode* only, and stores S at an address S^p passed as parameter. Note also that this function takes the least significant byte lsb of v as parameter (instead of v in Algorithm 15.2):

`tc_get_opcode`(lsb, S^p) $\rightarrow$ *opcode*

Step 1. Initialize i to 0.

Step 2. If i (in the 6^{th} stack frame slot) is greater than 32, go to step 4.

If the i^{th} value in `LSB` is equal to lsb, go to step 4.

Step 3. Increment the top stack value i by 1 and go back to step 2.

Step 4. Store the i^{th} value of `S` at S^p and return the top stack value i.

fn	02		**19**	C0FE2
cst_0		$\rightarrow i$	**00**	+002
get	06	i	**16**	+003
cst8	20		**02**	+005
ifgt	001F		**0E**	+007
cst	000C0F9E		**03**	+00A
get	06	i	**16**	+00F
add			**04**	+011
call	03C0	`load_byte`	**1A**	+012
get	00	*lsb*	**16**	+015
ifeq	001F		**0D**	+017
cst_1			**01**	+01A
add			**04**	+01B
goto	0003		**12**	+01C
get	01	S^p	**16**	+01F
cst	000C0FC0		**03**	+021
get	06	i	**16**	+026
add			**04**	+028
call	03C0	`load_byte`	**1A**	+029

```
store          14   +02C
retv           1E   +02D
```

With this we can now implement a function to perform a transition of the parser's Finite State Machine. The following function takes a pointer S^p to the current state S as parameter, as well as a token value v and the current value of dst. It performs the corresponding action, updates the value at S^p to the next state, and returns the new dst value. It has 4 main parts, corresponding to the 4 possible values of the current state, plus a shared 5^{th} part:

`tc_parse_token`(dst, v, S^p) $\rightarrow$ dst'

Get the value S at address S^p.

Part 1. If S (in the 7^{th} stack frame slot) is not 0, go to part 2.

Otherwise, compute $opcode(v)$ and store $S(v)$ at S^p by calling `tc_get_opcode` on the least significant byte of v, $v \wedge 255$. Write this opcode at dst by calling `tc_write_opcode`, and return the result.

Part 2. If S is not 1, go to part 3.

Otherwise, store the byte v at dst and go to part 5 (to simplify we do not check if v actually fits in a byte).

Part 3. If S is not 2, go to part 4.

Otherwise, store the half word v at dst and go to part 5 (to simplify we do not check if v actually fits in a half word).

Part 4. S is necessarily equal to 4. Store the word v at dst and continue to part 5.

Part 5. Update the value at S^p to 0, the next state after a transition from state 1, 2, or 4.

```
fn     03                    19   C1010
get    02   Sp               16   +002
load        → S              13   +004
get    07   S                16   +005
cst_0                        00   +007
ifne   001B                  10   +008
get    00   dst              16   +00B
get    01   v                16   +00D
cst8   FF                    02   +00F
and                          08   +011
get    02   Sp               16   +012
call   0FE2 tc_get_opcode    1A   +014
call   0F7D …write_opcode    1A   +017
retv                         1E   +01A
get    07   S                16   +01B
cst_1                        01   +01D
ifne   002B                  10   +01E
get    00   dst              16   +021
get    01   v                16   +023
call   0A04 store_byte       1A   +025
goto   0041                  12   +028
get    07   S                16   +02B
cst8   02                    02   +02D
ifne   003C                  10   +02F
get    00   dst              16   +032
get    01   v                16   +034
call   0F6B store_half       1A   +036
goto   0041                  12   +039
get    00   dst              16   +03C
get    01   v                16   +03E
store                        14   +040
get    02   Sp               16   +041
cst_0                        00   +043
store                        14   +044
```

Return the new dst value, $dst + S$ (since S is the number of bytes just written).

```
get   00  dst      16  +045
get   07  S        16  +047
add               04  +049
retv              1E  +04A
```

We can finally implement the compiler's main function. It starts by initializing src to $src\_buffer + 4$, src_end to $src + \text{mem32}[src\_buffer]$, dst to $dst\_buffer + 4$, the token value v to 0, and the Finite State Machine state S to 0, in stack frame slots 6, 7, 8, 9, and 10, respectively:

`tc_main`(src_buffer, dst_buffer) $\rightarrow$ $error$

```
fn    02              19  C105B
get   00  src_buffer  16  +002
cst8  04              02  +004
add       → src       04  +006
get   06  src         16  +007
get   00  src_buffer  16  +009
load                  13  +00B
add       → src_end   04  +00C
get   01  dst_buffer  16  +00D
cst8  04              02  +00F
add       → dst       04  +011
cst_0     → v         00  +012
cst_0     → S         00  +013
```

It continues with a loop which 1) skips spaces and returns 0 if src_end is reached, 2) reads a token and performs the corresponding Finite State Machine transition, 3) returns 1 if an invalid token was found:

Step 1. Update src to the result of `tc_skip_spaces`(src, src_end).

```
get   06  src          16  +014
get   07  src_end      16  +016
call  0F1F …skip_spaces 1A  +018
set   06  src          17  +01B
```

Step 2. If $src < src\_end$, go to step 3.

```
get   06  src          16  +01D
get   07  src_end      16  +01F
iflt  0031             0C  +021
```

Otherwise, *i.e.*, if the end of the program is reached, set the value at dst_buffer to the number of bytes written, $dst - dst\_buffer - 4$, and return 0 (meaning "no error"). To simplify, we do not check if S is 0 (if not the program ends in the middle of an instruction, which is an error).

```
get   01  dst_buffer   16  +024
get   08  dst          16  +026
get   01  dst_buffer   16  +028
sub                    05  +02A
cst8  04               02  +02B
sub                    05  +02D
store                  14  +02E
cst_0                  00  +02F
retv                   1E  +030
```

Step 3. Call the scanner to read a token and store its value in v. Update src to the result of `tc_read_token`.

```
get   06  src          16  +031
get   07  src_end      16  +033
ptr   09  v            15  +035
call  0F3C tc_read_token 1A +037
```

Step 4. Perform the Finite State Machine transition corresponding to v. Update dst to the result of `tc_parse_token`.

Step 5. If $dst \neq 0$, go to step 6.

Otherwise, *i.e.*, if an invalid token has been read, set the value at dst_buffer to the location of the error, $src - src\_buffer - 4$, and return 1 (meaning "error").

Step 6. Reinitialize v to 0 for the next loop iteration, and go back to step 1.

```
set    06  src              17  +03A
get    08  dst              16  +03C
get    09  v                16  +03E
ptr    0A  S                15  +040
call   1010 …parse_token    1A  +042
set    08  dst              17  +045
get    08  dst              16  +047
cst_0                       00  +049
ifne   005A                 10  +04A
get    01  dst_buffer       16  +04D
get    06  src              16  +04F
get    00  src_buffer       16  +051
sub                         05  +053
cst8   04                   02  +054
sub                         05  +056
store                       14  +057
cst_1                       01  +058
retv                        1E  +059
cst_0                       00  +05A
set    09  v                17  +05B
goto   0014                 12  +05D
```

In summary the full code of our initial compiler is the following:

```
00001900 0A020016 00190009 02001600 19002002 00160019  C0F04
10010F04 1A03C01A 06160001 11011606 16001602 191E011E  C0F1C
03C01A07 16002E11 01160716 00160319 1E000412 04010010  C0F34
04053002 03C01A07 16060A02 13021602 16002E00 010F041A  C0F4C
14090116 08FFFF00 00031300 16001602 191E0004 12040114  C0F64
00161E00 16001510 20020116 1E00000B 10210201 16021901  C0F7C
135C504A B8C4D0C2 1060CE14 4F4E1E04 01001600 041A0116  C0F94
04010000 00342872 1E94A2F0 5AB682D2 EA351A5B 1B614D2A  C0FAC
02020100 01010100 00020202 02020202 00000000 00000000  C0FC4
04061600 0C0F9E03 001F0E20 02061600 02190001 00000000  C0FDC
1A040616 000C0FC0 03011600 03120401 001F0D00 16003C01A  C0FF4
021608FF 02011600 16001B10 00071613 02160319 1E1403C0  C100C
16004112 0A041A01 16001600 2B100107 161E0F7D 1A0FE21A  C1024
00021614 01160016 0041120F 6B1A0116 0016003C 10020207  C103C
04020116 04130016 06160404 02001602 191E0407 16001614  C1054
16081601 16003100 07160616 06170F1F 1A071606 16000004  C106C
15091608 1606170F 3C1A0915 07160616 1E001405 04020501  C1084
01140504 02050016 06160116 005A1000 08160817 10101A0A  C109C
                                     001412 09170001E  C10B4
```

To store it in flash memory we must enter it in RAM first, lets say at address

20070000_{16}, and then save it in flash. In the memory editor, type "w20070000"+Enter, and then store the compiler size in bytes at this address by typing "w000001b7"+Enter. Continue by entering each word of the compiler code, listed above, by typing its value followed by Enter. Finally, save this code in flash memory (starting at $page = 15$) by running the following function:

```
save()
fn    00                19 00000
cst   20070000          03 +002
```

```
cst8  0F                02 +007
call  0B14 buffer_flash 1A +009
ret                     1D +00C
```

For this enter the full code of the above function in an unused RAM region, for instance starting at address 20080000_{16}:

```
1D 0B141A0F 02200700 00030019  00000
```

Then type "w20080000" followed by "r" to run it. Alternatively, if you don't want to enter the full compiler code manually with the memory editor, which is a bit tedious, you can "cheat" by saving it via an external computer, as follows. First run the `boot_mode_select_rom` function by typing "w000c02b4"+Enter, followed by "r". Then reset the Arduino and, on the host computer, run the following command to flash the compiler code and reset the Arduino again:

```
user@host:~$ python3 flash_helper.py < part3/opcodes_compiler.txt
>Reading page 1039... Done.
Reading page 1040... Done.
Writing page 1039... Done.
Writing page 1040... Done.
>Done.
```

15.4 Command editor

We can now write and compile our very first program in textual form. For this we first need to enter it in memory with the text editor. This requires calling the text editor, and then the compiler, with specific arguments. In turn, this currently requires typing a few bytecode instructions *in binary form* with the memory editor, as we did above to call `buffer_flash`. To avoid having to do this in the next chapters, our first program is a *command editor*. Its goal is to edit, compile and run small functions, called *commands*, such as the `save` function above.

15.4.1 User interface

A task such as writing and compiling a program requires less than a dozen distinct commands to edit the program, save it, compile it, save the compiled code, etc. However, each command must usually be run several times (if the compiler returns an error, the program must be edited, saved, and compiled again). In order to avoid

having to repeatedly type the same commands, the command editor should be able to save up to 12 distinct commands in flash memory. We number them from 1 to 12. It should then be able to load an existing command, and to edit it if necessary. Finally, it should be able to compile and run a command. To fulfill these requirements we define the command editor user interface as follows:

- typing a "Fi" key between "F1" and "F12" included should load command number i and display it. This command becomes the *current command*.
- typing "e" should run the text editor to edit the current command. Each command must be a function without argument, returning an integer value.
- typing "s" should save the current command in flash memory.
- typing "r" should compile the current command, run it, display its result, and wait until Enter is pressed (and not until any key press because releasing "r" can appear as a key press for commands using the flash memory driver – see Section 13.2). If the compilation fails, the compiler result should be displayed instead.
- typing Escape should exit the command editor.

Finally, when launched, the command editor should load and display command number 1. All commands are initially empty in flash memory.

15.4.2 Implementation

We can now write the command editor source code. For this we assume that its compiled code will eventually be stored in the next page after the opcodes compiler, *i.e.*, at address $C1100_{16}$=$C0000_{16}$+4352.

To implement the above requirements we reserve 12 pages of flash memory, one for each command, starting at address $D0000_{16}$. This gives $256 - 4 = 252$ bytes for the source code of each command, stored as a data buffer (see Section 13.1.3). We can then write a function to load command number *command* (here numbered from 0 to 11) at address *dst* (the right column shows source code; in particular, all numbers are in decimal form):

`ced_load`(*command*, *dst*)	**fn**	2		04356
Initialize *dst* to an empty buffer.	**get**	1	*dst*	+002
	cst_0			+004
	store			+005
Compute the *src* address of *command*.	**cst**	851968		+006
This is $D0000_{16} + 256 * command$.	**get**	0	*command*	+011
	cst8	8		+013
	lsl			+015
	add		$\rightarrow$ *src*	+016
If the *src* buffer size is greater than	**get**	6	*src*	+017
252 this means that no command has ever	**load**			+019
been stored here (each flash memory bit is	**cst8**	252		+020

initialized to 1). Then return directly.

Otherwise copy the *src* buffer to *dst* and return.

```
ifgt  32                    +022
get   6   src               +025
get   1   dst               +027
call  2708 buffer_copy      +029
ret                         +032
```

We continue with a function to display the command at *src*. For this we simply reuse the `ted_draw` function of the text editor:

`ced_draw`(*src*)

Set the color to yellow, to make it easier to distinguish the command editor and the text editor (which draws text in green).

Compute the *begin* address of the text, which is 4 bytes after *src*.

Compute the *end* address of the text, which is *n* bytes after *begin* (where *n*, the *src* buffer size, is the value at address *src*).

Draw the text with a zero *gap* and a *cursor* at the end (see Chapter 14).

```
fn    1                     04389
cst8  7                     +002
cst8  7                     +004
cst_0                       +006
call  1101 gpu_set_color    +007
get   0   src               +010
cst8  4                     +012
add       → begin           +014
get   5   begin             +015
get   0   src               +017
load                        +019
add       → end             +020
get   5   begin             +021
get   6   end               +023
cst_0                       +025
get   6   end               +026
call  3450 ted_draw         +028
ret                         +031
```

The next function compiles the source code at *src*, writes the compiled code at *dst*, and runs it. It then displays the result and waits until Enter is pressed.

`ced_run`(*src*, *dst*)

Compile the code. The result, noted *error*, is pushed in the 6^{th} stack frame slot.

If the compilation is successful (*i.e.*, if $error = 0$), run the compiled code (which starts after the 4 bytes *dst* header) and store its result in *error*. Otherwise skip this step.

Clear the screen, set the cursor to the

```
fn    2                     04421
get   0   src               +002
get   1   dst               +004
call  4187 tc_main          +006
get   6   error             +009
cst_0                       +011
ifne  23                    +012
get   1   dst               +015
cst8  4                     +017
add                         +019
calld                       +020
set   6   error             +021
call  1033 …clear_screen    +023
```

top-left corner, draw *error* in hexadecimal, and wait until Enter is pressed.

```
cst_0                                  +026
cst_0                                  +027
call    1058 …set_cursor               +028
get     6    error                     +031
call    1868 …draw_hex_word            +033
call    1504 …get_char                 +036
cst8    10                             +039
ifne    36                             +041
ret                                    +044
```

We can finally write the main command editor function. This function loops until Escape is pressed, and performs the appropriate action for any other typed key. It loads the current command in the 256 bytes region starting at address 20070000_{16}, and compiles and runs it the next 256 bytes.

`command_editor()`

```
fn      0                              04466
```

Initialize *src* to 20070000_{16}.

```
cst     537329664 → src                +002
```

Initialize *command* to 0.

```
cst_0         → command                +007
```

Initialize c to "F1" (see Table 11.3).

```
cst8    128   → c                      +008
```

Step 1. If c is not the Escape key go to step 2. Otherwise return.

```
get     6    c                         +010
cst8    27                             +012
ifne    18                             +014
ret                                    +017
```

Step 2. If c is not between "F1" and "F12" included go to step 3.

```
get     6    c                         +018
cst8    128                            +020
iflt    49                             +022
get     6    c                         +025
cst8    139                            +027
ifgt    49                             +029
```

Otherwise set *command* to $c-$"F1".

```
get     6    c                         +032
cst8    128                            +034
sub                                    +036
set     5    command                   +037
```

Then load this new command and go to step 6 to display it.

```
get     5    command                   +039
get     4    src                       +041
call    4356 ced_load                  +043
goto    110                            +046
```

Step 3. If c is not equal to "e" go to step 4.

```
get     6    c                         +049
cst8    101                            +051
ifne    67                             +053
```

Otherwise call the text editor to edit the current command (with a maximum text length of 252 bytes). Then go to step 6 to display it.

```
get     4    src                       +056
cst_0                                  +058
cst8    252                            +059
call    3581 text_editor               +061
```

Step 4. If c is not equal to "e" go to step 5.

Otherwise save the current command at address $D0000_{16}+256*command$, which corresponds to page $256 + command$.

Step 5. If c is not equal to "r" go to step 6.

Otherwise compile and run the current command. The compiled code is written at $dst = src + 256$. Then continue to step 6.

Step 6. Draw the current command, wait for a key to be pressed, store it in c, and go back to step 1 to handle it.

```
goto 110                          +064
get  6   c                        +067
cst8 115                          +069
ifne 90                           +071
get  4   src                      +074
cst  256                          +076
get  5   command                  +081
add                               +083
call 2836 buffer_flash            +084
goto 110                          +087
get  6   c                        +090
cst8 114                          +092
ifne 110                          +094
get  4   src                      +097
get  4   src                      +099
cst  256                          +101
add                               +106
call 4421 ced_run                 +107
get  4   src                      +110
call 4389 ced_draw                +112
call 1542 …wait_char              +115
set  6   c                        +118
goto 10                           +120
```

The command editor implementation is now complete, and is summarized below:

```
fn 2
  get 1 cst_0 store
  cst 851968 get 0 cst8 8 lsl add
  get 6 load cst8 252 ifgt 32 get 6 get 1 call 2708
  ret
fn 1
  cst8 7 cst8 7 cst_0 call 1101
  get 0 cst8 4 add
  get 5 get 0 load add
  get 5 get 6 cst_0 get 6 call 3450
  ret
fn 2
  get 0 get 1 call 4187
  get 6 cst_0 ifne 23 get 1 cst8 4 add calld set 6
  call 1033
  cst_0 cst_0 call 1058
  get 6 call 1868
  call 1504 cst8 10 ifne 36
  ret
fn 0
```

```
cst 537329664
cst_0
cst8 128
get 6 cst8 27 ifne 18 ret
get 6 cst8 128 iflt 49 get 6 cst8 139 ifgt 49
get 6 cst8 128 sub set 5
get 5 get 4 call 4356 goto 110
get 6 cst8 101 ifne 67 get 4 cst_0 cst8 252 call 3581 goto 110
get 6 cst8 115 ifne 90 get 4 cst 256 get 5 add call 2836 goto 110
get 6 cst8 114 ifne 110 get 4 get 4 cst 256 add call 4421
get 4 call 4389
call 1542 set 6 goto 10
```

15.4.3 Compilation

We now need to type this source code with the text editor, save it, compile it, and store the compiled code. These 4 steps are explained below.

Edit

Typing the source code requires launching the text editor first. For this, in the memory editor, type "w20080000"+Enter, followed by the code below (see Section 14.4):

```
1D 0DFD1A00 00100003 00200700 00030019  00000
```

Then initialize an empty text buffer by typing "w20070000"+Enter, followed by "00000000"+Enter. Run the text editor on this empty buffer by typing "20080000"+Enter, followed by "r". Finally, type the command editor source code listed above, followed by Escape to return in the memory editor.

Alternatively, if you don't want to type this source code, you can "cheat" by saving it via an external computer, as follows. First run the `boot_mode_select_rom` function by typing "w000c02b4"+Enter, followed by "r". Then reset the Arduino and, on the host computer, run the following command to flash the source code and reset the Arduino again (you can then skip the "Save" step below):

```
user@host:~$ python3 flash_helper.py < part3/command_editor.txt
```

Save

Before compiling this code we want to save it, in case something goes wrong. We can save it after the 12 pages reserved for the commands, at address $D0C00_{16}$, which corresponds to page $268 = 10C_{16}$. This can be done with the following function:

```
save_source()
fn    00              19  00000
cst   20070000        03  +002
cst   0000010C        03  +007
call  0B14 buffer_flash 1A  +00C
ret                   1D  +00F
```

Enter it in RAM after the "edit" function, at address 20080020_{16}, by typing "w20080020"+Enter, followed by the full code of this function, listed below. Then run it by typing "w20080020"+Enter, followed by "r".

```
1D0B141A 0000010C 03200700 00030019  00000
```

Compile

Compiling the code can be done with the following function, which writes the compiled code at address 20071000_{16} and the compiler's result value at address 20080060_{16}:

```
compile_source()
fn    00              19  00000
cst   20080060        03  +002
cst   000D0C00        03  +007
cst   20071000        03  +00C
call  105B tc_main    1A  +011
store                 14  +014
ret                   1D  +015
```

Enter it in RAM after the "save" function, at address 20080040_{16}, by typing "w20080040"+Enter, followed by the full code of this function, listed below. Then run it by typing "w20080040"+Enter, followed by "r".

```
1D14 105B1A20 07100003 000D0C00 03200800 60030019  00000
```

If all goes well the value at address 20080060_{16} should be 0, because the compiler returns 0 if and only if the compilation is successful. If this is not the case, run the "edit" function again, double check the source code and fix any error found (you can also get the location of the error at address 20071000_{16}). Then save and compile the code again. And repeat this until success.

Store

Once the compilation is successful, the compiled code can be stored in flash memory. The following function stores it in the next page after the compiler itself, *i.e.*, at address $C1100_{16}$, which corresponds to page $17 = 11_{16}$:

```
save_code()
fn    00              19  00000
cst   20071000        03  +002
cst8  11              02  +007
call  0B14 buffer_flash 1A  +009
ret                   1D  +00C
```

Enter it in RAM after the "compile" function, at address 20080080_{16}, by typing "w20080080"+Enter, followed by the full code of this function, listed below. Then run it by typing "w20080080"+Enter, followed by "r".

```
1D 0B141A11 02200710 00030019  00000
```

15.4.4 First commands

We can now try our command editor. Start it by typing "w000c1172"+Enter, followed by "r" (its main function is at address $C0000_{16}$+4466=$C1172_{16}$ – see Section 15.4.2). The screen should now be empty, because it displays command number 1, initially empty. Lets use this command to show a welcome message when the command editor starts. Type "e" to edit it, then type "Welcome to the command editor." followed by Escape. At this point the message you typed should be displayed in yellow. For now it is only in RAM. Type "s" to save it in flash memory. We now want to define some commands to create, load, edit, save, and compile a program.

New (F2) initializes an empty text buffer at address 537330176 = 20070200_{16}, just after the memory region used by the command editor (see Figure 15.3). Type "F2" followed by "e" to edit it, then type its source code followed by Escape and "s" (the dummy data at the end describes the command):

```
fn 0
  cst 537330176 cst_0 store
  cst_0 retv
d NEW_SOURCE_CODE
```

Load (F3) calls buffer_copy to load a program stored in flash memory at address 856064 = $D1000_{16}$, just after the command editor source code (see Figure 15.3). Store its source code in command number 3:

```
fn 0
  cst 856064 cst 537330176 call 2708
  cst_0 retv
d LOAD_SOURCE_CODE
```

Edit (F4) calls text_editor to edit the text buffer at address 20070200_{16}, with the word at address 537379328 = $2007C200_{16}$ as initial offset, and a maximum length of 48 KB (including the 4 bytes header). The initial offset corresponds to the header of a compiled code buffer (see Figure 15.3) which, in case of a compilation error, contains the error location. Hence, editing the source code after a compilation error opens the text editor at the location of this error. Store the following source code in command number 4:

```
fn 0
  cst 537330176 cst 537379328 load cst 49148 call 3581
  cst_0 retv
d EDIT_SOURCE_CODE
```

Save (F5) calls buffer_flash to save the edited program in flash memory at address $D1000_{16}$, which corresponds to page 272. Store it in command number 5:

```
fn 0
  cst 537330176 cst 272 call 2836
  cst_0 retv
d SAVE_SOURCE_CODE
```

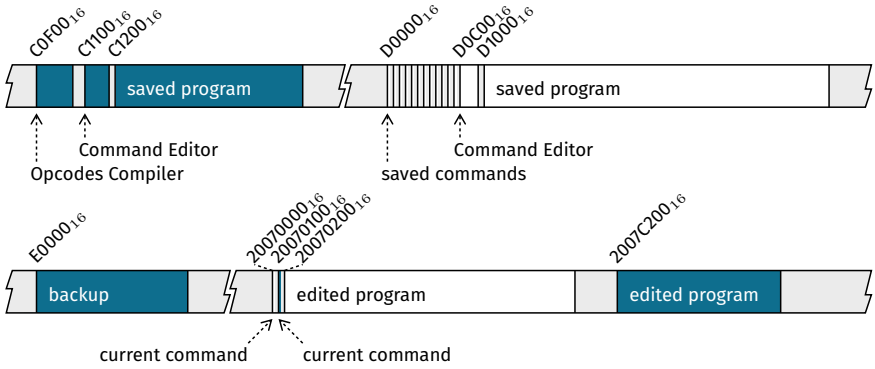


FIGURE 15.3 The flash memory and RAM regions used by the command editor, and by the commands defined in Section 15.4.4. White, blue and gray areas represent source code, bytecode and unused memory, respectively (not to scale).

Compile (F6) calls tc_main to compile the source code at address $D1000_{16}$, and to write the compiled code at address 537379328 = $2007C200_{16}$ (just after the source code in RAM, see Figure 15.3). It returns the compiler's result, which is non-zero if a compilation error occurs. Store it in command number 6:

```
fn 0
  cst 856064 cst 537379328 call 4187 retv
d COMPILE_SOURCE_CODE
```

Store (F7) calls buffer_flash to store the compiled code in flash memory at address 791040 = $C1200_{16}$, which corresponds to page 18 (after the command editor – see Figure 15.3). Before that, this command backs up the current compiled code by saving a copy of it at address 917504 = $E0000_{16}$, which corresponds to page 512. Store it in command number 7:

```
fn 0
  cst 791040 cst 512 call 2836
  cst 537379328 cst 18 call 2836
  cst_0 retv
d STORE_COMPILED_CODE
```

Restore (F8) calls buffer_flash to restore the backup created by the previous command, in case something goes wrong. Store it in command number 8:

```
fn 0
  cst 917504 cst 18 call 2836
  cst_0 retv
d RESTORE_BACKUP_CODE
```

15.4.5 Tests

In order to test the above commands, type “F2”+“r” to create a new program, and press Enter to return in the command editor. Then type “F4”+“r” to edit this program, and type the following code, which contains an error on purpose:

```
fn 0 cst_3 retv
```

Then type Escape to exit the text editor. The command’s result, 0, should be displayed. Press Enter to return in the command editor’s main loop (in the following we omit these “press Enter” instructions, for brevity).

Type “F5”+“r” to save this program, “F2”+“r” to create a new one, and “F4”+“r” to edit it. The screen should be empty. Type Escape to return in the memory editor, then type “F3”+“r” to load the previously saved program. Type “F4”+“r” to check that it is now loaded in RAM, and Escape to return in the command editor.

Type “F6”+“r” to compile this program. The result should be 1, because `cst_3` is an invalid opcode. Then type “F4”+“r” to edit the program. The cursor should be just after this invalid opcode. Enter the correct code below:

```
fn 0 cst8 3 retv
```

Finally, type Escape to exit the text editor, “F5”+“r” to save the corrected code, and “F6”+“r” to compile it. The result should be 0 this time.

CHAPTER

16 Labels Compiler

The opcodes compiler written in the previous chapter removes the need to manually convert each opcode into its numerical value, and to convert each opcode argument into the correct number of bytes. However, with this compiler, we still need to manually keep track of the address of each function, and of the offset of each instruction inside functions. Indeed, these values are needed for function call and jump instructions. To solve these issues, we implement in this chapter a new compiler, for a better programming language.

16.1 Requirements

So far we manually kept track of the address of each function by using a symbolic name for each function, and by keeping the address corresponding to each name in tables such as Table 12.1. To avoid this manual work, a solution is to use function names *in the program source code*, instead of outside of it (as we did so far). For instance, instead of writing "`fn 1`", we can write "`fn factorial 1`". We can then write "`call factorial`" to call this function, instead of something like "`call 4096`". From this, the compiler can do what we have been doing manually, *i.e.*, compute the address corresponding to each name, keep these addresses in a table, and replace each name with its corresponding address to produce the compiled code.

Similarly, so far, we used symbolic names such as "Step 1" to refer to some instructions in a function. And we added comments next to jump instructions, such as "go back to step 1", to make it easier to understand them. To avoid having to manually convert these symbolic names, called *labels*, into instruction offsets, we can use the same solution as above. Namely use labels in the program source code, instead of outside of it. For instance, we can write "`:step1 add`" to label an instruction, and then "`goto step1`" to jump to it (instead of something like "`goto 13`"). The compiler can then compute the offset corresponding to each label, keep them in a table, and replace each label with its corresponding offset to produce the compiled code.

Programs with bytecode instructions in textual form, augmented with function names and labels, can be defined with the following grammar:

program: (fn | static)* END
fn: "`fn`" fn_name fn_body

fn_name: IDENTIFIER
fn_body: INTEGER instruction* | ";"
instruction: label | "`cst_0`" | "`cst_1`" | "`cst8`" argument | "`cst32`" argument | ...
label: ":" IDENTIFIER
argument: INTEGER | IDENTIFIER
static: "`static`" IDENTIFIER INTEGER*

where IDENTIFIER refers to a token made of letters in "`a`" to "`z`", "`A`" to "`Z`", "`0`" to "`9`", or "`_`", not starting with a digit, and not equal to "`static`", "`fn`", "`cst_0`", etc.

The "fn" rule gives a symbolic name (defined by the "fn_name" rule) to a function. The body of this function is either an integer (its number of parameters) followed by a list of instructions, or a semi-colon. The latter case allows the introduction of a function name before its implementation is defined. It is explained in the next section.

The "instruction" rule is similar to the one in the previous chapter. The main differences are the introduction of labels (with the "label" rule), the generalization of arguments to either an INTEGER (as before) or an IDENTIFIER (which should be a function name, a label, or a static block name – see below), and the removal of the "`fn`" instruction (which now has its own rule, as explained above).

The "static" rule gives a symbolic name to the address of the first byte in a series of bytes. For instance, "`static FIBONACCI 0 1 1 2 3 5 8`" gives the name "`FIBONACCI`" to the address of the `0` byte, and compiles into the series of bytes `0 1 1 2 3 5 8`. It replaces the pseudo `d` instruction used in the previous chapter to insert data between code. And it avoids the need to manually compute the address of this data, just like function names.

Finally, the main "program" rule defines a program as any number of function and static blocks, in any order (the parentheses are not tokens; they mean that the "*" applies to their content). END is a special token representing the end of the source code. It does not correspond to any character. Its role is to ensure that the program is not followed by "garbage" content. Without it, for instance, "`lorem ipsum`" would be a valid (empty) program for the above grammar.

In this context, the precise requirements for our new compiler, also called an *assembler*, are mostly the same as in Section 15.1, but for the above grammar. The main difference is that the compiler now needs an additional input parameter, namely the address where the compiled code will be stored and executed. Indeed, this is needed to compute correct addresses in `call` instructions[1]. We call this new parameter $flash\_buffer$. We also want our compiler to detect more errors. In particular, it should detect all invalid opcode names, and all references to undefined names or labels.

16.2 Algorithms

Our new compiler is divided in a scanner, a parser and a backend, like the previous one. This section explains the algorithms used in each part.

[1]To simplify, the compiler does not produce position independent code (see Section 13.3).

16.2.1 Scanner

Based on the above grammar, the tokens that the scanner must recognize are integers, identifiers, *keywords* such as "`static`", "`fn`", "`cst_0`", etc, the colon and semi-colon characters, and the special END token. A token can thus no longer be represented with a single value v, as in the previous chapter. Instead, the scanner produces the following values for each token:

- $token$ identifies the token type. We use 0 for END, 1 for invalid tokens (such as "="), 2 for integers, 3 for identifiers, 115 ("s" in ASCII) for "`static`", 102 ("f" in ASCII) for "`fn`", 58 (":" in ASCII) for ":", 59 (";" in ASCII) for ";", and $opcode + 128$ for the token corresponding to $opcode$.
- $token\_data$ is the token's value for integer tokens (*e.g.*, 42 for the "42" token), or the address of the token's first character in the source code for identifier tokens.
- $token\_length$ is the number of characters of identifier tokens (*e.g.*, 3 for "`src`").

As before, the scanner could be represented with a Finite State Machine. But it would then have a lot of states. Another method is to use the following properties:

- keywords excepted, all tokens start with a different type of character: a digit for integers, a lowercase or uppercase letter (or an underscore) for identifiers, and a colon or a semi-colon for the single-character tokens.
- keywords are special cases of identifiers.

This suggests the following method to read a token: 1) look at its first character, 2) call a dedicated function to read the type of token corresponding to this first character, 3) if the token looks like an identifier, check if it is equal to a keyword and, if so, treat it as such. To implement this method, it is useful to have a `CHAR_TYPES` table indicating the type of each character: 1 for unsupported characters such as "=", 2 for digits, 3 for letters and the underscore, 58 for ":" and 59 for ";". It is also useful to have a `KEYWORDS` table of all the keywords, associated with their $token$ value. Each entry in this table can be stored as the keyword's number of characters n, followed by these n characters, and ending with the corresponding $token$ value:

`KEYWORDS` = 5 c s t _ 0 128 5 c s t _ 1 129 4 c s t 8 130 ... 0

where, to indicate the end of the table, a last "entry" with $n = 0$ is used. With this table, finding the $token$ value of an identifier or keyword starting at address $start$ and with $length$ characters can be done with Algorithm 16.2, which makes use of Algorithm 16.1 to compare two names (this algorithm simply compares the token characters with those of each keyword in the table, one by one).

The above method can then be split into 3 functions to read a character, an integer, or an identifier or keyword, called by a main function to read an arbitrary token:

- The function to read a character increments src by 1 and stores the *next* character to read and its `CHAR_TYPES` type in the new $next\_char$ and $next\_char\_type$ variables. Reading a character can then be done as described in Algorithm 16.3 (recall that src is defined as the address of the next character to read). In order to support the

ALGORITHM 16.1 Comparing the $size$ bytes starting at address $ptr1$ with those starting at address $ptr2$, and returning 0 if and only if they are equal.

1. initialize i to 0
2. while $i < size$ and the byte at $ptr1 + i$ is equal to the byte at $ptr2 + i$
3. increment i by 1
4. return $size$ - i

ALGORITHM 16.2 Computing the $token$ value of the identifier or keyword starting at address $start$ and made of $length$ characters.

1. initialize ptr to the address of the `KEYWORDS` table
2. load the byte at ptr (a keyword length) into len
3. if $len = 0$ return 3 (no keyword was found, hence this is an identifier)
4. if $len = length$ and if the len bytes starting at $start$ are equal to those starting at $ptr + 1$ (the keyword's characters)
5. return the keyword's $token$ value, *i.e.*, the byte at $ptr + len + 1$
6. otherwise
7. increment ptr by $len + 2$ and go back to step 2 to try the next keyword

END token, this algorithm "reads the character" at src_end by setting $next\_char$ and $next\_char\_type$ to 0. Trying to read any character after that returns an error.

- The function to read an integer reads characters one by one, while the next character is a digit. It computes its numerical value in $token\_data$ as in Algorithm 15.2, and returns the $token$ value of integer tokens (see Algorithm 16.4).
- The function to read an identifier or keyword reads characters one by one, while the next character is a letter, an underscore or a digit. It then returns the $token$ value computed with Algorithm 16.2 (see Algorithm 16.5).
- Finally, the function to read an arbitrary token reads characters one by one while the next character is a spacing character (for this it assumes that the `CHAR_TYPES` of the space, tab and "new line" characters is 32 – " " in ASCII). It then calls one of the above 3 functions, depending on the next character type (see Algorithm 16.6). It uses the property that the $token$ value of single character tokens and of the END token is equal to the character type (the `CHAR_TYPES` are chosen to ensure this).

Note that Algorithms 16.4 to 16.6 assume that the first character of the token to read, called a *lookahead* character, is already available in $next\_char$ and $next\_char\_type$. This property is ensured by the fact that, each time a character is read with Algorithm 16.3, the next character is stored in $next\_char$ and $next\_char\_type$.

16.2.2 Parser

The parser uses the scanner to read the source code, checks that the tokens follow the grammar, and generates the corresponding compiled code with the backend. Here the latter task requires building a table mapping identifiers to values (function names to

ALGORITHM 16.3 Reading a character.

1. if $src \geq src\_end$ return an error
2. increment src by 1, set $next\_char$ and $next\_char\_type$ to 0
3. if $src < src\_end$
4. set $next\_char$ to the character at src
5. set $next\_char\_type$ to the byte at CHAR_TYPES + $next\_char$

ALGORITHM 16.4 Reading an integer token.

1. initialize v to 0
2. while $next\_char\_type = 2$
3. update v to $10v + (next\_char - 30_{16})$
4. read a character with Algorithm 16.3
5. set $token\_data$ to v and return the $token$ value 2

function addresses, label names to instruction offsets), and using this table to find the value corresponding to an identifier. The following presents the algorithms used to do this, before presenting the overall parsing algorithm.

Symbol table

The above table can be stored in memory in several ways. We use here a *linked list* because it is simple to implement, although not very efficient. As its name implies, this is a list of elements, where each element has a pointer to (*i.e.*, the address of) the next one (or 0 for the last element). In our case we call each element a *symbol*. Besides a link to the next symbol, a symbol contains an identifier and a value. The identifier is represented with the address of its first character, and its length (see Figure 16.1).

Symbols must be stored in RAM in a different region than the dst_buffer used for the compiled code. We call this new region the *heap*, and we use a new $heap$ variable to indicate where a new symbol can be stored. We also use a new $symbols$ variable storing the address of the first symbol in the list (or 0 if the list is empty – see Figure 16.1). With this data structure, adding a new symbol in the table is very easy: we just need to store it in memory, starting at the $heap$ address, with $symbols$ as pointer to the next symbol. Finally, we need to set $symbols$ to $heap$, the address of the new first symbol, and to increase $heap$ by the number of bytes needed to store a symbol (see Figure 16.1). Finding the symbol corresponding to a given identifier I (specified by its address and length) is also very easy. We just need to iterate over all the symbols in the list, by using the pointers from each symbol to the next, until we find one whose identifier is equal to I (or we reach the end of the list – see Algorithm 16.7).

Forward references Sometimes a jump instruction needs to jump to a later instruction, whose offset is not yet known when the jump instruction is parsed and compiled. In such cases, called *forward references*, what we did manually so far was to leave a

ALGORITHM 16.5 Reading an identifier or keyword token.

1. set $start$ to src
2. while $next\_char\_type = 3$ or $next\_char\_type = 2$
3. read a character with Algorithm 16.3
4. set $length$ to $src - start$
5. set $token\_data$ to $start$ and $token\_length$ to $length$
6. return the token value computed with Algorithm 16.2 for $start$ and $length$

ALGORITHM 16.6 Reading an arbitrary token.

1. while $next\_char\_type = 32$
2. read a character with Algorithm 16.3
3. set $token$ to $next\_char\_type$
4. if $next\_char\_type = 2$ set $token$ to the result of Algorithm 16.4
5. if $next\_char\_type = 3$ set $token$ to the result of Algorithm 16.5
6. if $next\_char\_type \neq 0$ read a character with Algorithm 16.3

placeholder for the jump offset in the compiled code, and to fill it once the jump target was known. The compiler can do exactly the same, with the following algorithm:

- If a jump instruction uses a label which is not in the $symbols$ list, add a new symbol for this label, with the placeholder address as the symbol's value. To distinguish this symbol from normal ones, each symbol also stores its *kind*, which can be *resolved* (0) or, for forward references, *unresolved* (1) – see Figure 16.1.
- When a label is defined with the "*:label*" syntax, check if it is in the $symbols$ list. If so the symbol should be unresolved. Then fill the placeholder at the address stored in the symbol's value with the label offset (now known), store this offset in the symbol's value, and set the symbol kind to "resolved".

In fact several instructions can jump to a label which is not yet defined. In such cases we need to store several placeholder addresses. One way to do this would be to store in each placeholder the address of the previous one, yielding a linking list of placeholders. Unfortunately a placeholder has only 16 bits. To solve this issue, we can store instead in each placeholder the *offset* to the previous one (see Figure 16.2). In summary, adding a placeholder and filling a list of placeholders can be done as described in Algorithms 16.8 and 16.9. Finally, note that a call instruction can also refer to a function whose address is not yet known. These forward function references can be handled in exactly the same way as forward label references.

Local identifiers In compiled code a jump instruction can only jump to an instruction in the same function. Hence, in source code, a jump instruction should only use *local labels*, *i.e.*, labels defined in the same function. To ensure this, and to save memory at the same time, we can delete the symbols added for the labels of a function after it has

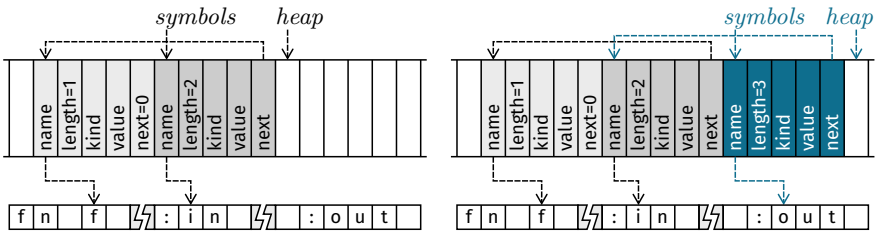


FIGURE 16.1 A linked list of symbols (top). Each symbol is made of 5 words. $name$ and $length$ refer to an identifier in the source code (bottom). $next$ is the address of the next symbol, or 0 for the last one (left). New symbols are stored at $heap$, which is then incremented by $5 * 4$ bytes (right). $symbols$ is the start of the linked list.

ALGORITHM 16.7 Finding the symbol corresponding to the identifier starting at address $name$ and with $length$ characters.

1. initialize $symbol$ to $symbols$
2. while $symbol \neq 0$
3. if the $symbol$'s $length$ is not equal to $length$ go to step 6
4. if the $symbol$'s $name$ is not equal to $name$ (cf. Algorithm 16.1), go to step 6
5. return $symbol$
6. update $symbol$ to the $symbol$'s $next$ value
7. no symbol has been found, return 0

been compiled. Doing this is very simple: we just need to save the value of the $heap$ and $symbols$ variables before compiling the function's body, and to restore $heap$ and $symbols$ to these saved values after the body has been compiled. Note however that this method removes *all* the symbols added during the function compilation. Hence, we cannot add a symbol for a forward function reference while compiling another (otherwise this forward reference would be lost and thus never resolved). This explains why, to simplify the compiler, we require an explicit *forward declaration* of functions with the "`fn` *name*`;`" syntax (compiled by adding an unresolved symbol for *name*, outside any function body).

Parsing algorithm

As in the previous chapter, the parser could be represented with a Finite State Machine. Doing so, however, would lead to a complex implementation. Moreover, in the next chapters, using a Finite State Machine is no longer possible. We thus use here another method, called a *recursive descent parser*. This method uses one parse_r function per rule r of the grammar. Each function checks that the next tokens follow the corresponding grammar rule. A parse_r function for a rule r using another rule r' can thus call parse_r' to check this subpart of r.

For instance, in our case, 3 of the parser functions are parse_fn, parse_fn_name,

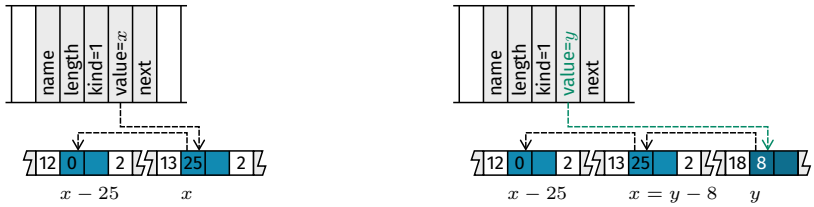


FIGURE 16.2 The $value$ of an unresolved symbol is the address x of its last placeholder in the compiled code (in blue). Each placeholder contains the offset to the previous one, or 0 for the first one. Adding a placeholder at address y simply requires storing $y - x$ in it, and updating $value$ to y (right).

ALGORITHM 16.8 Adding a new placeholder at y for an unresolved $symbol$.

1. initialize x to the $symbol$'s $value$ (the address of the previous placeholder)
2. set the $symbol$'s $value$ to y (the address of the new placeholder)
3. if $x = 0$, set x to y
4. set the half-word at y to the offset $y - x$ to the previous placeholder

and parse_fn_body, for the 3 rules fn, fn_name and fn_body. The parse_fn function is implemented by reading a token, checking that it is equal to "`fn`", and then calling parse_fn_name and parse_fn_body (because the fn rule is defined by "`fn`" fn_name fn_body – see Section 16.1). The parse_fn_body function is more complex because the fn_body rule has two alternatives. The body can either be an integer followed by instructions, or a semi-colon. To decide which alternative to use, the next token must be inspected. If it is an identifier the first alternative must be used. If it is a semi-colon the second one must be used. Any other case is an error. In the first case, zero or more instructions must be parsed after the integer. This can be done by calling the parse_instruction function, but how many times should it be called? The answer is as long as the next token is the start of an instruction. By looking at the grammar rules, we see that an instruction either starts with a label, or with an opcode keyword ("`cst_0`", etc). And a label starts with a colon. Thus, parse_instruction should be called as long as the next token is a colon or an opcode keyword. In summary, parse_fn_body can be implemented as follows:

1. if the next token is a semi-colon, read it
2. otherwise, if the next token is an integer
3. read it
4. while the next token is a colon or an opcode keyword, call parse_instruction
5. otherwise return an error

and the same principles can be used for all the other parsing functions. Note that such algorithms need the value of the "next token" (after the last one that has been read), called a *lookahead* token. For this we consider that what Algorithm 16.6 stores in $token$, $token\_data$, and $token\_length$ is actually about this lookahead token (by

ALGORITHM 16.9 Filling the placeholders of an unresolved $symbol$ to $value$.

1. initialize $placeholder$ to the $symbol$'s $value$
2. while $placeholder \neq 0$
3. set $offset$ to the half-word value at $placeholder$
4. store the half-world value $value$ at $placeholder$
5. if $offset = 0$ there is no previous placeholder, return
6. decrement $placeholder$ by $offset$

analogy with Algorithm 16.3). And, for this reason, we rename these variables to $next\_token$, $next\_token\_data$, and $next\_token\_length$ from now on. The above algorithm thus becomes:

1. if $next\_token$ is a semi-colon, read it with Algorithm 16.6
2. otherwise, if $next\_token$ is an integer
3. read it with Algorithm 16.6
4. while $next\_token$ is a colon or an opcode keyword, call parse_instruction
5. otherwise return an error

Finally, to produce the compiled code, calls to the backend are inserted in the parse_r functions where needed. For instance, for parse_fn_body, calls are inserted between steps 2 and 3 above to add the `fn` opcode to the compiled code, followed by $next\_token\_data$ (which contains the function's number of parameters). Other calls are also inserted to update the list of symbols when a new function or label is defined, and to use this list to compile function calls and jump instructions.

16.2.3 Error handling

Note that the above parse_fn_body algorithm can "return an error". In fact any parse_r function of the recursive descent parser might do so, and Algorithm 16.3 can as well. Thus, for instance, parse_program could call parse_fn, itself calling parse_fn_body, in turn calling parse_instruction, itself calling parse_label, where an error could be found (*e.g.*, if a colon is followed by integer). At this stage we would like to stop everything and simply return the error from parse_program. One way to do this is to check, after each function call, if an error occurred and, if so, immediately return from the current function. This is what we did in the previous chapter, but this was easy because only `tc_parse_token` could return an error, and it was called directly from the main `tc_compile` function. Doing the same here would be much more cumbersome.

Another solution is to return from parse_label, or in fact any function, *directly* into parse_program, without going through all the intermediate functions. But how can we do this? Here we need to remember how the bytecode interpreter returns from a function. The answer is by restoring 4 registers to the values which were saved on the callee's stack frame by the caller (see Chapter 8 and Figure 8.5). parse_label returns into parse_instruction because the saved register values in its stack frame were set by this caller. If we can somehow replace these saved register values with

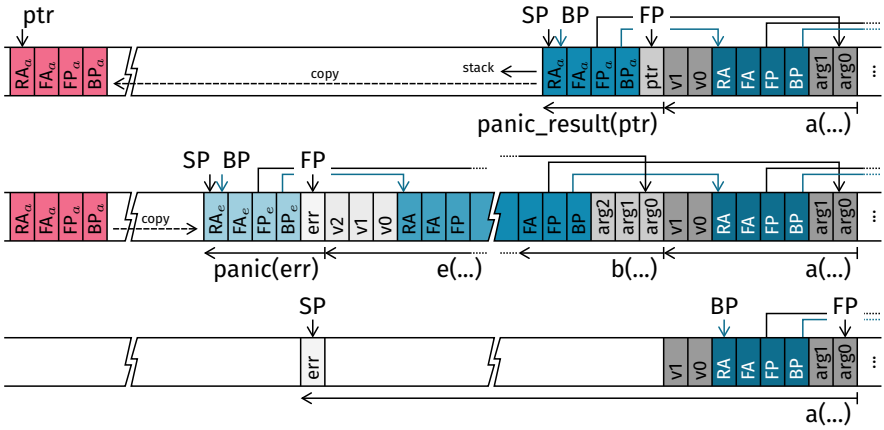


FIGURE 16.3 When `panic_result`(ptr) is called from `a`, the saved registers necessary to return in `a` (light blue) are copied at ptr (top, in red). Later on, in `e` called from `d`, called from `c`, ... called from `a`, a call to `panic(err)` copies the values at ptr in `panic`'s stack frame (middle). When `panic` returns err, it thus returns in a (bottom).

those set by parse_program when it calls another function, parse_label would actually return directly in parse_program! And this is in fact possible, since a function can access these saved register values in its stack frame (for instance with `get` and `set` instructions). The rest of this section presents a method to do this.

In a first step, the function we want to return to directly, say `a`, calls the predefined `panic_result`(ptr) function. This function copies at ptr the saved register values necessary to return into `a` (see Figure 16.3). This can be done by copying the 16 bytes starting 16 bytes before the address of the 0^{th} stack frame slot (given by `ptr` 0). `panic_result` then saves ptr at a fixed address A, and returns 0 ("no error").

Later on, for instance, `a` calls `b`, which calls `c`, ... which calls `e`, where an error err is found. To return this error directly into `a`, `e` calls the predefined `panic`(err) function. This function reads ptr at address A, and then copies the 16 bytes at ptr into its own stack frame (16 bytes before the address of the 0^{th} stack frame slot, as before – see Figure 16.3). It then returns err. By doing this, the Frame Pointer (FP) is restored to `a`'s stack frame, the Function Address (FA) is restored to `a`'s address, and the Instruction Counter (IC) is restored to `a`'s Return Address (RA), *i.e.*, to the instruction following the initial call to `panic_result`. In other words, from `a`'s point of view, `panic_result` is returning *twice*, the second time with the error value err (and from `e`'s point of view, `panic`(err) never returns). Note however that the Stack Pointer (SP) is updated by popping the `panic` stack frame, and then pushing the result value err. As a consequence, after a panic, `a`'s stack frame contains all the stack frames of the intermediate functions, with err on top (see Figure 16.3). When `a` itself returns, this large stack frame is popped and everything goes back to normal.

16.3 Implementation

We can now implement our new compiler. We first need to write it without using function names and labels, so that it can be compiled with the opcodes compiler (the only compiler we have for now). We then compile this source code, which gives us the labels compiler bytecode. Finally, we rewrite the labels compiler source code with function names and labels, and we compile it with the labels compiler bytecode.

In the following, to save space, we give the two compiler versions (with or without function names and labels) at the same time. Most of the code is the same in both versions, and is shown in black. Parts which are only in the 1^{st} version are highlighted in red. And parts which are only in the 2^{nd} version are highlighted in green. Some comments, in gray, make it easier to understand the code but are not part of it. These include function addresses and instruction offsets, shown at the beginning of each line (we still need to compute them manually in the 1^{st} version). Because of these comments, some lines are too long to fit on a page. In these cases they are wrapped on the next line, as indicated by ≀ marks (which are not part of the code).

We start with the compiler's `main` function, so that its address is easy to obtain ($C1200_{16} + 4 = C0000_{16} + 4612$, see Figure 15.3). Indeed, in the 2^{nd} version and in the next chapters, we have no easy way to get the address of the other functions. This function just calls a `tc_main` function, implemented at the very end. At this stage, in the 1^{st} version, we don't know `tc_main`'s address yet, and so we need to use a placeholder, filled at the end. Here we show the end result directly. In the 2^{nd} version we can just call `tc_main`, but we need to declare it first with "`fn tc_main;`":

```
    fn tc_main;
4612 fn main 3 (src_buffer, dst_buffer, flash_buffer)
  +2   get src_buffer:0 get dst_buffer:1 get flash_buffer:2 call 6673tc_ma≀
                                                              ≀in retv
```

We continue with functions to load and store bytes and half-words. We already have such functions, but we re-implement them here to avoid the need of function addresses in the 2^{nd} version (4294967040 = FFFFFF00$_{16}$, 4294901760 = FFFF0000$_{16}$):

```
4624 fn load8 1 (ptr)
  +2   get ptr:0 load cst8 255 and retv
4633 fn load16 1 (ptr)
  +2   get ptr:0 load cst 65535 and retv
4645 fn store8 2 (ptr, value)
  +2   get ptr:0 get ptr:0 load cst 4294967040 and get value:1 or store ret
4663 fn store16 2 (ptr, value)
  +2   get ptr:0 get ptr:0 load cst 4294901760 and get value:1 or store ret
```

We then implement the panic functions, using an auxiliary function to copy 16 bytes between two addresses (for the same reason as above, we don't reuse the already existing `mem_copy` function – see Table 13.1):

```
4681 fn panic_copy 2 (src, dst)
  +2   get dst:1 get src:0 load store
```

```
  +8   get dst:1 cst8 4 add get src:0 cst8 4 add load store
 +20   get dst:1 cst8 8 add get src:0 cst8 8 add load store
 +32   get dst:1 cst8 12 add get src:0 cst8 12 add load store
 +44   ret
4726 fn panic_result 1 (ptr)
  +2   ptr ptr:0 cst8 16 sub get ptr:0 call 4681panic_copy
 +12   cst A=1074666152 get ptr:0 store
 +20   cst_0 retv
4748 fn panic 1 (error)
  +2   cst A=1074666152 load ptr error:0 cst8 16 sub call 4681panic_copy
 +16   get error:0 retv
```

where A=1074666152 = $400E1AA8_{16}$ is the 7^{th} General Purpose Backup Register (we already used 3 in Section 11.4.2, and 3 more in Section 12.2).

16.3.1 Scanner

We start the scanner implementation with the `CHAR_TYPES` and `KEYWORDS` tables (the former has 256 values, one for each possible character; the latter omits the keywords that we don't need for now, such as `lsl` and `lsr`):

```
4767 static TC_CHAR_TYPES
       d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 32 d 32 d 1 d 1 d 1 d 1 d 1
       d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1
       d 32 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1
       d 2 d 2 d 2 d 2 d 2 d 2 d 2 d 2 d 2 d 2 d 58 d 59 d 1 d 1 d 1 d 1
       d 1 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3
       d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 1 d 1 d 1 d 1 d 3
       d 1 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3
       d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 3 d 1 d 1 d 1 d 1 d 1
       d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1
       d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1
       d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1
       d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1
       d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1
       d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1
       d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1
       d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1 d 1
5023 static TC_KEYWORDS
       d 5 d 'c'99 d 's'115 d 't'116 d '_'95 d '0'48 d 128
       d 5 d 'c'99 d 's'115 d 't'116 d '_'95 d '1'49 d 129
       d 4 d 'c'99 d 's'115 d 't'116 d '8'56 d 130
       d 3 d 'c'99 d 's'115 d 't'116 d 131
       d 3 d 'a'97 d 'd'100 d 'd'100 d 132
       d 3 d 's'115 d 'u'117 d 'b'98 d 133
       d 3 d 'm'109 d 'u'117 d 'l'108 d 134
       d 3 d 'd'100 d 'i'105 d 'v'118 d 135
       d 3 d 'a'97 d 'n'110 d 'd'100 d 136
```

```
d 2 d 'o'111 d 'r'114 d 137
d 4 d 'i'105 d 'f'102 d 'l'108 d 't'116 d 140
d 4 d 'i'105 d 'f'102 d 'e'101 d 'q'113 d 141
d 4 d 'i'105 d 'f'102 d 'g'103 d 't'116 d 142
d 4 d 'i'105 d 'f'102 d 'l'108 d 'e'101 d 143
d 4 d 'i'105 d 'f'102 d 'n'110 d 'e'101 d 144
d 4 d 'i'105 d 'f'102 d 'g'103 d 'e'101 d 145
d 4 d 'g'103 d 'o'111 d 't'116 d 'o'111 d 146
d 4 d 'l'108 d 'o'111 d 'a'97 d 'd'100 d 147
d 5 d 's'115 d 't'116 d 'o'111 d 'r'114 d 'e'101 d 148
d 3 d 'p'112 d 't'116 d 'r'114 d 149
d 3 d 'g'103 d 'e'101 d 't'116 d 150
d 3 d 's'115 d 'e'101 d 't'116 d 151
d 3 d 'p'112 d 'o'111 d 'p'112 d 152
d 4 d 'c'99 d 'a'97 d 'l'108 d 'l'108 d 154
d 3 d 'r'114 d 'e'101 d 't'116 d 157
d 4 d 'r'114 d 'e'101 d 't'116 d 'v'118 d 158
d 2 d 'f'102 d 'n'110 d 102
d 6 d 's'115 d 't'116 d 'a'97 d 't'116 d 'i'105 d 'c'99 d 115
d 0
```

We continue with the implementation of Algorithms 16.1 and 16.2:

```
5182 fn mem_compare 3 (ptr1, ptr2, size)
  +2   cst_0  → i
  +3 :step2
  +3   get i:7 get size:2 ifge 38step4
 +10   get ptr1:0 get i:7 add call 4624load8 get ptr2:1 get i:7 add call 4⸘
                                                    ⸘624load8 ifne 38step4
 +29   get i:7 cst_1 add set i:7 goto 3step2
 +38 :step4
 +38   get size:2 get i:7 sub retv

5226 fn tc_get_keyword 2 (start, length)
  +2   cst_0  → len
  +3   cst 791455TC_KEYWORDS  → ptr
  +8 :step2
  +8   get ptr:7 call 4624load8 set len:6
 +15   get len:6 cst_0 ifne 24step4
 +21   cst8 TC_IDENTIFIER=3 retv
 +24 :step4
 +24   get length:1 get len:6 ifne 57step7
 +31   get start:0 get ptr:7 cst_1 add get length:1 call 5182mem_compare c⸘
                                                        ⸘st_0 ifne 57step7
 +46   get ptr:7 get len:6 add cst_1 add call 4624load8 retv
 +57 :step7
 +57   get ptr:7 get len:6 add cst8 2 add set ptr:7 goto 8step2
```

To implement the next scanner functions we need to decide where to store the 7 scanner variables *src*, *src_end*, *next_char*, *next_char_type*, *next_token*,

$next\_token\_data$ and $next\_token\_length$. Passing them as function parameters would be cumbersome (because there are many variables, and because we would actually need to pass their addresses). Storing them at some fixed addresses, such as 537329664, 537329668, 537329672, etc is not better. Instead, we assume that these variables are stored next to each other, somewhere in RAM, and we pass the address of the first one, noted $self$, to each scanner function. Hence, src, src_end, $next\_char$, etc can be read or written by loading or writing the word at $self$, $self + 4$, $self + 8$, etc (respectively). With this hypothesis, Algorithm 16.3 can be implemented with the following function, which returns $next\_char\_type$ (or panics with $error = 10$):

```
5296 fn tc_read_char 1 (self)
  +2   get self:0 tc_src=cst_0 add load  → src
  +7   get self:0 cst8 tc_src_end=4 add load  → src_end
 +13   get src:5 get src_end:6 iflt 25step2
 +20   cst8 10 call 4748panic
 +25 :step2
 +25   get src:5 cst_1 add set src:5 cst_0  → char cst_0  → type
 +33   get src:5 get src_end:6 ifge 60end
 +40   get src:5 call 4624load8 set char:7
 +47   cst 791199TC_CHAR_TYPES get char:7 add call 4624load8 set type:8
 +60 :end
 +60   get self:0 tc_src=cst_0 add get src:5 store
 +67   get self:0 cst8 tc_next_char=8 add get char:7 store
 +75   get self:0 cst8 tc_next_char_type=12 add get type:8 store
 +83   get type:8 retv
```

We continue with the implementation of Algorithm 16.4 (as before, to simplify, we do not check if v fits in a word):

```
5382 fn tc_read_integer 1 (self)
  +2   get self:0 cst8 tc_next_char_type=12 add load  → type
  +8   cst_0  → v
  +9 :step2
  +9   get type:5 cst8 TC_INTEGER=2 ifne 43step5
 +16   get v:6 cst8 10 mul get self:0 cst8 tc_next_char=8 add load cst8 48↵
                                                          ↵ sub add set v:6
 +33   get self:0 call 5296tc_read_char set type:5
 +40   goto 9step2
 +43 :step5
 +43   get self:0 cst8 tc_next_token_data=20 add get v:6 store
 +51   cst8 TC_INTEGER=2 retv
```

and of Algorithm 16.5:

```
5436 fn tc_read_identifier 1 (self)
  +2   get self:0 tc_src=cst_0 add load  → start
  +7   get self:0 cst8 tc_next_char_type=12 add load  → type
 +13 :step2
 +13   get type:6 cst8 TC_IDENTIFIER=3 ifeq 27step3
```

```
+20   get type:6 cst8 TC_INTEGER=2 ifne 37step4
+27 :step3
+27   get self:0 call 5296tc_read_char set type:6 goto 13step2
+37 :step4
+37   get self:0 tc_src=cst_0 add load get start:5 sub  → length
+45   get self:0 cst8 tc_next_token_data=20 add get start:5 store
+53   get self:0 cst8 tc_next_token_length=24 add get length:7 store
+61   get start:5 get length:7 call 5226tc_get_keyword retv
```

With this we can finally implement Algorithm 16.6 in the main scanner function:

```
5505 fn tc_read_token 1 (self)
  +2   get self:0 cst8 tc_next_char_type=12 add load  → type
  +8 :step1
  +8   get type:5 cst8 32 ifne 25step3
 +15   get self:0 call 5296tc_read_char set type:5 goto 8step1
 +25 :step3
 +25   get type:5  → token
 +27   get type:5 cst8 TC_INTEGER=2 ifne 44step5
 +34   get self:0 call 5382tc_read_integer set token:6 goto 73end
 +44 :step5
 +44   get type:5 cst8 TC_IDENTIFIER=3 ifne 61step6
 +51   get self:0 call 5436tc_read_identifier set token:6 goto 73end
 +61 :step6
 +61   get type:5 cst_0 ifeq 73end
 +67   get self:0 call 5296tc_read_char pop
 +73 :end
 +73   get self:0 cst8 tc_next_token=16 add get token:6 store
 +81   ret
```

16.3.2 Backend

We start the backend implementation with a small function which increments the pointer at address ptr^p by $size$ and returns the *previous* pointer at this address. We use it later on to increment dst and $heap$:

```
5587 fn mem_allocate 2 (size, ptr_p)
  +2   get ptr_p:1 load  → ptr
  +5   get ptr_p:1 get ptr:6 get size:0 add store
 +13   get ptr:6 retv
```

To implement the backend itself we need to decide where to store the dst variable. We assume here that it is stored after the 7 scanner variables. Hence, dst can be read and written by loading or writing the word at $self + 28$. We can then implement 3 functions to write a byte, a half-word or a word at dst, respectively:

```
5603 fn tc_write8 2 (self, value)
  +2   cst_1 get self:0 cst8 tc_dst=28 add call 5587mem_allocate get value⸮
                                                              ⸮:1 call 4645store8
```

```
 +16   ret
5620 fn tc_write16 2 (self, value)
  +2   cst8 2 get self:0 cst8 tc_dst=28 add call 5587mem_allocate get valu⦚
                                                          ⦚e:1 call 4663store16
 +17   ret
5638 fn tc_write32 2 (self, value)
  +2   cst8 4 get self:0 cst8 tc_dst=28 add call 5587mem_allocate get valu⦚
                                                                ⦚e:1 store
 +15   ret
```

We continue the backend with functions to manage placeholders. Placeholders were introduced while describing the parser, but managing them involves writing and updating bytes in the *dst* buffer, which is the backend's responsibility. The following function adds a new placeholder for an unresolved symbol, with Algorithm 16.8. It takes the symbol's value address $value^p$ as parameter, and uses $y = dst$. Step 4 only returns $y - x$, and lets the caller fill the placeholder:

```
5654 fn tc_add_placeholder 2 (self, value_p)
  +2   get self:0 cst8 tc_dst=28 add load  → y
  +8   get value_p:1 load  → x
 +11   get value_p:1 get y:6 store
 +16   get x:7 cst_0 ifne 26step4 get y:6 set x:7
 +26 :step4
 +26   get y:6 get x:7 sub retv
```

The next function fills all the placeholders in the list starting at *placeholder* with *value*, using Algorithm 16.9 (step 1 must be done by the caller):

```
5686 fn tc_fill_placeholders 2 (placeholder, value)
  +2   cst_0  → offset
  +3 :step2
  +3   get placeholder:0 cst_0 ifeq 40end
  +9 :step3
  +9   get placeholder:0 call 4633load16 set offset:6
 +16   get placeholder:0 get value:1 call 4663store16
 +23   get offset:6 cst_0 ifne 30step6 ret
 +30 :step6
 +30   get placeholder:0 get offset:6 sub set placeholder:0
 +37   goto 3step2
 +40 :end
 +40   ret
```

We finish the backend with a small function to write a `fn` instruction and its argument, the function's number of parameters, also called its *arity*:

```
5727 fn tc_write_fn_insn 2 (self, arity)
  +2   get self:0 cst8 25 call 5603tc_write8
  +9   get self:0 get arity:1 call 5603tc_write8
 +16   ret
```

16.3.3 Parser

We start the parser implementation with a function to search for a symbol in the list of symbols starting with *symbol*, with Algorithm 16.7. We assume that the symbol's *name*, *length*, *kind*, *data*, and *next* symbol are stored at *symbol*, *symbol* + *4*, *symbol* + *8*, *symbol* + *12*, and *symbol* + *16*, respectively (as in Figure 16.1.):

```
5744 fn sym_lookup 3 (symbol, name, length)
  +2 :step2
  +2   get symbol:0 cst_0 ifeq 49step7
  +8   get symbol:0 cst8 sym_length=4 add load get length:2 ifne 38step6
 +19   get symbol:0 sym_name=cst_0 add load get name:1 get length:2 call 5≀
                                          ≀182mem_compare cst_0 ifne 38step6
 +35   get symbol:0 retv
 +38 :step6
 +38   get symbol:0 cst8 sym_next=16 add load set symbol:0 goto 2step2
 +49 :step7
 +49   cst_0 retv
```

We continue with a function to add a new symbol with the given name, length, kind and value in the *symbols* list. We assume here that the *heap* and *symbols* variables are stored after *dst*, *i.e.*, at *self* + *32* and *self* + *36*, respectively. We start by reserving 20 bytes to store the 5 words of the new symbol at *heap*:

```
5795 fn tc_add_symbol 5 (self, name, length, kind, value)
  +2   cst8 20 get self:0 cst8 tc_heap=32 add call 5587mem_allocate  → sym≀
                                                                  ≀ → bol
```

If the symbol is already in the *symbols* list, this is an error:

```
 +12   get self:0 cst8 tc_symbols=36 add load get name:1 get length:2 call≀
                                        ≀ 5744sym_lookup cst_0 ifeq 34ok
 +29   cst8 30 call 4748panic
```

otherwise we store the symbol's name, length, kind and value, and set its next symbol to *symbols*. Finally, we update *symbols* and return the new symbol:

```
 +34 :ok
 +34   get symbol:9 sym_name=cst_0 add get name:1 store
 +41   get symbol:9 cst8 sym_length=4 add get length:2 store
 +49   get symbol:9 cst8 sym_kind=8 add get kind:3 store
 +57   get symbol:9 cst8 sym_value=12 add get value:4 store
 +65   get symbol:9 cst8 sym_next=16 add get self:0 cst8 tc_symbols=36 add≀
                                                      ≀ load store
 +77   get self:0 cst8 tc_symbols=36 add get symbol:9 store
 +85   get symbol:9 retv
```

The following function adds a symbol with the given name, length and value to the *symbols* list, as a resolved symbol. Unless this symbol is already in the list, in which case it resolves it. We for this we first search this symbol. If it is not found, we add one and return it:

```
5883 fn tc_add_or_resolve_symbol 4 (self, name, length, value)
  +2   get self:0 cst8 tc_symbols=36 add load get name:1 get length:2 call≀
                                                  ≀ 5744sym_lookup  → symbol
 +15   get symbol:8 cst_0 ifne 34found
 +21   get self:0 get name:1 get length:2 SYM_RESOLVED=cst_0 get value:3 c≀
                                                 ≀all 5795tc_add_symbol retv
```

If the symbol already exists, then it should be unresolved (otherwise it means we are trying to define this symbol more than once). We thus check that this is the case, and panic otherwise:

```
 +34 :found
 +34   get symbol:8 cst8 sym_kind=8 add load SYM_UNRESOLVED=cst_1 ifeq 49o≀
                                                  ≀k cst8 31 call 4748panic
```

If everything is fine, we fill the symbol's placeholders with $value$, update its $kind$ to "resolved", update its value to $value$, and return it:

```
 +49 :ok
 +49   get symbol:8 cst8 sym_value=12 add load get value:3 call 5686tc_fil≀
                                                              ≀l_placeholders
 +60   get symbol:8 cst8 sym_kind=8 add SYM_RESOLVED=cst_0 store
 +67   get symbol:8 cst8 sym_value=12 add get value:3 store
 +75   get symbol:8 retv
```

To make it easier to implement the parsing functions corresponding to each grammar rule, we first provide 3 basic functions to parse a token, an integer or an identifier. The first one panics if the next token is not the one passed as parameter. Otherwise it just reads this token:

```
5961 fn tc_parse_token 2 (self, token)
  +2   get self:0 cst8 tc_next_token=16 add load get token:1 ifeq 18ok cst≀
                                                    ≀8 20 call 4748panic
 +18 :ok
 +18   get self:0 call 5505tc_read_token
 +23   ret
```

The second panics if the next token is not an integer. Otherwise it reads this token and returns its numeric value:

```
5985 fn tc_parse_integer 1 (self)
  +2   get self:0 cst8 tc_next_token=16 add load cst8 TC_INTEGER=2 ifeq 18≀
                                                  ≀ok cst8 21 call 4748panic
 +18 :ok
 +18   get self:0 cst8 tc_next_token_data=20 add load  → value
 +24   get self:0 call 5505tc_read_token
 +29   get value:5 retv
```

The last one panics if the next token is not an identifier. Otherwise it reads this token, sets its length at address $length^p$ and returns the address of its first character:

```
6017 fn tc_parse_identifier 2 (self, length_p)
  +2   get self:0 cst8 tc_next_token=16 add load cst8 TC_IDENTIFIER=3 ifeq⌇
                                                ⌇ 18ok cst8 22 call 4748panic
 +18 :ok
 +18   get self:0 cst8 tc_next_token_data=20 add load  → name
 +24   get length_p:1 get self:0 cst8 tc_next_token_length=24 add load sto⌇
                                                                       ⌇re
 +33   get self:0 call 5505tc_read_token
 +38   get name:6 retv
```

We can now implement the parsing functions, one per rule of the grammar. We implement them in reverse order, starting with the "static" rule. Parsing a `static` block starts by parsing this keyword and the following identifier. We then compute the final address of the following data, $dst - dst\_buffer + flash\_buffer$, and add a symbol for this identifier and value. Here we assume that $dst\_buffer - flash\_buffer$ is stored in a $flash\_offset$ variable at $self + 40$:

```
6058 fn tc_parse_static 1 (self)
  +2   get self:0 cst8 TC_STATIC=115 call 5961tc_parse_token
  +9   cst_0  → length
 +10   get self:0 ptr length:5 call 6017tc_parse_identifier  → name
 +17   get self:0 cst8 tc_dst=28 add load get self:0 cst8 tc_flash_offset=⌇
                                                ⌇40 add load sub  → value
 +30   get self:0 get name:6 get length:5 SYM_RESOLVED=cst_0 get value:7 c⌇
                                                ⌇all 5795tc_add_symbol pop
```

Finally, while the next token is an integer, we parse it and append the corresponding byte to the compiled code (to simplify we do not check if it fits in a byte):

```
 +43 :loop
 +43   get self:0 cst8 tc_next_token=16 add load cst8 TC_INTEGER=2 ifne 67⌇
                                                                     ⌇end
 +54   get self:0 get self:0 call 5985tc_parse_integer call 5603tc_write8
 +64   goto 43loop
 +67 :end
 +67   ret
```

The next function implements the "argument" rule. If the next token is an integer we just parse it and return its numeric value:

```
6126 fn tc_parse_argument 1 (self)
  +2   get self:0 cst8 tc_next_token=16 add load cst8 TC_INTEGER=2 ifne 19⌇
                    ⌇identifier get self:0 call 5985tc_parse_integer retv
```

Otherwise the next token should be an identifier. We get its name and length with `tc_parse_identifier`:

```
 +19 :identifier
 +19   cst_0  → length
 +20   get self:0 ptr length:5 call 6017tc_parse_identifier  → name
```

and then search it in the symbol table:

```
+27   get self:0 cst8 tc_symbols=36 add load get name:6 get length:5 call⌇
                                                  ⌇ 5744sym_lookup  → symbol
```

If no symbol is found we add an unresolved one for this identifier:

```
+40   get symbol:7 cst_0 ifne 59found
+46   get self:0 get name:6 get length:5 SYM_UNRESOLVED=cst_1 cst_0 call ⌇
                                          ⌇5795tc_add_symbol set symbol:7
```

Otherwise we jump here directly. In both cases we now have a symbol for the identifier, but it might be unresolved. If it is, we add a new placeholder and return the value to store in it:

```
+59 :found
+59   get symbol:7 cst8 sym_kind=8 add load SYM_UNRESOLVED=cst_1 ifne 80r⌇
                                                              ⌇esolved
+69   get self:0 get symbol:7 cst8 sym_value=12 add call 5654tc_add_place⌇
                                                    ⌇holder retv
```

Finally, if it is resolved, we just return its value:

```
+80 :resolved
+80   get symbol:7 cst8 sym_value=12 add load retv
```

Parsing a label starts by parsing a colon (58 in ASCII) and then an identifier:

```
6213 fn tc_parse_label 1 (self)
  +2   get self:0 cst8 58 call 5961tc_parse_token
  +9   cst_0  → length
 +10   get self:0 ptr length:5 call 6017tc_parse_identifier  → name
```

We then want to compute the instruction offset corresponding to this label. This is dst minus the address of the current function's first instruction, noted fn_dst. This address can be computed in the `parse_fn` function. We assume here that it is stored after tc_flash_offset, *i.e.*, at $self + 44$:

```
+17   get self:0 cst8 tc_dst=28 add load get self:0 cst8 tc_fn_dst=44 add⌇
                                                  ⌇ load sub  → value
```

Finally, we add a symbol for this label or resolve it, and return:

```
+30   get self:0 get name:6 get length:5 get value:7 call 5883tc_add_or_r⌇
                                                    ⌇esolve_symbol ret
```

To implement `parse_instruction` it is useful to have in a table – noted `S` in the previous chapter – the argument size of each opcode (in bytes):

```
6255 static ARG_SIZES
       d 0 d 0 d 1 d 4 d 0 d 0 d 0 d 0 d 0 d 0 d 0 d 0 d 2 d 2 d 2 d 2
       d 2 d 2 d 2 d 0 d 0 d 1 d 1 d 1 d 0 d 0 d 2 d 0 d 0 d 0 d 0 d 0
```

If the next token is a colon (58 in ASCII) we just call `parse_label` and return:

```
6287 fn tc_parse_instruction 1 (self)
  +2   get self:0 cst8 tc_next_token=16 add load  → token
  +8   get token:5 cst8 58 ifne 21not_label
 +15   get self:0 call 6213tc_parse_label
 +20   ret
```

Otherwise the next token should be an opcode keyword, and we assume here that the caller has already checked this. We then compute the corresponding opcode ($token - 128$ by construction), write it, and read the token:

```
 +21 :not_label
 +21   get token:5 cst8 128 sub  → opcode
 +26   get self:0 get opcode:6 call 5603tc_write8
 +33   get self:0 call 5505tc_read_token
```

Then there are 4 cases, depending on the argument size (computed with `ARG_SIZES`). If it is 0 we have nothing to do and thus return directly:

```
 +38   cst 792687ARG_SIZES get opcode:6 add call 4624load8  → arg_size
 +49   get arg_size:7 cst_0 ifne 56not0
 +55   ret
```

Otherwise we need to parse the argument, and then write it in 1, 2 or 4 bytes:

```
 +56 :not0
 +56   get self:0 call 6126tc_parse_argument  → arg
 +61   get arg_size:7 cst_1 ifne 75not1
 +67   get self:0 get arg:8 call 5603tc_write8 ret
 +75 :not1
 +75   get arg_size:7 cst8 2 ifne 90not2
 +82   get self:0 get arg:8 call 5620tc_write16 ret
 +90 :not2
 +90   get self:0 get arg:8 call 5638tc_write32 ret
```

Parsing a function name starts by parsing an identifier, and by storing its address (dst) in fn_dst, as we assumed above. We then compute the value of this identifier, *i.e.*, the argument which much be used in a `call` instruction to call this function. This is its final address in flash memory, $dst - dst\_buffer + flash\_buffer$, minus $\mathrm{C0000}_{16}$.

```
6385 fn tc_parse_fn_name 1 (self)
  +2   cst_0  → length
  +3   get self:0 ptr length:5 call 6017tc_parse_identifier  → name
 +10   get self:0 cst8 tc_dst=28 add load  → fn_dst
 +16   get self:0 cst8 tc_fn_dst=44 add get fn_dst:7 store
 +24   get fn_dst:7 get self:0 cst8 tc_flash_offset=40 add load sub cst 78≀
                                                          ≀6432 sub  → value
```

Finally, we add or resolve a symbol with this value, and return it:

```
 +39   get self:0 get name:6 get length:5 get value:8 call 5883tc_add_or_r≀
                                                        ≀esolve_symbol retv
```

Parsing the body of a function is done as explained in the previous section. The following function takes as parameter the symbol computed by `parse_fn_name`. If the next token is a semi-colon (59 in ASCII), it reads this token and changes this symbol to "unresolved", with an empty list of placeholders:

```
6436 fn tc_parse_fn_body 2 (self, function)
  +2   get self:0 cst8 tc_next_token=16 add load cst8 59 ifne 33body
 +13   get self:0 call 5505tc_read_token
 +18   get function:1 cst8 sym_kind=8 add SYM_UNRESOLVED=cst_1 store
 +25   get function:1 cst8 sym_value=12 add cst_0 store
 +32   ret
```

Otherwise it parses an integer (the function's arity) and writes it as an argument of an `fn` instruction:

```
 +33 :body
 +33   get self:0 get self:0 call 5985tc_parse_integer call 5727tc_write_f⟅
                                                                    ⟅n_insn
```

Finally, it parses an instruction while the next token is a colon or an opcode (*i.e.*, while $next\_token = 58$ or $next\_token \geq 128$):

```
 +43 :loop
 +43   get self:0 cst8 tc_next_token=16 add load cst8 58 ifeq 66insn
 +54   get self:0 cst8 tc_next_token=16 add load cst8 128 ifge 66insn
 +65   ret
 +66 :insn
 +66   get self:0 call 6287tc_parse_instruction goto 43loop
```

A jump instruction can reference a label which is not yet defined, but this label must be defined before the end of the function. We thus need to verify, when we reach the end of the function, that all its label symbols are resolved. To this end, we implement the following function, which checks that all the symbols from $symbol$ to end_symbol (excluded) are resolved (and panics otherwise):

```
6510 fn tc_check_symbols 2 (symbol, end_symbol)
  +2 :loop
  +2   get symbol:0 get end_symbol:1 ifeq 35end
  +9   get symbol:0 cst8 sym_kind=8 add load SYM_UNRESOLVED=cst_1 ifne 24n⟅
                                                                    ⟅ext
 +19   cst8 32 call 4748panic
 +24 :next
 +24   get symbol:0 cst8 sym_next=16 add load set symbol:0 goto 2loop
 +35 :end
 +35   ret
```

With this we can now implement the `parse_fn` function. It simply needs to parse the `fn` token, and then call the above functions to parse the function name and body, and to check that all the labels are resolved. As explained in the previous section, this function saves the $heap$ and $symbols$ variables before parsing the body, and

restores their values once this is done. Note that we only check the symbols added in `tc_parse_fn_body`. Indeed, it is not an error if a symbol added before that is undefined (it corresponds to a forward function reference).

```
6546 fn tc_parse_fn 1 (self)
  +2   get self:0 cst8 TC_FN=102 call 5961tc_parse_token
  +9   get self:0 call 6385tc_parse_fn_name  → function
 +14   get self:0 cst8 tc_heap=32 add load  → heap
 +20   get self:0 cst8 tc_symbols=36 add load  → symbols
 +26   get self:0 get function:5 call 6436tc_parse_fn_body
 +33   get self:0 cst8 tc_symbols=36 add load get symbols:7 call 6510tc_ch⸘
                                                              ⸘eck_symbols
 +44   get self:0 cst8 tc_symbols=36 add get symbols:7 store
 +52   get self:0 cst8 tc_heap=32 add get heap:6 store
 +60   ret
```

We can finally implement the last parsing function, corresponding to the main grammar rule. For this we loop while the next token is `fn` or `static` and, if so, call the corresponding parsing function:

```
6607 fn tc_parse_program 1 (self)
  +2 :loop
  +2   get self:0 cst8 tc_next_token=16 add load cst8 TC_FN=102 ifne 21not⸘
                                                                  ⸘_fn
 +13   get self:0 call 6546tc_parse_fn goto 2loop
 +21 :not_fn
 +21   get self:0 cst8 tc_next_token=16 add load cst8 TC_STATIC=115 ifne 4⸘
                                                                ⸘0end
 +32   get self:0 call 6058tc_parse_static goto 2loop
```

We then test if the next token is the special END token (0), and panic otherwise. Finally, we check that all the symbols are resolved, *i.e.*, that all the functions declared with the `fn` *name*`;` syntax are effectively implemented:

```
 +40 :end
 +40   get self:0 cst8 tc_next_token=16 add load cst_0 ifeq 55ok
 +50   cst8 23 call 4748panic
 +55 :ok
 +55   get self:0 cst8 tc_symbols=36 add load cst_0 call 6510tc_check_symb⸘
                                                             ⸘ols ret
```

The last function of our compiler is the `tc_main` function which was declared at the very beginning. This function initializes the compiler variables, sets a panic handler, and finally calls `tc_parse_program`. We start by initializing the variables defined above *on the stack*. Since the stack grows in decreasing address order, we define them in reverse order, *i.e.*, from fn_dst to src. fn_dst can be set to any value (it is reset in `parse_fn`). $flash\_offset$ is equal to $flash\_buffer - dst\_buffer$. $symbols$ must be initialized to 0 (an empty list). We set $heap$ 12 KB after dst_buffer, which leaves more than enough space for the compiled code (the current compiled code size is about 2 KB). dst starts after the 4 bytes buffer header. The next scanner variables can be set to any value (see below):

```
6673 fn tc_main 3 (src_buffer, dst_buffer, flash_buffer)
  +2   cst_0  → fn_dst
  +3   get dst_buffer:1 get flash_buffer:2 sub  → flash_offset
  +8   cst_0  → symbols
  +9   get dst_buffer:1 cst 12288 add  → heap
 +17   get dst_buffer:1 cst8 4 add  → dst
 +22   cst_0  → next_token_length
 +23   cst_0  → next_token_data
 +24   cst_0  → next_token
 +25   cst_0  → next_char_type
 +26   cst_0  → next_char
```

We then initialize src_end to the end of the source code, $src\_buffer + 4 + \mathrm{mem32}[src\_buffer]$, and src to the start of the source code *minus* 1 (see below):

```
 +27   get src_buffer:0 cst8 4 add get src_buffer:0 load add  → src_end
 +36   get src_buffer:0 cst8 3 add  → src
```

We continue with a call to `panic_result` so that calls to `panic` directly return here, just after the call instruction. As explained in the previous section, this function saves the 4 values necessary for this at some ptr address. Here we push 4 zeros on the stack and use the address of the top one as ptr:

```
 +41   cst_0  → panic3
 +42   cst_0  → panic2
 +43   cst_0  → panic1
 +44   cst_0  → panic0
 +45   cst_0  → error
 +46   ptr panic0:22 call 4726panic_result set error:23
```

This first call returns 0 but, in case of panic, `panic_result` returns again with a non-zero value (stored in $error$ just above). Thus, if $error \neq 0$, we store the location of this error in the source code, $src - src\_buffer - 4$ at the dst_buffer address, and return the error itself (as per the compiler requirements):

```
 +53   get error:23 cst_0 ifeq 73ok
 +59   get dst_buffer:1 get src:18 get src_buffer:0 sub cst8 4 sub store
 +70   get error:23 retv
```

We finally call `tc_parse_program`, set the compiled code size $dst - dst\_buffer - 4$ in the dst_buffer header, and return 0 (meaning "no error"). Before this, since all parsing functions, and `tc_parse_program` in particular, assume that information about the next token is already available in $next\_token$, we call `tc_read_token`. And since this function assumes that information about the next character is already available in $next\_char$, we call `tc_read_char` first (which explains why src was set to 1 byte before the start of the source code, and why the other scanner variables could be initialized to any value).

```
 +73 :ok
 +73   ptr src:18 call 5296tc_read_char pop
```

```
 +79   ptr src:18 call 5505tc_read_token
 +84   ptr src:18 call 6607tc_parse_program
 +89   get dst_buffer:1 get dst:11 get dst_buffer:1 sub cst8 4 sub store
+100   cst_0 retv
```

16.4 Compilation and tests

To compile the above source code proceed as follows (see also Figure 16.4). First launch the command editor by typing "w000c1172"+Enter in the memory editor, followed by "r".

Edit v1 Type "F2"+"r" to create a new program, and "F4"+"r" to edit it. Then type the 1^{st} version of the labels compiler (that is, the above code with the parts highlighted in red). For convenience, we also provide this code in the `labels_compiler_v1.txt` file in https://ebruneton.github.io/toypc/sources.zip. When you are done, type Escape to exit the text editor and "F5"+"r" to save your work. Alternatively, you can "cheat" by saving it via an external computer, as follows. First run the `boot_mode_select_rom` function by typing "w000c02b4"+Enter in the memory editor, followed by "r". Then reset the Arduino and, on the host computer, run the following command (then restart the command editor and type "F3"+"r"):

```
user@host:~$ python3 flash_helper.py < part3/labels_compiler_v1.txt
```

Compile v1 In the command editor, type "F6"+"r" to compile the code you typed. If all goes well, after about 1 second, you should get a result equal to 0, meaning that no error was found. If this is not the case, type "F4"+"r" to fix the error. The text editor should open right at the error location. Fix the error indicated by the error code returned by the compiler (see Appendix D), save the program and compile it again. Repeat this process until the compilation is successful. Then type "F7"+"r" to save the result.

Test v1 Type "F9"+"e" to edit a new command, and type the following code:

```
fn 0
  cst 537330176 cst 537379328 cst 803328 call 4612
  get 4 cst_0 ifne 47
  cst 537379328 cst 66 call 2836
  cst 803332 calld set 4
  get 4 retv
d TEST_COMPILER
```

Then type Escape and "s" to save it. The first line calls the labels compiler with src_buffer=20070200_{16}, dst_buffer=$2007C200_{16}$, and $flash\_buffer$=$C4200_{16}$ (12 KB after "saved program" – see Figure 15.3). If there is a compilation error the second line returns it by jumping to the fifth line. Otherwise the third line calls buffer_flash to save the compiled program in $flash\_buffer$, which corresponds to page 66. The last lines call this compiled program and return its result.

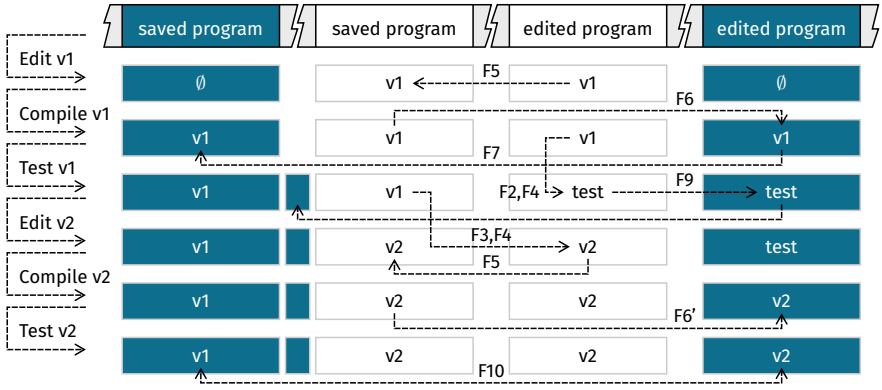


FIGURE 16.4 The memory content after each step of Section 16.4. White, blue and gray areas represent source code, bytecode and unused memory, respectively (not to scale). See also Figure 15.3.

Type "F2"+"r" to create a new program, "F4"+"r" to edit it, and type the following small test program, which computes the factorial of 6:

```
fn factorial;
fn test 0
  cst 6 call factorial retv
fn factorial 1
  get 0 cst_0 ifne not_zero cst_1 retv
  :not_zero get 0 cst_1 sub call factorial get 0 mul retv
```

Then type "F9"+"r" to compile and run it. The result should be 720 = $2D0_{16}$. If this is not the case this means that the compiler is wrong. Repeat the previous steps and double check everything until this test passes.

Edit v2 Type "F3"+"r" to load the 1^{st} version of the labels compiler and "F4"+"r" to edit it. Then update it to the 2^{nd} version (that is, the code in the previous section, with the parts highlighted in green). For convenience, we also provide this code in the `labels_compiler_v2.txt` file. Then save this new version with the F5 command. Alternatively, run the following command on an external computer (see above):

```
user@host:~$ python3 flash_helper.py < part3/labels_compiler_v2.txt
```

Compile v2 To compile this new code we need to update command number 6 first, in order to use the labels compiler instead of the opcodes compiler. Type "F6"+"e" to edit this command and change its code to the following:

```
fn 0
  cst 856064 cst 537379328 cst 791040 call 4612 retv
d COMPILE_SOURCE_CODE
```

Then type "s" to save it, and "r" to run it. The result should be 0, meaning "no error". If this is not the case, repeat the "Edit v2" and "Compile v2" steps until all errors are fixed, as explained in "Compile v1".

Test v2 The compilation of the 2^{nd} version of the labels compiler should give the same function addresses and instruction offsets as those we computed manually to write the 1^{st} version. Consequently, the compiled code of the 2^{nd} version (at address $2007C200_{16}$) should be identical to that of the 1^{st} version (at address $C1200_{16}$). To check this we can use the following function, which compares the buffers at these two addresses with Algorithm 16.1:

`compare_compiled_code()`	**fn**	0		00000
Step 1. Initialize *ptr1* and *ptr2* to the above addresses.	**cst**	537379328	→ *ptr1*	+002
	cst	791040	→ *ptr2*	+007
Initialize *size* to the size of the *ptr1* buffer, which is the value at this address plus 4.	**get**	4	*ptr1*	+012
	load			+014
	cst8	4		+015
	add		→ *size*	+017
Initialize i to 0.	**cst_0**		→ i	+018
Step 2. If i is not less than *size*, go to step 4.	**get**	7	i	+019
	get	6	*size*	+021
	ifge	50		+023
If the two bytes at $ptr1+i$ and $ptr2+i$ are different, go to step 4.	**get**	4	*ptr1*	+026
	get	7	i	+028
	add			+030
	call	960	`load_byte`	+031
	get	5	*ptr2*	+034
	get	7	i	+036
	add			+038
	call	960	`load_byte`	+039
	ifne	50		+042
Step 3. Increment the top stack value i by 1 and go back to step 2.	**cst_1**			+045
	add			+046
	goto	19		+047
Step 4. Return $size - i$.	**get**	6	*size*	+050
	get	7	i	+052
	sub			+054
	retv			+055

Type "F10"+"e" to edit a new command, and type the source code of the above function:

```
fn 0
  cst 537379328 cst 791040
  get 4 load cst8 4 add
  cst_0
  get 7 get 6 ifge 50
  get 4 get 7 add call 960 get 5 get 7 add call 960 ifne 50
  cst_1 add goto 19
```

```
  get 6 get 7 sub retv
d COMPARE_COMPILED_CODE
```

Then type Escape and "s" to save it. Finally, type "r" to run it. The result should be 0, indicating that the two compiler versions have the same compiled code. If this is not the case, repeat the steps from "Edit v2" until this test passes. It might also happen that the 1^{st} version is wrong despite the "Test v1" step. In this case you need to restore the "F6" command to its previous value and restart from the "Edit v1" step.

CHAPTER

17 Expressions Compiler

The labels compiler written in the previous chapter removes the need to manually compute function addresses and instruction offsets, which is a huge improvement. However, it still requires us to use numbers to refer to function arguments or values on the stack. This is easier to do than to use function addresses and instruction offsets, but it would be better if we could avoid this. Another issue is that computing simple expressions such as $2 + 3 * 4$ requires a "lot" of code, in an unnatural order (`cst8 2 cst8 3 cst8 4 mul add`). It would be better if we could just type `2+3*4` instead. This chapter extends our toy programming language and its compiler in order to solve these issues.

17.1 Requirements

17.1.1 Function parameters

So far we used comments to give a symbolic name to each function parameter. We also used these symolic names in comments next to each `get`, `set` or `ptr` instruction, in order to make them easier to understand. To avoid using numbers in these instructions, the same solution as in the previous chapter can be used: we can use symbolic names directly in the source code, instead of in comments. The compiler can then keep track of the value of these symbols (*i.e.*, their index in the stack frame), like it does for function names and labels. To this end, we now require function parameters to be declared after the function name, between parentheses and separated by commas, as in the following example:

```
fn gpu_set_color(red, green, blue) ...
```

We could then use these names in instructions such as `get red`, `set green` or `ptr blue` (instead of writing `get 0`, `set 1` or `ptr 2`). In fact, to get shorter programs we simply use "`red`" instead of `get red` and "`&blue`" instead of `ptr blue` (see below).

17.1.2 Expressions

A well-formed series of arithmetic and logic instructions (see Section 8.2.1) computes a single value on the stack which can be written in a shorter mathematical form. For instance, as noted above, `cst8 2 cst8 3 cst8 4 mul add` computes `2+3*4`, which is

much shorter to write and is called an *expression*. For this reason, we now require our programming language to support expressions. This means that it should be possible to write `2+3*4` in a program, for instance, and that the compiler should automatically compile this into `cst8 2 cst8 3 cst8 4 mul add` (in binary form).

More precisely, our programming language should support the following expressions (where e_i is an expression and `code[`e_i`]` the corresponding compiled code, x is a symbolic name corresponding to the i^{th} stack frame slot, and f is the name of a function whose address is $a + C0000_{16}$):

- integer constants: compiled to `cst_0`, `cst_1`, `cst8`, or `cst`, depending on the value.
- e_1 `+` e_2: compiled to `code[`e_1`] code[`e_2`] add`.
- e_1 `-` e_2: compiled to `code[`e_1`] code[`e_2`] sub`.
- e_1 `*` e_2: compiled to `code[`e_1`] code[`e_2`] mul`.
- e_1 `/` e_2: compiled to `code[`e_1`] code[`e_2`] div`.
- e_1 `&` e_2: compiled to `code[`e_1`] code[`e_2`] and`.
- e_1 `|` e_2: compiled to `code[`e_1`] code[`e_2`] or`.
- `*`e_1: compiled to `code[`e_1`] load`.
- `&`x: compiled to `ptr` i.
- x: compiled to `get` i.
- f`(`e_1`, `e_2`, ...)`: compiled to `code[`e_1`] code[`e_2`] ... call` a.

These expressions correspond to all the bytecode instructions which produce a value on the stack, except the ones we don't need for now (namely `lsl`, `lsr`, `callr`, and `calld`). The `call` instruction is a special case: the callee might not return a value. However, functions used in subexpressions *must* return a value.

17.1.3 Local variables

The result of most expressions is immediately consumed in other expressions or instructions. Some results, however, are left on the stack and used later on with `get`, `set` or `ptr` instructions. This currently requires keeping track of which value is stored in which stack frame slot. To avoid this, our programming language should provide a way to give a symbolic name to an expression whose value is stored on the stack. In this chapter we use the "`let` x `;`" syntax, where x is an identifier, and e an expression. x can then be used in other expressions, such as "`'`x`+1`" or "`&`x" (unlike labels, x must be *declared* with `let` before it can be used). It is called a *local variable* because it can only be used in the current function (since `get`, `set` and `ptr` can only refer to slots in the top stack frame). This does not prevent another function to declare a variable with the same name, but it is then independent (*i.e.*, refers to a different slot).

17.1.4 Grammar

We can now extend the grammar of our programming language in order to support the above requirements. Lets look at expressions first. The above definitions might suggest a grammar rule of the following form:

expr: expr (“+” | “-” | “*” | “/” | ...) expr | “*” expr | “&” IDENTIFIER | ...

meaning that an expression is either a *binary* expression made of two subexpressions with an operator in between, or an *unary* expression with an operator followed by an expression, etc. However, this rule has several issues:

- It is ambiguous. Consider for instance the $2 + 3 * 4$ expression. It can be seen as a binary $+$ expression with subexpressions 2 and $3 * 4$. But it can also be seen as a binary $*$ expression with subexpressions $2 + 3$ and 4. Both interpretations are valid for the above grammar, but they don't give the same value! In practice anyone would give the value 14 because we use implicit *operator precedence* rules. One such rule is that multiplications have a higher precedence than additions, meaning that they must be performed first. When we want to use the other interpretation we use parentheses, which have the highest precedence: $(2 + 3) * 4$.
  Another ambiguity of the above rule is that $2 - 3 - 4$ can be seen as “$2 - 3$” minus 4, or as 2 minus “$3 - 4$”. Here again, both interpretations are valid for the grammar (and for the precedence rules), but they don't give the same value. In practice one always uses the first interpretation, because we use another implicit rule saying that operations are done from left to right.
- It can not be implemented with a recursive descent parser. This is because the “expr” rule is used on the leftmost position in the definition of one of its alternatives (such grammars are called *left recursive*). This would give a `parse_expr` function which would call itself recursively without reading any token in between, *i.e.*, indefinitely (if the stack was unbounded).

To solve these issues, a solution is to use several rules, one per precedence level. For instance, considering only the four basic operations for now, we can use

expr: term ((“+” | “-”) term)*
term: factor ((“*”| “/”) factor)*
factor: INTEGER | “(” expr “)”

where precedence increases from top to bottom. Indeed, with these rules, $2 + 3 * 4$ can be interpreted in only one way, the one we are used to (because $2 + 3$ is not a factor). Similarly, $2 - 3 - 4$ can only be seen as an expression with an unambiguously ordered list of 3 terms. This does not tell in itself whether these terms must be evaluated from left to right or right to left, but this choice can be enforced in the compiler implementation. Finally, these rules can be implemented with a recursive descent parser (“expr” indirectly uses itself recursively, but only after the “(” token; hence there is no left recursion).

The following grammar applies this idea to all the expressions in Section 17.1.2, and takes into account the requirements in Sections 17.1.1 and 17.1.3. It does this by extending the previous grammar as follows (unchanged parts are in gray):

program: (fn | static | const)* END
fn: “fn” fn_name fn_parameters fn_body

fn_name: IDENTIFIER
fn_parameters: "(" (IDENTIFIER ("," IDENTIFIER)*)? ")"
fn_body: "{" statement* "}" | ";"
statement: label | let_stmt | (expr ("," expr)*)? instruction? ";"
let_stmt: "`let`" IDENTIFIER expr ";"
instruction: "`iflt`" argument | "`ifeq`" argument | ... | "`store`" | "`pop`" | ...
expr: bit_and_expr ("|" bit_and_expr)*
bit_and_expr: add_expr ("&" add_expr)*
add_expr: mult_expr (("+" | "-") mult_expr)*
mult_expr: pointer_expr (("*" | "/") pointer_expr)*
pointer_expr: "*" pointer_expr | "&" IDENTIFIER | primitive_expr
primitive_expr: INTEGER | IDENTIFIER fn_arguments? | "(" expr ")"
fn_arguments: "(" (expr ("," expr)*)? ")"
label: ":" IDENTIFIER
argument: IDENTIFIER
static: "`static`" IDENTIFIER INTEGER*
const: "`const`" IDENTIFIER INTEGER

where "?" denotes an optional element. Thus, for instance, the "fn_parameters" rule means "a left parenthesis, optionally followed by an a non-empty list of parameters, followed by a right parenthesis" (the non-empty list of parameters being defined as an identifier, followed by any number of "comma identifier" groups).

The body of a function is now defined as a list of *statements* between curly braces (added to more clearly separate functions from each other, but also to simplify the parser). Each statement is either a label, a local variable declaration, or a comma separated list of expressions[1] followed by an optional instruction and ending with a semi-colon. Examples of the latter case include "`ret;`", "`0 set x;`", "`x, y ifeq ok;`" (an instruction preceded by 0, 1 or 2 expressions, respectively), or "`panic(1);`" (an expression not followed by any instruction).

Instructions are defined as in the previous chapter, except that all the instructions listed in Section 17.1.2 (`add`, `sub`, etc) are now removed (expressions must be instead). Similarly, instruction arguments can no longer be integers: label, function parameter or local variable names must be used instead.

Expressions are defined as explained above, with 6 levels of precedence. Constants, identifiers, function calls, parentheses, and the *address-of* operator "&" have the same highest precedence[2]. They are followed by the *dereference operator* "`*`" which loads the value at some address. Then comes multiplicative expressions, additive expressions, bitwise and expressions, and finally bitwise or expressions. Thus, for instance, "`a + *b * c & d`" is equivalent to "`(a + ((*b) * c)) & d`" and not to

[1] It is possible to use more strict rules to enforce a precise number of expressions before each instruction (*e.g.*, 0 before `ret`, 1 before `set`, or 2 before `ifeq`). We use this less strict rule to simplify the grammar, and thus the implementation.

[2] We put the address-of operator in the pointer_expr rule for convenience, but it could be moved in the primitive_expr rule instead, thus showing that it has the same precedence as the others.

"`a + ((*b) * (c & d))`" or "`a + (*(b * c) & d)`", for instance. However, when in doubt, it is preferable to use explicit parentheses.

Finally, the new "const" rule adds a syntax to give a symbolic name to a constant value. For instance, "`const RESOLVED 0`" makes it possible to use "`RESOLVED`" in an expression, which is more meaningful than using "`0`". To simplify the implementation, a constant must be defined before it is used.

17.1.5 Scanner

As a last new requirement, our programming language should support INTEGER tokens of the form "'c'", where c is a printable character (ASCII code between 32 and 127, excluded), and whose numeric value is the ASCII code of c. For instance, it should be possible to write `'a'` instead of a's ASCII code `97`, or `''''` instead of the quote's ASCII code `39`.

17.2 Algorithms

Compiling function parameters, local variable names and `const` declarations can be done with the same algorithms used for function names and labels. Namely those to add a symbol in the list of symbols, and to use this list to find the value of a symbol. Similarly, compiling expressions can be done with the same recursive descent method already used, namely with one function per grammar rule. Hence we do not really need any new algorithm in this chapter. Instead, we list here the main implementation differences compared with the previous compiler version.

The scanner needs an updated `CHAR_TYPES` table to support the new single character tokens (parentheses, curly braces, +, -, *, /, &, | and ,). It also needs an updated `KEYWORDS` table (`let` and `const` are added, `add`, `sub`, etc are removed). Finally, a new function is needed to read the new quoted character tokens defined above.

The backend should provide new functions to write the opcode instructions needed to compile expressions (`cst_0`, ... `add`, ...). The goal is to simplify the parser by *encapsulating* the low level instruction encoding details in simple to use functions.

The compiler needs to keep track of the stack frame slot index corresponding to each function parameter and local variable. The former is easy: the i^{th} parameter is in the i^{th} slot. For the latter we assume that `let` statements are the only ones which leave a value on the stack (this forbids, for instance, statements such as "1;"). We also assume, without verification, that `let` statements are executed in the same order as in the source code, and exactly once (unless the function returns before). Then the i^{th} local variable is in the $(i + 4)^{th}$ slot (recall that 4 saved register values are pushed after the function arguments).

Compiling a `const` c v declaration does not need to produce any code. Instead, we can simply add a corresponding symbol in the symbols list. Then, each time c is used, a `cst*` instruction to push v on the stack can be produced. Note however that, in order to do this, the parser must know that c refers to a constant. Indeed, if c is

referring to a local variable or function argument, a `get` instruction must be produced instead. To this end, we introduce a new symbol kind, `VARIABLE` (2), in addition to the `RESOLVED` and `UNRESOLVED` kinds. And we use `RESOLVED` for `const` symbols and `VARIABLE` for local variables and function arguments.

17.3 Implementation

We can now extend the labels compiler in order to support expressions. We first need to write it without using expressions, so that it can be compiled with the labels compiler. We then compile this source code, which gives us the expressions compiler bytecode. Finally, we rewrite this source code with expressions, and we compile it with the expressions compiler bytecode. As before, to save space, we give the two compiler versions at the same time (without expressions in red, with in green).

The start of the compiler does not change in the 1^{st} version (changes are indicated with a vertical bar in the margin), but can be rewritten in a clearer way in the 2^{nd}:

```
fn tc_main(src_buffer, dst_buffer, flash_buffer);
fn main 3(src_buffer, dst_buffer, flash_buffer) {
  get 0 get 1 get 2 call tc_main(src_buffer, dst_buffer, flash_buffer) ret
                                                                      v;
}

fn load8 1(ptr) { get 0 load cst8 255 and(*ptr) & 255 retv; }
fn load16 1(ptr) { get 0 load cst 65535 and(*ptr) & 65535 retv; }
fn store8 2(ptr, value) { get 0ptr, get 0 load cst 4294967040 and get 1 or
                                  (*ptr) & 4294967040 | value store; ret; }
fn store16 2(ptr, value) { get 0ptr, get 0 load cst 4294901760 and get 1 o
                                 r(*ptr) & 4294901760 | value store; ret; }

const PANIC_BUFFER 1074666152
fn panic_copy 2(src, dst) {
  get 1dst, get 0 load*src store;
  get 1 cst8 4 add(dst + 4), get 0 cst8 4 add load*(src + 4) store;
  get 1 cst8 8 add(dst + 8), get 0 cst8 8 add load*(src + 8) store;
  get 1 cst8 12 add(dst + 12), get 0 cst8 12 add load*(src + 12) store;
  ret;
}
fn panic_result 1(ptr) {
  ptr 0 cst8 16 sub get 0 call panic_copy(&ptr - 16, ptr);
  cst 1074666152PANIC_BUFFER, get 0ptr store;
  cst_0 retv;
}
fn panic 1(error) {
  cst 1074666152 load ptr 0 cst8 16 sub call panic_copy(*PANIC_BUFFER, &er
                                                              ror - 16);
  get 0error retv;
}
```

17.3.1 Shared constants

We then declare, in the 2^{nd} version, a set of constants for $token$ values, symbol's $kind$ values, for the offset of each symbol's variable (and their total size), and for the offset of the compiler variables from $self$. For convenience, we set the token values of +, -, *, /, &, and | to their corresponding opcode (4, 5, 6, 7, 8, and 9, respectively).

```
const TC_INTEGER 2
const TC_IDENTIFIER 3
const TC_ADD 4
const TC_SUB 5
const TC_MUL 6
const TC_DIV 7
const TC_BIT_AND 8
const TC_BIT_OR 9
const TC_FN 'f'
const TC_LET 'l'
const TC_CONST 'c'
const TC_STATIC 's'
```

```
const SYM_RESOLVED 0
const SYM_UNRESOLVED 1
const SYM_VARIABLE 2
```

```
const sym_name 0
const sym_length 4
const sym_kind 8
const sym_value 12
const sym_next 16
const sizeof_symbol 20
```

```
const tc_src 0
const tc_src_end 4
const tc_next_char 8
const tc_next_char_type 12
const tc_next_token 16
const tc_next_token_data 20
const tc_next_token_length 24
const tc_dst 28
const tc_heap 32
const tc_symbols 36
const tc_flash_offset 40
const tc_fn_dst 44
```

17.3.2 Scanner

The `CHAR_TYPES` table must be updated to support the new single character tokens (+, -, *, /, &, |, comma, parentheses and curly braces). For convenience, we set the

character types of +, -, *, /, &, and | to the corresponding token values. And we set the type of the others, including the quote, to their ASCII code. The KEYWORDS table must also be updated to remove instructions now handled with expressions (add, sub, etc) and to add the new let and const keywords.

```
static TC_CHAR_TYPES
  1 1 1 1 1 1 1 1 1 32 32 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  32 1 1 1 1 1 8 39 40 41 6 4 44 5 1 7 2 2 2 2 2 2 2 2 2 2 58 59 1 1 1 1
  1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 1 1 1 3
  1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 123 9 125 1 1
  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
static TC_KEYWORDS
  4 105'i' 102'f' 108'l' 116't' 140
  4 105'i' 102'f' 101'e' 113'q' 141
  4 105'i' 102'f' 103'g' 116't' 142
  4 105'i' 102'f' 108'l' 101'e' 143
  4 105'i' 102'f' 110'n' 101'e' 144
  4 105'i' 102'f' 103'g' 101'e' 145
  4 103'g' 111'o' 116't' 111'o' 146
  5 115's' 116't' 111'o' 114'r' 101'e' 148
  3 115's' 101'e' 116't' 151
  3 112'p' 111'o' 112'p' 152
  3 114'r' 101'e' 116't' 157
  4 114'r' 101'e' 116't' 118'v' 158
  2 102'f' 110'n' 102
  3 108'l' 101'e' 116't' 108
  5 99'c' 111'o' 110'n' 115's' 116't' 99
  6 115's' 116't' 97'a' 116't' 105'i' 99'c' 115
  0
```

The first scanner functions are unchanged compared with the labels compiler:

```
fn mem_compare 3(ptr1, ptr2, size) {
  let i cst_0;
:step2
  get 7i, get 2size ifge step4;
  get 0 get 7 add call load8(ptr1 + i), get 1 get 7 add call load8(ptr2 + ≀
                                                                ≀i) ifne step4;
  get 7 cst_1 addi + 1 set 7i; goto step2;
:step4
  get 2 get 7 subsize - i retv;
}

fn tc_get_keyword 2(start, length) {
  let len cst_0;
  let ptr cst TC_KEYWORDS;
```

```
:step2
  get 7 call load8(ptr) set 6len;
  get 6len, cst_0 ifne step4;
  cst8 3TC_IDENTIFIER retv;
:step4
  get 1length, get 6len ifne step7;
  get 0 get 7 cst_1 add get 1 call mem_compare(start, ptr + 1, length), cs
                                                                    t_0 ifne step7;
  get 7 get 6 add cst_1 add call load8(ptr + len + 1) retv;
:step7
  get 7 get 6 add cst8 2 addptr + len + 2 set 7ptr; goto step2;
}

fn tc_read_char 1(self) {
  let src get 0 cst_0 add load*(self+tc_src);
  let src_end get 0 cst8 4 add load*(self+tc_src_end);
  get 5src, get 6src_end iflt step2;
  cst8 10 call panic(10);
:step2
  get 5 cst_1 addsrc + 1 set 5src;
  let c cst_0;
  let type cst_0;
  get 5src, get 6src_end ifge end;
  get 5 call load8(src) set 7c;
  cst TC_CHAR_TYPES get 7 add call load8(TC_CHAR_TYPES + c) set 8type;
:end
  get 0 cst_0 add(self+tc_src), get 5src store;
  get 0 cst8 8 add(self+tc_next_char), get 7c store;
  get 0 cst8 12 add(self+tc_next_char_type), get 8type store;
  get 8type retv;
}

fn tc_read_integer 1(self) {
  let type get 0 cst8 12 add load*(self+tc_next_char_type);
  let v cst_0;
:step2
  get 5type, cst8 2TC_INTEGER ifne step5;
  get 6 cst8 10 mul get 0 cst8 8 add load cst8 48 sub addv * 10 + (*(self+
                                                    tc_next_char) - '0') set 6v;
  get 0 call tc_read_char(self) set 5type;
  goto step2;
:step5
  get 0 cst8 20 add(self+tc_next_token_data), get 6v store;
  cst8 2TC_INTEGER retv;
}
```

To support the new quoted characters tokens such as "'a'" we add the following function. It starts by reading the first quote (the caller should check that the next

character is a quote). It then checks if the second character, in $value$, is printable, and panics otherwise. Finally, it checks that the third character is a quote, sets the $next\_token\_data$ to $value$ and return the INTEGER token type:

```
fn tc_read_quoted_char 1(self) {
  get 0 call tc_read_char(self) pop;
  let value get 0 cst8 8 add load*(self+tc_next_char);
  get 5value, cst8 32 iflt not_printable;
  get 5value, cst8 127 iflt printable;
:not_printable
  cst8 11 call panic(11);
:printable
  get 0 call tc_read_char(self), cst8 39''' ifeq ok;
  cst8 12 call panic(12);
:ok
  get 0 call tc_read_char(self) pop;
  get 0 cst8 20 add(self+tc_next_token_data), get 5value store;
  cst8 2TC_INTEGER retv;
}
```

The remaining scanner functions are essentially unchanged compared with the labels compiler. We just add a new case in tc_read_token, which calls the above function if the next character is a quote:

```
fn tc_read_identifier 1(self) {
  let start get 0 cst_0 add load*(self+tc_src);
  let type get 0 cst8 12 add load*(self+tc_next_char_type);
:step2
  get 6type, cst8 3TC_IDENTIFIER ifeq step3;
  get 6type, cst8 2TC_INTEGER ifne step4;
:step3
  get 0 call tc_read_char(self) set 6type; goto step2;
:step4
  let length get 0 cst_0 add load get 5 sub*(self+tc_src) - start;
  get 0 cst8 20 add(self+tc_next_token_data), get 5start store;
  get 0 cst8 24 add(self+tc_next_token_length), get 7length store;
  get 5 get 7 call tc_get_keyword(start, length) retv;
}

fn tc_read_token 1(self) {
  let type get 0 cst8 12 add load*(self+tc_next_char_type);
:step1
  get 5type, cst8 32' ' ifne step3;
  get 0 call tc_read_char(self) set 5type; goto step1;
:step3
  let token get 5type;
  get 5type, cst8 2TC_INTEGER ifne step4;
  get 0 call tc_read_integer(self) set 6token; goto end;
:step4
```

```
  get 5type, cst8 39''' ifne step5;
  get 0 call tc_read_quoted_char(self) set 6token; goto end;
:step5
  get 5type, cst8 3TC_IDENTIFIER ifne step6;
  get 0 call tc_read_identifier(self) set 6token; goto end;
:step6
  get 5type, cst_0 ifeq end;
  get 0 call tc_read_char(self) pop;
:end
  get 0 cst8 16 add(self+tc_next_token), get 6token store;
  ret;
}
```

17.3.3 Backend

The backend is extended with new functions to write the opcode instructions needed by the parser. Its first functions are unchanged compared with the labels compiler:

```
fn mem_allocate 2(size, ptr_p) {
  let ptr get 1 load*ptr_p;
  get 1ptr_p, get 6 get 0 addptr + size store;
  get 6ptr retv;
}
fn tc_write8 2(self, value) {
  cst_1 get 0 cst8 28 add call mem_allocate get 1 call store8(mem_allocate⌇
                                                    ⌇(1, self+tc_dst), value);
  ret;
}
fn tc_write16 2(self, value) {
  cst8 2 get 0 cst8 28 add call mem_allocate get 1 call store16(mem_alloca⌇
                                                  ⌇te(2, self+tc_dst), value);
  ret;
}
fn tc_write32 2(self, value) {
  cst8 4 get 0 cst8 28 add call mem_allocate(4, self+tc_dst), get 1value s⌇
                                                                  ⌇tore;
  ret;
}
```

We then add a generic utility function to write an instruction with a single byte argument, used later on:

```
fn tc_write_insn 3(self, opcode, argument) {
  get 0 get 1 call tc_write8(self, opcode);
  get 0 get 2 call tc_write8(self, argument);
  ret;
}
```

The next two functions are unchanged (`last_placeholder` and `new_placeholder` refer to x and y in Algorithm 16.8, respectively):

```
fn tc_add_placeholder 2(self, placeholder_p) {
  let new_placeholder get 0 cst8 28 add load*(self+tc_dst);
  let last_placeholder get 1 load*placeholder_p;
  get 1placeholder_p, get 6new_placeholder store;
  get 7last_placeholder, cst_0 ifne step4; get 6new_placeholder set 7last_⌇
                                                                ⌇placeholder;
:step4
  get 6 get 7 subnew_placeholder - last_placeholder retv;
}

fn tc_fill_placeholders 2(placeholder, value) {
  let offset cst_0;
:step2
  get 0placeholder, cst_0 ifeq end;
:step3
  get 0 call load16(placeholder) set 6offset;
  get 0 get 1 call store16(placeholder, value);
  get 6offset, cst_0 ifne step6; ret;
:step6
  get 0 get 6 subplaceholder - offset set 0placeholder;
  goto step2;
:end
  ret;
}
```

The following new function writes an instruction to push a given value on the stack. It encapsulates the details related to the `cst_0`, `cst_1`, `cst8` and `cst` instructions by writing the appropriate instruction depending on *value*:

```
fn tc_write_cst_insn 2(self, value) {
  get 1value, cst_1 ifgt not0_or_1;
  get 0 get 1 call tc_write8(self, value); ret;
:not0_or_1
  get 1value, cst 256 ifge not_byte;
  get 0 cst8 2 get 1 call tc_write_insn(self, 2, value); ret;
:not_byte
  get 0 cst8 3 call tc_write8(self, 3);
  get 0 get 1 call tc_write32(self, value); ret;
}
```

The next function writes the instruction to perform the arithmetic operation specified by *token*, which must be one of `TC_ADD`, `TC_SUB`, `TC_MUL`, `TC_DIV`, `TC_BIT_AND`, or `TC_BIT_OR`. It is trivial since these values are equal to the corresponding opcodes:

```
fn tc_write_binary_insn 2(self, token) {
  get 0 get 1 call tc_write8(self, token);
  ret;
}
```

The following 3 functions write the instruction corresponding to their name. They encapsulate the details related to their encoding.

```
fn tc_write_load_insn 1(self) {
  get 0 cst8 19 call tc_write8(self, 19);
  ret;
}
fn tc_write_ptr_insn 2(self, variable) {
  get 0 cst8 21 get 1 call tc_write_insn(self, 21, variable);
  ret;
}
fn tc_write_get_insn 2(self, variable) {
  get 0 cst8 22 get 1 call tc_write_insn(self, 22, variable);
  ret;
}
```

The next function is simplified by using the new `tc_write_insn` function:

```
fn tc_write_fn_insn 2(self, arity) {
  get 0 cst8 25 get 1 call tc_write_insn(self, 25, arity);
  ret;
}
```

The last backend function writes the instruction to call a given *function*, specified with a symbol. It writes the `call` opcode, followed either by a new placeholder if the symbol is unresolved, or by the symbol's value. By hypothesis, this value is the `call` instruction argument which must be used to call *function*. The following function encapsulates the details of its computation.

```
fn tc_get_fn_value 2(self, fn_dst) {
  get 1 get 0 cst8 40 add load sub cst 786432 subfn_dst - *(self+tc_flash_⸮
                                                    ⸮offset) - 786432 retv;
}
fn tc_write_call_insn 2(self, function) {
  get 0 cst8 26 call tc_write8(self, 26);
  get 1 cst8 8 add load*(function+sym_kind), cst_1SYM_UNRESOLVED ifne reso⸮
                                                                  ⸮lved;
  get 0 get 0 get 1 cst8 12 add call tc_add_placeholder call tc_write16(se⸮
                        ⸮lf, tc_add_placeholder(self, function+sym_value));
  ret;
:resolved
  get 0 get 1 cst8 12 add load call tc_write16(self, *(function+sym_value)⸮
                                                                      ⸮);
  ret;
}
```

17.3.4 Parser

The start of the parser is the same as in the labels compiler:

```
fn sym_lookup 3(symbol, name, length) {
:step2
```

```
  get 0symbol, cst_0 ifeq step7;
  get 0 cst8 4 add load*(symbol+sym_length), get 2length ifne step6;
  get 0 cst_0 add load get 1 get 2 call mem_compare(*(symbol+sym_name), na⌇
                                                  ⌇me, length), cst_0 ifne step6;
  get 0symbol retv;
:step6
  get 0 cst8 16 add load*(symbol+sym_next) set 0symbol; goto step2;
:step7
  cst_0 retv;
}

fn tc_add_symbol 5(self, name, length, kind, value) {
  let symbol cst8 20 get 0 cst8 32 add call mem_allocate(sizeof_symbol, se⌇
                                                                ⌇lf+tc_heap);
  get 0 cst8 36 add load get 1 get 2 call sym_lookup(*(self+tc_symbols), n⌇
                                                  ⌇ame, length), cst_0 ifeq ok;
  cst8 30 call panic(30);
:ok
  get 9 cst_0 add(symbol+sym_name), get 1name store;
  get 9 cst8 4 add(symbol+sym_length), get 2length store;
  get 9 cst8 8 add(symbol+sym_kind), get 3kind store;
  get 9 cst8 12 add(symbol+sym_value), get 4value store;
  get 9 cst8 16 add(symbol+sym_next), get 0 cst8 36 add load*(self+tc_symb⌇
                                                                ⌇ols) store;
  get 0 cst8 36 add(self+tc_symbols), get 9symbol store;
  get 9symbol retv;
}

fn tc_add_or_resolve_symbol 4(self, name, length, value) {
  let symbol get 0 cst8 36 add load get 1 get 2 call sym_lookup(*(self+tc_⌇
                                                  ⌇symbols), name, length);
  get 8symbol, cst_0 ifne found;
  get 0 get 1 get 2 cst_0 get 3 call tc_add_symbol(self, name, length, SYM⌇
                                                  ⌇_RESOLVED, value) retv;
:found
  get 8 cst8 8 add load*(symbol+sym_kind), cst_1SYM_UNRESOLVED ifeq ok;
  cst8 31 call panic(31);
:ok
  get 8 cst8 12 add load get 3 call tc_fill_placeholders(*(symbol+sym_valu⌇
                                                                ⌇e), value);
  get 8 cst8 8 add(symbol+sym_kind), cst_0SYM_RESOLVED store;
  get 8 cst8 12 add(symbol+sym_value), get 3value store;
  get 8symbol retv;
}

fn tc_parse_token 2(self, token) {
  get 0 cst8 16 add load*(self+tc_next_token), get 1token ifeq ok;
```

```
  cst8 20 call panic(20);
:ok
  get 0 call tc_read_token(self);
  ret;
}
fn tc_parse_integer 1(self) {
  get 0 cst8 16 add load*(self+tc_next_token), cst8 2TC_INTEGER ifeq ok;
  cst8 21 call panic(21);
:ok
  let value get 0 cst8 20 add load*(self+tc_next_token_data);
  get 0 call tc_read_token(self);
  get 5value retv;
}
fn tc_parse_identifier 2(self, length_p) {
  get 0 cst8 16 add load*(self+tc_next_token), cst8 3TC_IDENTIFIER ifeq ok⌇
                                                                      ⌇;
  cst8 22 call panic(22);
:ok
  let name get 0 cst8 20 add load*(self+tc_next_token_data);
  get 1length_p, get 0 cst8 24 add load*(self+tc_next_token_length) store;
  get 0 call tc_read_token(self);
  get 6name retv;
}
```

Here we add a new utility function to parse an identifier which must correspond to an existing symbol. This function parses an identifier and returns its corresponding symbol in the list passed as argument in *symbol* (or panics if no symbol is found):

```
fn tc_parse_symbol 2(self, symbol) {
  let length cst_0;
  let name get 0 ptr 6 call tc_parse_identifier(self, &length);
  get 1 get 7 get 6 call sym_lookup(symbol, name, length) set 1symbol;
  get 1symbol, cst_0 ifne ok;
  cst8 33 call panic(33);
:ok
  get 1symbol retv;
}
```

The `tc_parse_static` function is unchanged, but a new trivial `tc_parse_const` function is added for the new "`const` x v" syntax. This function simply adds a new symbol for x, with value v.

```
fn tc_parse_const 1(self) {
  get 0 cst8 99 call tc_parse_token(self, TC_CONST);
  let length cst_0;
  let name get 0 ptr 5 call tc_parse_identifier(self, &length);
  get 0 get 6 get 5 cst_0 get 0 call tc_parse_integer call tc_add_symbol(s⌇
              ⌇elf, name, length, SYM_RESOLVED, tc_parse_integer(self)) pop;
  ret;
```

```
}
fn tc_parse_static 1(self) {
  get 0 cst8 115 call tc_parse_token(self, TC_STATIC);
  let length cst_0;
  let name get 0 ptr 5 call tc_parse_identifier(self, &length);
  let value get 0 cst8 28 add load get 0 cst8 40 add load sub*(self+tc_dst⸘
                                                  ⸘) - *(self+tc_flash_offset);
  get 0 get 6 get 5 cst_0 get 7 call tc_add_symbol(self, name, length, SYM⸘
                                                    ⸘_RESOLVED, value) pop;
:loop
  get 0 cst8 16 add load*(self+tc_next_token), cst8 2TC_INTEGER ifne end;
  get 0 get 0 call tc_parse_integer call tc_write8(self, tc_parse_integer(⸘
                                                      ⸘self)); goto loop;
:end
  ret;
}
```

The `parse_argument` function is updated to match the new "argument" grammar rule, which no longer allows INTEGER arguments. As a consequence, the start of this function, which was calling `tc_parse_integer`, is removed. On the other hand, the `tc_parse_label` function is unchanged (the "label" rule has not changed):

```
fn tc_parse_argument 1(self) {
  let length cst_0;
  let name get 0 ptr 5 call tc_parse_identifier(self, &length);
  let symbol get 0 cst8 36 add load get 6 get 5 call sym_lookup(*(self+tc_⸘
                                                    ⸘symbols), name, length);
  get 7symbol, cst_0 ifne found;
  get 0 get 6 get 5 cst_1 cst_0 call tc_add_symbol(self, name, length, SYM⸘
                                                  ⸘_UNRESOLVED, 0) set 7symbol;
:found
  get 7 cst8 8 add load*(symbol+sym_kind), cst_1SYM_UNRESOLVED ifne resolv⸘
                                                                  ⸘ed;
  get 0 get 7 cst8 12 add call tc_add_placeholder(self, symbol+sym_value) ⸘
                                                                  ⸘retv;
:resolved
  get 7 cst8 12 add load*(symbol+sym_value) retv;
}

fn tc_parse_label 1(self) {
  get 0 cst8 58 call tc_parse_token(self, ':');
  let length cst_0;
  let name get 0 ptr 5 call tc_parse_identifier(self, &length);
  let value get 0 cst8 28 add load get 0 cst8 44 add load sub*(self+tc_dst⸘
                                                  ⸘) - *(self+tc_fn_dst);
  get 0 get 6 get 5 get 7 call tc_add_or_resolve_symbol(self, name, length⸘
                                                        ⸘, value);
  ret;
}
```

The following functions are the main new part of the compiler. They implement the expressions rules in reverse order, starting with "fn_arguments". This rule uses "expr" but since `tc_parse_expr` is implemented last, we need to declare it first.

```
fn tc_parse_expr(self);
```

The `tc_parse_fn_arguments` function parses the arguments of a function call $f(e_0, e_1, \ldots)$, and takes as argument the symbol corresponding to f. It first checks that this symbol is not a local variable, and panics otherwise. It then parses the arguments with the recursive descent method: after parsing the opening parenthesis, it parses a first expression unless the next token is a closing parenthesis. Then, while the next token is a comma, it reads it and parses another expression. This generates the compiled code for the arguments, after which we just need to write a `call` instruction.

```
fn tc_parse_fn_arguments 2(self, function) {
  get 1 cst8 8 add load*(function+sym_kind), cst8 2SYM_VARIABLE ifne ok;
  cst8 34 call panic(34);
:ok
  get 0 cst8 40 call tc_parse_token(self, '(');
  get 0 cst8 16 add load*(self+tc_next_token), cst8 41')' ifeq end;
  get 0 call tc_parse_expr(self);
:loop
  get 0 cst8 16 add load*(self+tc_next_token), cst8 44',' ifne end;
  get 0 call tc_read_token(self);
  get 0 call tc_parse_expr(self);
  goto loop;
:end
  get 0 cst8 41 call tc_parse_token(self, ')');
  get 0 get 1 call tc_write_call_insn(self, function);
  ret;
}
```

A primitive expression can either start with an integer, an identifier, or an opening parenthesis. In the first case we generate the code to push this integer on the stack:

```
fn tc_parse_primitive_expr 1(self) {
  let symbol cst_0;
  get 0 cst8 16 add load*(self+tc_next_token), cst8 2TC_INTEGER ifne not_i⸘
                                                                  ⸘nteger;
  get 0 get 0 call tc_parse_integer call tc_write_cst_insn(self, tc_parse_⸘
                                                               ⸘integer(self));
  ret;
```

In the second case, there must be a symbol for this identifier. If it is followed by an opening parenthesis this is a function call, and we parse it with the above function.

```
:not_integer
  get 0 cst8 16 add load*(self+tc_next_token), cst8 3TC_IDENTIFIER ifne pa⸘
                                                                  ⸘rentheses;
  get 0 get 0 cst8 36 add load call tc_parse_symbol(self, *(self+tc_symbol⸘
```

```
                                                                ⌇s)) set 5symbol;
  get 0 cst8 16 add load*(self+tc_next_token), cst8 40'(' ifne identifier;
  get 0 get 5 call tc_parse_fn_arguments(self, symbol);
  ret;
```

Otherwise we write the appropriate instruction depending on the kind of symbol referred to by the identifier (or panic if the symbol is unresolved).

```
:identifier
  get 5 cst8 8 add load*(symbol+sym_kind), cst8 2SYM_VARIABLE ifne not_var⌇
                                                                  ⌇iable;
  get 0 get 5 cst8 12 add load call tc_write_get_insn(self, *(symbol+sym_v⌇
                                                                ⌇alue));
  ret;
:not_variable
  get 5 cst8 8 add load*(symbol+sym_kind), cst_0SYM_RESOLVED ifne error;
  get 0 get 5 cst8 12 add load call tc_write_cst_insn(self, *(symbol+sym_v⌇
                                                                ⌇alue));
  ret;
:error
  cst8 35 call panic(35);
```

In the last case, we simply need to parse an expression between parentheses:

```
:parentheses
  get 0 cst8 40 call tc_parse_token(self, '(');
  get 0 call tc_parse_expr(self);
  get 0 cst8 41 call tc_parse_token(self, ')');
  ret;
}
```

A pointer expression either starts with "*" or "&", or is a primitive expression. Following the recursive descent method, the first case is trivial: we just need to read the "*" token, parse a pointer expression recursively, and finally generate a `load` instruction (recall that "*e" means "the value at address e"). The second case is also simple: the "&" must be followed by an identifier which must correspond to a local variable or function parameter x. If it is we must generate a `ptr` x instruction (recall that "&x" means "the address of x's stack frame slot"), otherwise this is an error:

```
fn tc_parse_pointer_expr 1(self) {
  let symbol cst_0;
  get 0 cst8 16 add load*(self+tc_next_token), cst8 6TC_MUL ifne not_mul;
  get 0 call tc_read_token(self);
  get 0 call tc_parse_pointer_expr(self);
  get 0 call tc_write_load_insn(self);
  ret;
:not_mul
  get 0 cst8 16 add load*(self+tc_next_token), cst8 8TC_BIT_AND ifne not_b⌇
                                                                  ⌇it_and;
  get 0 call tc_read_token(self);
```

```
  get 0 get 0 cst8 36 add load call tc_parse_symbol(self, *(self+tc_symbol⸮
                                                          ⸮s)) set 5symbol;
  get 5 cst8 8 add load*(symbol+sym_kind), cst8 2SYM_VARIABLE ifne error;
  get 0 get 5 cst8 12 add load call tc_write_ptr_insn(self, *(symbol+sym_v⸮
                                                              ⸮alue));
  ret;
:error
  cst8 36 call panic(36);
:not_bit_and
  get 0 call tc_parse_primitive_expr(self);
  ret;
}
```

The remaining expression parsing functions are straightforward, again by following the recursive descent method. After parsing a first subexpression, a loop is used, while the next token is a permitted operator (*e.g.*, "+" or "−" for the "add_expr" rule), to read the operator, parse a subexpression, and write the operator's instruction:

```
fn tc_parse_mult_expr 1(self) {
  get 0 call tc_parse_pointer_expr(self);
  let next_token get 0 cst8 16 add load*(self+tc_next_token);
:loop
  get 5next_token, cst8 6TC_MUL ifeq mul_or_div;
  get 5next_token, cst8 7TC_DIV ifne end;
:mul_or_div
  get 0 call tc_read_token(self);
  get 0 call tc_parse_pointer_expr(self);
  get 0 get 5 call tc_write_binary_insn(self, next_token);
  get 0 cst8 16 add load*(self+tc_next_token) set 5next_token;
  goto loop;
:end
  ret;
}
fn tc_parse_add_expr 1(self) {
  get 0 call tc_parse_mult_expr(self);
  let next_token get 0 cst8 16 add load*(self+tc_next_token);
:loop
  get 5next_token, cst8 4TC_ADD ifeq add_or_sub;
  get 5next_token, cst8 5TC_SUB ifne end;
:add_or_sub
  get 0 call tc_read_token(self);
  get 0 call tc_parse_mult_expr(self);
  get 0 get 5 call tc_write_binary_insn(self, next_token);
  get 0 cst8 16 add load*(self+tc_next_token) set 5next_token;
  goto loop;
:end
  ret;
}
fn tc_parse_bit_and_expr 1(self) {
```

```
  get 0 call tc_parse_add_expr(self);
:loop
  get 0 cst8 16 add load*(self+tc_next_token), cst8 8TC_BIT_AND ifne end;
  get 0 call tc_read_token(self);
  get 0 call tc_parse_add_expr(self);
  get 0 cst8 8 call tc_write_binary_insn(self, TC_BIT_AND);
  goto loop;
:end
  ret;
}
fn tc_parse_expr 1(self) {
  get 0 call tc_parse_bit_and_expr(self);
:loop
  get 0 cst8 16 add load*(self+tc_next_token), cst8 9TC_BIT_OR ifne end;
  get 0 call tc_read_token(self);
  get 0 call tc_parse_bit_and_expr(self);
  get 0 cst8 9 call tc_write_binary_insn(self, TC_BIT_OR);
  goto loop;
:end
  ret;
}
```

The `tc_parse_instruction` function is similar to its previous version in the labels compiler and is even simpler, since an instruction can no longer be a label (labels have been moved to the "statement" rule):

```
static ARG_SIZES
  0 0 1 4 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 0 0 1 1 1 0 0 2 0 0 0 0
```

```
fn tc_parse_instruction 1(self) {
  let opcode get 0 cst8 16 add load cst8 128 sub*(self+tc_next_token) - 12≀
                                                                    ≀8;
  get 0 get 5 call tc_write8(self, opcode);
  get 0 call tc_read_token(self);
  let arg_size cst ARG_SIZES get 5 add call load8(ARG_SIZES + opcode);
  get 6arg_size, cst_0 ifne not0;
  ret;
:not0
  let arg get 0 call tc_parse_argument(self);
  get 6arg_size, cst_1 ifne not1;
  get 0 get 7 call tc_write8(self, arg); ret;
:not1
  get 6arg_size, cst8 2 ifne not2;
  get 0 get 7 call tc_write16(self, arg); ret;
:not2
  get 0 get 7 call tc_write32(self, arg); ret;
}
```

The following function parses the new "`let` x e;" syntax. It takes as parameter the stack frame slot index of x and returns the index to use for the next `let`. The

parsing itself is trivial. No code needs to be generated besides the one generated while parsing e. We just need to add x in the list of symbols, with value $variable$.

```
fn tc_parse_let_stmt 2(self, variable) {
  get 0 cst8 108 call tc_parse_token(self, TC_LET);
  let length cst_0;
  let name get 0 ptr 6 call tc_parse_identifier(self, &length);
  get 0 call tc_parse_expr(self);
  get 0 cst8 59 call tc_parse_token(self, ';');
  get 0 get 7 get 6 cst8 2 get 1 call tc_add_symbol(self, name, length, SY⸘
                                                  ⸘M_VARIABLE, variable) pop;
  get 1 cst_1 addvariable + 1 retv;
}
```

The next function implements the "statement" rule. It takes as parameter the stack frame slot to use if the statement is a `let` construct, and returns the slot to use for the next statement. If the next token is a colon or the `let` keyword, we just need to call the function to parse a label or a let statement. Otherwise, if the next token is not an opcode keyword ($token \geq 128$) or a semicolon, we need to parse an expression. Then, while the next token is a comma, we should read it and parse another expression. Finally, we should parse an instruction if the next token is not a semicolon.

```
fn tc_parse_statement 2(self, next_variable) {
  get 0 cst8 16 add load*(self+tc_next_token), cst8 58':' ifne not_label;
  get 0 call tc_parse_label(self);
  goto end;
:not_label
  get 0 cst8 16 add load*(self+tc_next_token), cst8 108TC_LET ifne expr_or⸘
                                                              ⸘_insn;
  get 0 get 1 call tc_parse_let_stmt(self, next_variable) retv;
:expr_or_insn
  get 0 cst8 16 add load*(self+tc_next_token), cst8 59';' ifeq insn;
  get 0 cst8 16 add load*(self+tc_next_token), cst8 128 ifge insn;
  get 0 call tc_parse_expr(self);
:loop
  get 0 cst8 16 add load*(self+tc_next_token), cst8 44',' ifne insn;
  get 0 call tc_read_token(self);
  get 0 call tc_parse_expr(self);
  goto loop;
:insn
  get 0 cst8 16 add load*(self+tc_next_token), cst8 59';' ifeq insn_end;
  get 0 cst8 16 add load*(self+tc_next_token), cst8 128 ifge ok;
  cst8 24 call panic(24);
:ok
  get 0 call tc_parse_instruction(self);
:insn_end
  get 0 cst8 59 call tc_parse_token(self, ';');
:end
```

```
  get 1next_variable retv;
}
```

The `tc_parse_fn_name` function is the same as in the labels compiler, except that it now computes the symbol's value with the new `tc_get_fn_value` function:

```
fn tc_parse_fn_name 1(self) {
  let length cst_0;
  let name get 0 ptr 5 call tc_parse_identifier(self, &length);
  let fn_dst get 0 cst8 28 add load*(self+tc_dst);
  get 0 cst8 44 add(self+tc_fn_dst), get 7fn_dst store;
  let value get 0 get 7 call tc_get_fn_value(self, fn_dst);
  get 0 get 6 get 5 get 8 call tc_add_or_resolve_symbol(self, name, length
                                                        , value) retv;
}
```

The overall algorithm of the `tc_parse_fn_parameters` function is the same as the one parsing function arguments, and derives from the recursive descent method. This function parses an opening parenthesis and then, while the next token is not a closing parenthesis, parses a comma (except at the first iteration) and an identifier. Each identifier is added to the list of symbols with its index i as value. i is initialized to 0, incremented after each identifier, and finally returned to the caller.

```
fn tc_parse_fn_parameters 1(self) {
  let i cst_0;
  let name cst_0;
  let length cst_0;
  get 0 cst8 40 call tc_parse_token(self, '(');
:loop
  get 0 cst8 16 add load*(self+tc_next_token), cst8 41')' ifeq end;
  get 5i, cst_0 ifle identifier;
  get 0 cst8 44 call tc_parse_token(self, ',');
:identifier
  get 0 ptr 7 call tc_parse_identifier(self, &length) set 6name;
  get 0 get 6 get 7 cst8 2 get 5 call tc_add_symbol(self, name, length, SY
                                                    M_VARIABLE, i) pop;
  get 5 cst_1 addi + 1 set 5i;
  goto loop;
:end
  get 0 call tc_read_token(self);
  get 5i retv;
}
```

The `parse_fn_body` function is updated to parse the new curly braces, and no longer parses the function's arity, now passed as argument. It also parses statements instead of instructions, and keeps track of the stack frame slot to use for `let` statements (initialized to $arity + 4$ – cf. Section 17.2).

```
fn tc_parse_fn_body 3(self, function, arity) {
  get 0 cst8 16 add load*(self+tc_next_token), cst8 59';' ifne body;
```

```
  get 0 call tc_read_token(self);
  get 1 cst8 8 add(function+sym_kind), cst_1SYM_UNRESOLVED store;
  get 1 cst8 12 add(function+sym_value), cst_0 store;
  ret;
:body
  get 0 cst8 123 call tc_parse_token(self, '{');
  get 0 get 2 call tc_write_fn_insn(self, arity);
  let next_variable get 2 cst8 4 addarity + 4;
:loop
  get 0 cst8 16 add load*(self+tc_next_token), cst8 125'}' ifeq end;
  get 0 get 7 call tc_parse_statement(self, next_variable) set 7next_varia≀
                                                                  ≀ble;
  goto loop;
:end
  get 0 call tc_read_token(self);
  ret;
}
```

The `tc_check_symbols` function is unchanged, while `tc_parse_fn` is updated to parse the function parameters before parsing its body. Note that restoring the *heap* and *symbols* variables now also deletes the symbols added for the function parameters and for the local variables (in addition to the label symbols). This is what we want, so that parameters and local variables defined in one function cannot be used in another.

```
fn tc_check_symbols 2(symbol, end_symbol) {
:loop
  get 0symbol, get 1end_symbol ifeq end;
  get 0 cst8 8 add load*(symbol+sym_kind), cst_1SYM_UNRESOLVED ifne next;
  cst8 32 call panic(32);
:next
  get 0 cst8 16 add load*(symbol+sym_next) set 0symbol; goto loop;
:end
  ret;
}
fn tc_parse_fn 1(self) {
  get 0 cst8 102 call tc_parse_token(self, TC_FN);
  let function get 0 call tc_parse_fn_name(self);
  let heap get 0 cst8 32 add load*(self+tc_heap);
  let symbols get 0 cst8 36 add load*(self+tc_symbols);
  let arity get 0 call tc_parse_fn_parameters(self);
  get 0 get 5 get 8 call tc_parse_fn_body(self, function, arity);
  get 0 cst8 36 add load get 7 call tc_check_symbols(*(self+tc_symbols), s≀
                                                              ≀ymbols);
  get 0 cst8 36 add(self+tc_symbols), get 7symbols store;
  get 0 cst8 32 add(self+tc_heap), get 6heap store;
  ret;
}
```

Finally, `tc_parse_program` is updated to handle the new "`const`" case, while the `tc_main` function is unchanged:

```
fn tc_parse_program 1(self) {
:loop
  get 0 cst8 16 add load*(self+tc_next_token), cst8 102TC_FN ifne not_fn;
  get 0 call tc_parse_fn(self); goto loop;
:not_fn
  get 0 cst8 16 add load*(self+tc_next_token), cst8 115TC_STATIC ifne not_
                                                                    static;
  get 0 call tc_parse_static(self); goto loop;
:not_static
  get 0 cst8 16 add load*(self+tc_next_token), cst8 99TC_CONST ifne end;
  get 0 call tc_parse_const(self); goto loop;
:end
  get 0 cst8 16 add load*(self+tc_next_token), cst_0 ifeq ok;
  cst8 23 call panic(23);
:ok
  get 0 cst8 36 add load cst_0 call tc_check_symbols(*(self+tc_symbols), 0
                                                                    ); ret;
}
fn tc_main 3(src_buffer, dst_buffer, flash_buffer) {
  let fn_dst cst_0;
  let flash_offset get 1 get 2 subdst_buffer - flash_buffer;
  let symbols cst_0;
  let heap get 1 cst 12288 adddst_buffer + 12288;
  let dst get 1 cst8 4 adddst_buffer + 4;
  let next_token_length cst_0;
  let next_token_data cst_0;
  let next_token cst_0;
  let next_char_type cst_0;
  let next_char cst_0;
  let src_end get 0 cst8 4 add get 0 load addsrc_buffer + 4 + *src_buffer;
  let src get 0 cst8 3 addsrc_buffer + 3;
  let panic3 cst_0;
  let panic2 cst_0;
  let panic1 cst_0;
  let panic0 cst_0;
  let error cst_0;
  ptr 22 call panic_result(&panic0) set 23error;
  get 23error, cst_0 ifeq ok;
  get 1dst_buffer, get 18 get 0 sub cst8 4 subsrc - src_buffer - 4 store;
  get 23error retv;
:ok
  ptr 18 call tc_read_char(&src) pop;
  ptr 18 call tc_read_token(&src);
  ptr 18 call tc_parse_program(&src);
  get 1dst_buffer, get 11 get 1 sub cst8 4 subdst - dst_buffer - 4 store;
  cst_0 retv;
}
```

17.4 Compilation and tests

To compile the above source code proceed as follows (see also Figure 16.4). First launch the command editor by typing "w000c1172"+Enter in the memory editor, followed by "r".

Edit v1 Type "F3"+"r" and "F4"+"r" to load and edit the current compiler version. Then update it to the 1^{st} version of the expressions compiler (that is, the above code with the parts highlighted in red). For convenience, we also provide this code in the `expressions_compiler_v1.txt` file in https://ebruneton.github.io/toypc/sources.zip. When you are done, exit the text editor and type "F5"+"r" to save your work. Alternatively, you can "cheat" by running the following command on an external computer (see Section 16.4 for more details):

```
user@host:~$ python3 flash_helper.py < part3/expressions_compiler_v1.txt
```

Compile v1 In the command editor, type "F6"+"r" to compile the code you typed. If all goes well, after about 2 seconds, you should get a result equal to 0 (meaning that no error was found). If this is not the case use Appendix D to get the error code meaning, fix this error, save the program and compile it again. Repeat this process until the compilation is successful. Then type "F7"+"r" to save the result.

Test v1 Type "F2"+"r" to create a new program, "F4"+"r" to edit it, and type the following small test program, which computes the factorial of 6:

```
fn factorial(n);
fn test() { factorial(6) retv; }
fn factorial(n) {
  n, 0 ifne not_zero; 1 retv;
  :not_zero factorial(n - 1) * n retv;
}
```

Then type "F9"+"r" to compile and run it. If the result is not 720 = $2D0_{16}$ this means that the compiler is wrong. In this case, type "F8"+"r" to restore the labels compiler. Then repeat the previous steps and double check everything until this test passes.

Edit v2 Type "F3"+"r" to load the 1^{st} version of the expression compiler and "F4"+"r" to edit it. Then update it to the 2^{nd} version (that is, the code in the previous section, with the parts highlighted in green). For convenience, we also provide this code in the `expressions_compiler_v2.txt` file. Then save this new version with the F5 command. Alternatively, run the following command on an external computer:

```
user@host:~$ python3 flash_helper.py < part3/expressions_compiler_v2.txt
```

Compile v2 Type"F6"+"r" to compile this new code. The result should be 0, meaning "no error". If this is not the case, repeat the "Edit v2" and "Compile v2" steps until all errors are fixed.

Test v2 The compilation of the 2^{nd} version of the expressions compiler should give the same code for each expression as the manually written code in the 1^{st} version. Consequently, the compiled code of the 2^{nd} version should be identical to that of the 1^{st} version. To check this type "F10"+"r". The result should be 0. If this is not the case, repeat the steps from "Edit v2" until this test passes.

CHAPTER

18 Statements Compiler

We now have a compiler for a toy programming language which no longer requires us to manually compute function addresses, instruction offsets or stack frame slot indices. We also have a natural syntax for expressions. However, writing conditional instructions or loops is still not very easy. The main reason is that one must still use raw bytecode instructions for that, which have two main drawbacks. The first is an unnatural syntax. For instance, one must write "`x, y ifgt greater;`" to implement "if $x > y$ jump to the `greater` label". The second drawback is the use of labels. Although they are much better than instruction offsets, their use is still not very natural. For instance, to implement "if $x \leq y$ call $f(x)$", one must put a label after the call to jump to it if the condition is *not* true: "`x, y ifgt greater; f(x); :greater`". This chapter extends our toy programming language and its compiler in order to solve these issues.

18.1 Requirements

This section introduces new grammar rules to replace the remaining bytecode instructions in our current programming language, namely `iflt`, `ifeq`, `ifgt`, `ifle`, `ifne`, `ifge`, `goto`, `store`, `set`, `pop`, `ret`, and `retv`.

18.1.1 Assignment and return

To replace `ret` and `retv` we introduce the new "`return`" keyword. We then use "`return;`" instead of "`ret;`", and "`return` e`;`" instead of "e `retv;`" (where e is an expression). To replace `set` we introduce the *assignment* syntax "x `=` e`;`", where x is a function parameter or local variable name, and e an expression. "x `=` e`;`" is equivalent to "e `set` x`;`" (sometimes noted $x \leftarrow e$). Similarly, to replace `store`, we use the assignment syntax "`*`a `=` e`;`", where a and e are expressions. "`*`a `=` e`;`" is equivalent to "a`,` e `store;`" and means "store e at address a".

18.1.2 Expression statements

To remove the `pop` instruction we make it implicit in all *expression statements* "e`;`". This syntax was already used in the previous chapter, but it now means "e `pop;`".

Note that *this requires all functions to return a value*. We remove this restriction in Chapter 19. In the mean time, we redefine "`return;`" as a shorthand for "`return 0;`".

18.1.3 Conditional statements

To implement cases such as "if $x < y$ call $f(x)$" in a natural way we require our programming language to support this kind of syntax directly. More precisely it should be possible to write "`if x < y { f(x); }`" for this. The curly braces are mandatory and allow several statements to be executed if the condition is true (as in a function's body). To call $f(y)$ if the condition is false, it should be possible to write "`if x < y { f(x); } else { f(y); }`". We also require the possibility to chain `if` statements, as in, for instance "`if x == 0 {...} else if x == 1 {...} else if x == 2 {...} ...`".

Besides the new "<" operator, we also introduce "==", ">", "<=", "!=", and ">=", to implement $x = y$, $x > y$, $x \leq y$, $x \neq y$, and $x \geq y$ (respectively). Conditions can be more complex than a single comparison. Calling $f(x)$ if $x \geq 0$ *and* $x < 10$ could be done with "`if x >= 0 { if x < 10 { f(x); } }`", but it is more natural to use an explicit "and" operator, noted "&&": "`if x >= 0 && x < 10 { f(x); }`". Similarly, calling $f(x)$ if $x < 5$ *or* $x = 7$ could be done with "`if x < 5 { f(x); } else if x == 7 { f(x); }`" but it is better to use an explicit "or" operator, noted "||": "`if x < 5 || x == 7 { f(x); }`". It should also be possible to combine these operators, as in the following example (which tests if x is a digit or a lowercase letter): "`if x >= '0' && x <= '9' || x >= 'a' && x <= 'z' { ... }`".

18.1.4 Loops

Executing several instructions repeatedly currently requires a `goto` after the last instruction to go back at the beginning. To avoid this we introduce the new "`loop`" keyword and the "`loop { ... }`" syntax. By definition, this executes the statements inside the mandatory curly braces repeatedly, forever. To stop the loop one could use a `return` statement, but this would return from the whole function. To avoid this we introduce a "`break`" statement. By definition, it jumps to the next statement after the "`loop`" block. For instance, in "`loop { if x == 0 { break; } ... } f(y);`", the `break` exits the loop and jumps to the `f(y);` statement. This pattern is actually quite frequent. To make it easier to use, we introduce the new "`while`" keyword and the "`while` b `{ ... }`" syntax (where b is a *boolean* expression using the "<", "==", ">", "<=", "!=", ">=", "&&", and "||" operators defined above). By definition, this executes the statements inside the curly braces while b is true. The above `break` example can then by rewritten into the simpler form "`while x != 0 { ... } f(y);`".

18.1.5 Grammar

In order to support the above requirements we extend the grammar of our programming language as follows (unchanged parts are in gray):

```
program: (fn | static | const)* END
fn: "fn" fn_name fn_parameters fn_body
fn_name: IDENTIFIER
fn_parameters: "(" (IDENTIFIER "," IDENTIFIER)*)? ")"
fn_body: "{" (let_stmt | stmt)* "}" | ";"
let_stmt: "let" IDENTIFIER "=" expr ";"
stmt: if_stmt | while_or_loop_stmt | break_stmt | return_stmt | expr_or_assign_stmt
if_stmt: "if" boolean_expr block_stmt ("else" (block_stmt | if_stmt))?
while_or_loop_stmt: ("while" boolean_expr | "loop") block_stmt
break_stmt: "break" ";"
return_stmt: "return" expr? ";"
expr_or_assign_stmt: expression assigment? ";"
assignment: "=" expr
block_stmt: "{" stmt* "}"
boolean_expr: and_expr ("||" and_expr)*
and_expr: comparison_expr ("&&" comparison_expr)*
comparison_expr: expr ("<" | "==" | ">" | "<=" | "!=" | ">=") expr
expr: bit_and_expr ("|" bit_and_expr)*
bit_and_expr: shift_expr ("&" shift_expr)*
shift_expr: add_expr (("<<" | ">>") add_expr)*
add_expr: mult_expr (("+" | "-") mult_expr)*
mult_expr: pointer_expr (("*" | "/") pointer_expr)*
pointer_expr: "*" pointer_expr | "&" IDENTIFIER | primitive_expr
primitive_expr: INTEGER | IDENTIFIER fn_arguments? | "(" expr ")"
fn_arguments: "(" (expr "," expr)*)? ")"
static: "static" IDENTIFIER "=" "[" INTEGER ("," INTEGER)* "]" ";"
const: "const" IDENTIFIER "=" INTEGER ";"
```

This grammar no longer uses any bytecode instruction. Indeed the "instruction" rule has been removed, and replaced with several rules for statements. These new rules should be self-explanatory, but two points should be noted:

- "`let`" statements cannot be used in conditional or loop statements. This is done on purpose, so that they are executed in the same order as in the source code, and exactly once (unless the function returns before). We assumed this without verification in the previous chapter, but this is now enforced automatically.
- the assignment statement rule is merged with the expression statement rule and allows invalid assignments such as "`2 = 3;`" (2 cannot be changed to be equal to 3!). This is because using separate rules would lead to an ambiguous grammar. Indeed, after a "`*`" token, there would be no way to decide whether to parse an expression statement or an assignment statement (this depends on the presence or not of a "`=`" token, which could be arbitrarily far after the "`*`"). Invalid assignments must thus be detected by using another method than grammar rules.

This grammar also introduces a new expression rule for the `lsl` and `lsr` instructions,

which were not taken into account in the previous chapter. "e << n" (resp. "e >> n") means e shifted to the left (resp. right) by n bits. These new operators have an intermediate precedence between the bit and "&" and the plus and minus operators. Finally, for consistency with assignment statements, the "const", "static" and "let_stmt" rules are updated to use the assignment notation "=".

18.2 Algorithms

18.2.1 Scanner

The above grammar adds several new keywords. We associate them with $token$ values starting at 128, in alphabetical order (values $128 + x$ for opcode x are no longer used since we removed the opcode keywords). It also adds the "=", "[", and "]" punctuation tokens, as well as the "<<", ">>", "<", "==", ">", "<=", "!=", ">=", "&&", and "||" operator tokens. We associate the former with their ASCII code (61, 91, and 93, respectively), and the latter with values from 10 to 19, respectively (the first 8 are the `lsl`, `lsr`, `iflt`, `ifeq`, `ifgt`, `ifle`, `ifne`, and `ifge` opcodes, respectively).

The new operator tokens require a new algorithm to read them. Indeed, if the next character is a "<", for instance, the second next character must be inspected to know if this is a "<", "<<", or "<=" token. The following table gives the $token$ value depending on these two characters, and defines new `CHAR_TYPES` for the first ones:

1^{st} character	!	&	<	=	>	|
`CHAR_TYPES`	10	11	12	13	14	15
any other case	1	`and`	`iflt`	`'='`	`ifgt`	`or`
2^{nd} character = 1^{st}	1	18	`lsl`	`ifeq`	`lsr`	19
2^{nd} character = =	`ifne`	1	`ifle`	`ifeq`	`ifge`	1

We can then support the operator tokens by extending the algorithm to read a token as follows (where $(column, row)$ table cell coordinates are counted from 0):

1. if the next character's `CHAR_TYPES` t is between 10 and 15
2. read this character c
3. if the next character is equal to c
4. read it and set the $token$ value to the $(t - 10, 1)$ cell's content
5. if the next character is equal to =
6. read it and set the $token$ value to the $(t - 10, 2)$ cell's content
7. in any other case, set the $token$ value to the $(t - 10, 0)$ cell's content
8. otherwise continue with the current algorithm

18.2.2 Parser

Parsing the new grammar rules can be done with the recursive descent method, and is not difficult. However, producing the corresponding compiled code is not as easy as

for expressions. This section presents the main algorithms for doing this.

Assignments

Because the expression and assignment statements rules are merged, when a "=" token is found after an expression e, e has already been parsed and compiled. Hence for instance, for "`x = 1;`", "`get 0`" has already been written in the dst buffer (assuming that x is in the 0^{th} stack frame slot). But this not what we want, since "`x = 1;`" should produce "`cst_1 set 0`". Similarly, for "`*x = 1;`", "`get 0 load`" has already been written, but this statement should produce "`get 0 cst_1 store`".

To solve these issues, and to detect invalid assignments such as "`2 = 3;`", we make each expression parsing function return the *origin* of the value of this expression. By definition, this origin is `ADDRESS` for a dereference expression such as "`*x`", `VARIABLE` for an identifier expression referring to a function argument or a local variable, such as "`x`", and `OTHER` for any other case. Then, when a "=" token is found after an expression e, there are 3 cases:

- if e's origin is `ADDRESS` we decrement dst by 1 to remove the "`load`" which has already been written but is not wanted. We then parse and compile the right hand side of the assignment, and finally write a "`store`" instruction.
- if e's origin is `VARIABLE` we get the variable slot index i by reading the byte at $dst - 1$, and decrement dst by 2 to remove the "`get` i" which has already been written but is not wanted. We then parse and compile the right hand side of the assignment, and finally write a "`set` i" instruction.
- any other case is an invalid assignment.

Conditional statements and loops

Comparison expressions such as "e_1 `<` e_2" are compiled into the code for e_1, followed by the code for e_2, followed by some `if...` instruction to conditionally jump somewhere. However, the precise instruction to use and its target depend on the context. Consider for instance the example in Figure 18.1, where each c_i is a comparison expression. If c_1 is false there is no need to compute c_2 and c_3 to know that their *conjunction* "c_1 and c_2 and c_3" is false. We thus want to jump to the next comparison which could impact the end result, namely c_4. The same reasoning applies to c_2, c_4, and c_5. If c_3 is true, we know at this point that c_1 and c_2 are true as well, and thus that the *disjunction* of the 3 conjunctions is true. We thus want to jump directly to "`f(x)`". The same reasoning applies to c_6. Finally, if any of c_7, c_8, or c_9 is false, the whole condition is false and we want to jump to "`g(y)`".

In this example, c_1 jumps to c_4 when it is *false*, but c_3, in a very similar context, jumps to "`f(x)`" if it is *true*. In order to generate the correct jump instruction and jump target for each comparison expression we use the following method:

- The function compiling a comparison expression only produces the code for its two subexpressions. It does *not* generate a jump instruction, and returns instead the $token$ value corresponding to the comparison operator.

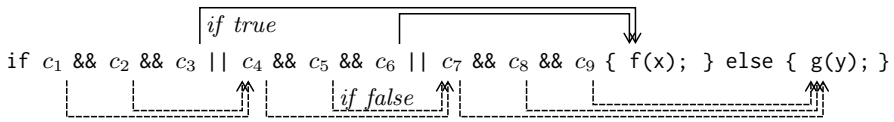


FIGURE 18.1 The jump target of each comparison expression c_i in a general `if` statement, and whether to jump if the comparison is true or false.

- The function compiling conjunctions "c_1 `&&` c_2 `&&` ... c_n" takes as parameter a label l where to jump if the conjunction is false, and generates after each c_i *except the last* an `if...` instruction jumping to l if c_i is false. It returns the $token$ value corresponding to the comparison operator of c_n.
- The function compiling disjunctions "c_1 `||` c_2 `||` ... c_n" takes as parameter a label l where to jump if the disjunction is true. It compiles each conjunction c_i *except the last* by 1) creating a label l_i, 2) passing it to the above function, 3) generating an `if...` instruction jumping to l if c_i is true, 4) placing l_i just after. It compiles the last conjunction c_n with steps 1) and 2), then generates an `if...` instruction jumping to l_n if c_n is *false*, and finally returns the l_n label to the caller.

A whole "`if` e `{...} else {...}`" statement can then be compiled as follows:

- create a l_{then} label and pass it to the above function to parse and compile e. Store the resulting label in an l_{else} variable.
- place the l_{then} label just before parsing and compiling the first block statement.
- create a l_{end} label and generate a `goto` l_{end} instruction.
- place the l_{else} label just before parsing and compiling the second block statement.
- place the l_{end} label.

Similarly, a "`while` e `{...}`" statement can then be compiled as follows:

- create an l_{body} label and pass it to the above function to parse and compile e. Store the resulting label in an l_{end} variable.
- place the l_{body} label just before parsing and compiling the block statement.
- generate a `goto` instruction going back just before e's code.
- place the l_{end} label.

These algorithms generate code of the following form:

```
    code[e], jumps to :then or :else
:then code[then_block] goto end;
:else code[else_block]
:end
```

```
:loop code[e], jumps to :body or :end
:body code[loop_block]
      goto loop;
:end
```

To conclude this section, it should be noted that we no longer need to use symbols for labels. Indeed, we used symbols so far because we needed to find labels from their

name in the program, and to keep track of their resolved or unresolved status. Here we no longer have label names, and we know for each label above when it is resolved or not (each label is initially unresolved, and becomes resolved when it is "placed"). The only remaining piece of data which is needed for each label is its placeholders list. And this boils down to a single address, the one of the last placeholder (since each placeholder contains the offset of the previous one – see Figure 16.2). Hence, in the above algorithms, "creating a label l" simply means initializing a local variable "`l`" to 0, representing an empty list of placeholders. And "passing l" to a function means passing "`&l`", the local variable's address, so that the callee can add placeholders for l. Finally, "returning l" means returning "`l`" and "placing l" means filling the placeholders starting at address "`l`" with l's instruction offset.

Reachability analysis

The above algorithms are sufficient to compile statements, but we add one more in order to detect more errors. One error in particular is a missing "`return`" at the end of a function, *e.g.*, "`f(a) { *a = 0; } g(b) {...}`". In such cases execution "falls through" the end of "`f`" and continues in "`g`", which is wrong. To solve this we could require a "`return`" at the end of every function, but this would be too strict, for instance, for "`h(x) { loop { if x == 0 { return; } x = x-1; } }`". Indeed, execution cannot fall through the end of this function even if it does not end with a "`return`". This is because the end of its "`loop`" statement is not *reachable*, meaning that a statement put immediately after its closing brace could never be executed. On the other hand, the end of the above "`*a = 0;`" assignment *is* reachable. Hence, a more precise test to detect a missing return is to check if the end of the function's body *might be* reachable. Computing this for a statement s can be done as follows:

- if s is an expression or assignment statement we conservatively assume that its end might be reachable (this is not always the case: a function call may never return).
- if s is a "`break`" or "`return`" statement its end is never reachable.
- if s is a block statement "`{` s_1`;` ... s_n`;}`" its end might be reachable if s_n's end might be. If the end of s_i for $i < n$ is unreachable this probably means that s is not doing what the user wants, and we thus raise an error (although s would execute without any issue, unlike a function without return).
- if s is an "`if`" statement we conservatively assume that its end might be reachable if the end of at least one of its branches might be reachable. Note that a missing "`else`" branch is equivalent to "`else {}`", whose end is reachable.
- if s is a "`while`" statement we conservatively assume that its condition might be false at least once, and thus that its end might be reachable (this is not always the case: "`while 0 == 0 {}`" never ends).
- finally, if s is a "`loop`" statement its end might be reachable only if it contains a "`break`" statement.

To compute this we make each statement parsing function return whether the statement's end might be reachable or not. We then use this to detect missing

returns, but also, as a small code size optimization in conditional statements, to avoid generating a "goto" after a block whose end is not reachable (see Section 18.2.2).

18.3 Implementation

We can now extend the expressions compiler in order to support statements. As before, we need to write it in two steps, first without using statements, then with them. To save space, we give the two compiler versions at the same time (without statements in red, with in green). The start of the compiler does not change in the 1^{st} version, but can be rewritten in a clearer way in the 2^{nd}:

```
fn tc_main(src_buffer, dst_buffer, flash_buffer);
fn main(src_buffer, dst_buffer, flash_buffer) {
  return tc_main(src_buffer, dst_buffer, flash_buffer) retv;
}

fn load8(ptr) { return (*ptr) & 255 retv; }
fn load16(ptr) { return (*ptr) & 65535 retv; }
fn store8(ptr, value) { *ptr, = (*ptr) & 4294967040 | value store; retretu⸘
                                                                  ⸘rn; }
fn store16(ptr, value) { *ptr, = (*ptr) & 4294901760 | value store; retret⸘
                                                                  ⸘urn; }

const PANIC_BUFFER = 1074666152;
fn panic_copy(src, dst) {
  *dst, = *src store;
  *(dst + 4), = *(src + 4) store;
  *(dst + 8), = *(src + 8) store;
  *(dst + 12), = *(src + 12) store;
  retreturn;
}
fn panic_result(ptr) {
  panic_copy(&ptr - 16, ptr);
  *PANIC_BUFFER, = ptr store;
  return 0 retv;
}
fn panic(error) {
  panic_copy(*PANIC_BUFFER, &error - 16);
  return error retv;
}
```

18.3.1 Shared constants

Here we add new constants for the new $token$ values (changes are indicated with a vertical bar in the margin). We rename SYM_RESOLVED to SYM_FN and SYM_UNRESOLVED to SYM_FORWARD_FN since these values are now only used for functions – labels no

longer use symbols. Finally, we add two new symbol *kind* values, SYM_CONST and SYM_STATIC, for the symbols defined with the `const` *X* ... and `static` *X* ... syntax. The value of a symbol is now defined as follows:

- for SYM_FN symbols: the `call` instruction argument to call the function.
- for SYM_FORWARD_FN symbols: the address of the last placeholder.
- for SYM_VARIABLE symbols: the stack frame slot index of the argument or variable.
- for SYM_CONST symbols: the numeric value of the constant.
- for SYM_STATIC symbols: the *dst* address of the first byte of data.

```
const TC_INTEGER = 2;
const TC_IDENTIFIER = 3;
const TC_ADD = 4;
const TC_SUB = 5;
const TC_MUL = 6;
const TC_DIV = 7;
const TC_BIT_AND = 8;
const TC_BIT_OR = 9;
const TC_SHIFT_LEFT = 10;
const TC_SHIFT_RIGHT = 11;
const TC_LT = 12;
const TC_GE = 17;
const TC_AND = 18;
const TC_OR = 19;
const TC_BREAK = 128;
const TC_CONST = 129;
const TC_ELSE = 130;
const TC_FN = 131;
const TC_IF = 132;
const TC_LET = 133;
const TC_LOOP = 134;
const TC_RETURN = 135;
const TC_STATIC = 136;
const TC_WHILE = 137;

const SYM_FN = 0;
const SYM_FORWARD_FN = 1;
const SYM_VARIABLE = 2;
const SYM_CONST = 3;
const SYM_STATIC = 4;

const sym_name = 0;
const sym_length = 4;
const sym_kind = 8;
const sym_value = 12;
const sym_next = 16;
const sizeof_symbol = 20;
```

```
const tc_src = 0;
const tc_src_end = 4;
const tc_next_char = 8;
const tc_next_char_type = 12;
const tc_next_token = 16;
const tc_next_token_data = 20;
const tc_next_token_length = 24;
const tc_dst = 28;
const tc_heap = 32;
const tc_symbols = 36;
const tc_flash_offset = 40;
const tc_fn_dst = 44;
const sizeof_compiler = 48;
```

18.3.2 Scanner

The CHAR_TYPES and KEYWORDS tables are updated to take the new tokens into account, and a new OPERATOR table is introduced, corresponding to the one in Section 18.2.1 (each column is stored one after the other). Note that the new grammar now requires commas between each value.

```
static TC_CHAR_TYPES = [
  1,1,1,1,1,1,1,1,1,32,32,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  32,10,1,1,1,1,11,39,40,41,6,4,44,5,1,7,2,2,2,2,2,2,2,2,2,2,58,59,12,13,
  14,1,1,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,91,1,93,1,3,
  1,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,123,15,125,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1];
static TC_OPERATORS = [1,1,16,8,18,1,12,10,15,61,13,13,14,11,17,9,19,1];
static TC_KEYWORDS = [
  5,'b','r','e','a','k',128,
  5,'c','o','n','s','t',129,
  4,'e','l','s','e',130,
  2,'f','n',131,
  2,'i','f',132,
  3,'l','e','t',133,
  4,'l','o','o','p',134,
  6,'r','e','t','u','r','n',135,
  6,'s','t','a','t','i','c',136,
  5,'w','h','i','l','e',137,
  0];
```

The first scanner functions are unchanged compared with the expressions compiler:

```
fn mem_compare(ptr1, ptr2, size) {
  let i = 0;
```

```
  :step2while i, < size ifge step4;&& load8(ptr1 + i), == load8(ptr2 + i) ≀
                                                                  ≀ifne step4;{
    i = i + 1 set i;
  goto step2;}
  :step4return size - i retv;
}

fn tc_get_keyword(start, length) {
  let len = 0;
  let ptr = TC_KEYWORDS;
  :step2loop {
    len = load8(ptr) set len;
    if len, == 0 ifne step4;{ return TC_IDENTIFIER retv; }
    :step4if length, == len ifne step7;&& mem_compare(start, ptr + 1, leng≀
                                                          ≀th), == 0 ifne step7;{
      return load8(ptr + len + 1) retv;
    }
    :step7ptr = ptr + len + 2 set ptr;
  goto step2;}
}

fn tc_read_char(self) {
  let src = *(self+tc_src);
  let src_end = *(self+tc_src_end);
  if src, >= src_end iflt step2;{ panic(10); }
  :step2src = src + 1 set src;
  let c = 0;
  let type = 0;
  if src, < src_end ifge end;{
    c = load8(src) set c;
    type = load8(TC_CHAR_TYPES + c) set type;
  }
  :end*(self+tc_src), = src store;
  *(self+tc_next_char), = c store;
  *(self+tc_next_char_type), = type store;
  return type retv;
}

fn tc_read_integer(self) {
  let type = *(self+tc_next_char_type);
  let v = 0;
  :step2while type, == TC_INTEGER ifne step5;{
    v = v * 10 + (*(self+tc_next_char) - '0') set v;
    type = tc_read_char(self) set type;
  goto step2;}
  :step5*(self+tc_next_token_data), = v store;
  return TC_INTEGER retv;
```

```
}

fn tc_read_quoted_char(self) {
  tc_read_char(self) pop;
  let value = *(self+tc_next_char);
  if value, < 32 iflt not_printable;|| value, >= 127 iflt printable;{ :not⸮
                                                          ⸮_printablepanic(11); }
  :printableif tc_read_char(self), != ''' ifeq ok;{ panic(12); }
  :oktc_read_char(self) pop;
  *(self+tc_next_token_data), = value store;
  return TC_INTEGER retv;
}

fn tc_read_identifier(self) {
  let start = *(self+tc_src);
  let type = *(self+tc_next_char_type);
  :step2while type, == TC_IDENTIFIER ifeq step3;|| type, == TC_INTEGER ifn⸮
                                                                ⸮e step4;{
    :step3type = tc_read_char(self) set type;
  goto step2;}
  :step4let length = *(self+tc_src) - start;
  *(self+tc_next_token_data), = start store;
  *(self+tc_next_token_length), = length store;
  return tc_get_keyword(start, length) retv;
}
```

The next function implements the algorithm in Section 18.2.1. Note that, since the OPERATORS table is stored one column after the other, the value of its $(column, row)$ cell is the byte at address OPERATORS$+3.column + row$. We then call this function in tc_read_token, if the character type is between 10 and 20 (we only use types 10 to 15, but reserve 5 more for future use):

```
fn tc_read_operator(self, first_char_type) {
  let second_char_type = tc_read_char(self);
  let index = 3 * (first_char_type - 10);
  if second_char_type, == first_char_type ifne step1;{
    tc_read_char(self) pop;
    index = index + 1 set index;
  goto end;} else :step1if *(self+tc_next_char), == '=' ifne end;{
    tc_read_char(self) pop;
    index = index + 2 set index;
  }
  :endreturn load8(TC_OPERATORS + index) retv;
}

fn tc_read_token(self) {
  let type = *(self+tc_next_char_type);
  :step1while type, == ' ' ifne step3;{
```

```
    type = tc_read_char(self) set type;
  goto step1;}
  :step3let token = type;
  if type, == TC_INTEGER ifne step4;{
    token = tc_read_integer(self) set token;
  goto end;} else :step4if type, == ''' ifne step5;{
    token = tc_read_quoted_char(self) set token;
  goto end;} else :step5if type, == TC_IDENTIFIER ifne step6;{
    token = tc_read_identifier(self) set token;
  goto end;} else :step6if type, >= 10 iflt step7;&& type, < 20 ifge step7⸮
                                                                    ⸮;{
    token = tc_read_operator(self, type) set token;
  goto end;} else :step7if type, != 0 ifeq end;{
    tc_read_char(self) pop;
  }
  :end*(self+tc_next_token), = token store;
  retreturn;
}
```

18.3.3 Backend

The backend is extended with new functions to write the new opcode instructions needed by the parser. Its first functions are the same as in the expressions compiler:

```
fn mem_allocate(size, ptr_p) {
  let ptr = *ptr_p;
  *ptr_p, = ptr + size store;
  return ptr retv;
}
fn tc_write8(self, value) {
  store8(mem_allocate(1, self+tc_dst), value);
  retreturn;
}
fn tc_write16(self, value) {
  store16(mem_allocate(2, self+tc_dst), value);
  retreturn;
}
fn tc_write32(self, value) {
  *mem_allocate(4, self+tc_dst), = value store;
  retreturn;
}
fn tc_write_insn(self, opcode, argument) {
  tc_write8(self, opcode);
  tc_write8(self, argument);
  retreturn;
}
```

The `tc_add_placeholder` function is updated to write the placeholder instead of just returning its value, and is renamed accordingly. The next function is unchanged. A

new `tc_fill_label_placeholders` function fills the placeholders in the list starting at *placeholder*, with the current instruction offset $dst - fn\_dst$ as value:

```
fn tc_write_placeholder(self, placeholder_p) {
  let new_placeholder = *(self+tc_dst);
  let last_placeholder = *placeholder_p;
  *placeholder_p, = new_placeholder store;
  if last_placeholder, == 0 ifne step4;{ last_placeholder = new_placeholde⸮
                                                ⸮r set last_placeholder; }
  :step4tc_write16(self, new_placeholder - last_placeholder);
  retreturn;
}
fn tc_fill_placeholders(placeholder, value) {
  let offset = 0;
  :step2while placeholder, != 0 ifeq end;{
    :step3offset = load16(placeholder) set offset;
    store16(placeholder, value);
    if offset, == 0 ifne step6;{ retbreak; }
    :step6placeholder = placeholder - offset set placeholder;
  goto step2;}
  :endretreturn;
}
fn tc_fill_label_placeholders(self, placeholder) {
  tc_fill_placeholders(placeholder, *(self+tc_dst) - *(self+tc_fn_dst));
  retreturn;
}
```

The remaining functions write bytecode instructions and encapsulate the details of their encoding. We only comment the new ones which are not completely trivial.

```
fn tc_write_cst_insn(self, value) {
  if value, <= 1 ifgt not0_or_1;{
    tc_write8(self, value);
  ret;} else :not0_or_1if value, < 256 ifge not_byte;{
    tc_write_insn(self, 2, value);
  ret;} else :not_byte{
    tc_write8(self, 3);
    tc_write32(self, value);
  }
  retreturn;
}
```

The next function writes the instruction to push on the stack the final address of some `static` data. This is its *dst* address minus *flash_offset*.

```
fn tc_write_static_insn(self, dst) {
  tc_write_cst_insn(self, dst - *(self+tc_flash_offset));
  retreturn;
}
fn tc_write_binary_insn(self, token) {
```

```
  tc_write8(self, token);
  retreturn;
}
```

In the following function, *token* must be between `TC_LT` and `TC_GE` (included). The next two functions write a goto instruction for a forward or backward jump.

```
fn tc_write_jump_insn(self, token, placeholder_p) {
  tc_write8(self, token);
  tc_write_placeholder(self, placeholder_p);
  retreturn;
}
fn tc_write_goto_insn(self, placeholder_p) {
  tc_write_jump_insn(self, 18, placeholder_p);
  retreturn;
}
fn tc_write_loop_insn(self, loop_dst) {
  tc_write8(self, 18);
  tc_write16(self, loop_dst - *(self+tc_fn_dst));
  retreturn;
}
fn tc_write_load_insn(self) {
  tc_write8(self, 19);
  retreturn;
}
```

The next function erases the last written instruction, supposed to be a `load`. It is used for the algorithm in Section 18.2.2.

```
fn tc_erase_load_insn(self) {
  *(self+tc_dst), = *(self+tc_dst) - 1 store;
  retreturn;
}
fn tc_write_store_insn(self) {
  tc_write8(self, 20);
  retreturn;
}
fn tc_write_ptr_insn(self, variable) {
  tc_write_insn(self, 21, variable);
  retreturn;
}
fn tc_write_get_insn(self, variable) {
  tc_write_insn(self, 22, variable);
  retreturn;
}
```

Similarly, the following function erases the last written instruction, supposed to be a `get`. It returns the argument of this instruction.

```
fn tc_erase_get_insn(self) {
  let dst = *(self+tc_dst);
```

```
  *(self+tc_dst), = dst - 2 store;
  return load8(dst - 1) retv;
}
fn tc_write_set_insn(self, variable) {
  tc_write_insn(self, 23, variable);
  retreturn;
}
fn tc_write_pop_insn(self) {
  tc_write8(self, 24);
  retreturn;
}
fn tc_write_fn_insn(self, arity) {
  tc_write_insn(self, 25, arity);
  retreturn;
}
fn tc_get_fn_value(self, fn_dst) {
  return fn_dst - *(self+tc_flash_offset) - 786432 retv;
}
fn tc_write_call_insn(self, function) {
  tc_write8(self, 26);
  if *(function+sym_kind), == SYM_FORWARD_FN ifne resolved;{
    tc_write_placeholder(self, function+sym_value);
  ret;} else :resolved{
    tc_write16(self, *(function+sym_value));
  }
  retreturn;
}
fn tc_write_return_insn(self) {
  tc_write8(self, 30);
  retreturn;
}
```

18.3.4 Parser

The parser starts with the same utility functions as before, mostly unchanged (we rename `tc_add_or_resolve_symbol` to `tc_add_or_resolve_fn_symbol` since it is now only used for functions).

```
fn sym_lookup(symbol, name, length) {
  :step2while symbol, != 0 ifeq step7;{
    if *(symbol+sym_length), == length ifne step6;&&
        mem_compare(*(symbol+sym_name), name, length), == 0 ifne step6;{
      return symbol retv;
    }
    :step6symbol = *(symbol+sym_next) set symbol;
  goto step2;}
  :step7return 0 retv;
```

```
}

fn tc_add_symbol(self, name, length, kind, value) {
  let symbol = mem_allocate(sizeof_symbol, self+tc_heap);
  if sym_lookup(*(self+tc_symbols), name, length), != 0 ifeq ok;{ panic(30⸮
                                                                    ⸮); }
  :ok*(symbol+sym_name), = name store;
  *(symbol+sym_length), = length store;
  *(symbol+sym_kind), = kind store;
  *(symbol+sym_value), = value store;
  *(symbol+sym_next), = *(self+tc_symbols) store;
  *(self+tc_symbols), = symbol store;
  return symbol retv;
}

fn tc_add_or_resolve_fn_symbol(self, name, length, value) {
  let symbol = sym_lookup(*(self+tc_symbols), name, length);
  if symbol, == 0 ifne found;{
    return tc_add_symbol(self, name, length, SYM_FN, value) retv;
  }
  :foundif *(symbol+sym_kind), != SYM_FORWARD_FN ifeq ok;{ panic(31); }
  :oktc_fill_placeholders(*(symbol+sym_value), value);
  *(symbol+sym_kind), = SYM_FN store;
  *(symbol+sym_value), = value store;
  return symbol retv;
}

fn tc_parse_token(self, token) {
  if *(self+tc_next_token), != token ifeq ok;{ panic(20); }
  :oktc_read_token(self);
  retreturn;
}

fn tc_parse_integer(self) {
  if *(self+tc_next_token), != TC_INTEGER ifeq ok;{ panic(21); }
  :oklet value = *(self+tc_next_token_data);
  tc_read_token(self);
  return value retv;
}

fn tc_parse_identifier(self, length_p) {
  if *(self+tc_next_token), != TC_IDENTIFIER ifeq ok;{ panic(22); }
  :oklet name = *(self+tc_next_token_data);
  *length_p, = *(self+tc_next_token_length) store;
  tc_read_token(self);
  return name retv;
}
```

```
fn tc_parse_symbol(self, symbol) {
  let length = 0;
  let name = tc_parse_identifier(self, &length);
  symbol = sym_lookup(symbol, name, length) set symbol;
  if symbol, == 0 ifne ok;{ panic(33); }
  :okreturn symbol retv;
}
```

The `tc_parse_const` and `tc_parse_static` functions are updated in a trivial way to match the new "const" and "static" rules (and to use the new `CONST` and `STATIC` symbol kinds).

```
fn tc_parse_const(self) {
  tc_parse_token(self, TC_CONST);
  let length = 0;
  let name = tc_parse_identifier(self, &length);
  tc_parse_token(self, '=');
  tc_add_symbol(self, name, length, SYM_CONST, tc_parse_integer(self)) pop;
  tc_parse_token(self, ';');
  retreturn;
}
fn tc_parse_static(self) {
  tc_parse_token(self, TC_STATIC);
  let length = 0;
  let name = tc_parse_identifier(self, &length);
  tc_add_symbol(self, name, length, SYM_STATIC, *(self+tc_dst)) pop;
  tc_parse_token(self, '=');
  tc_parse_token(self, '[');
  tc_write8(self, tc_parse_integer(self));
  :loopwhile *(self+tc_next_token), == ',' ifne end;{
    tc_read_token(self);
    tc_write8(self, tc_parse_integer(self));
  goto loop;}
  :endtc_parse_token(self, ']');
  tc_parse_token(self, ';');
  retreturn;
}
```

Here we delete the old `tc_parse_argument` and `tc_parse_label` functions, and add constants for the possible origin of the value of an expression (see Section 18.2.2). We update the next function to test the symbol kind in a better way, and to return `OTHER` (note also the pop instructions, since `tc_parse_expr` now returns an origin):

```
const FROM_ADDRESS = 0;
const FROM_VARIABLE = 1;
const FROM_OTHER = 255;

fn tc_parse_expr(self);
```

```
fn tc_parse_fn_arguments(self, function) {
  if *(function+sym_kind), != SYM_FN ifeq ok;&& *(function+sym_kind), != S⸮
                                                  ⸮YM_FORWARD_FN ifeq ok;{
    panic(34);
  }
  :oktc_parse_token(self, '(');
  if *(self+tc_next_token), != ')' ifeq end;{
    tc_parse_expr(self) pop;
    :loopwhile *(self+tc_next_token), == ',' ifne end;{
      tc_read_token(self);
      tc_parse_expr(self) pop;
    goto loop;}
  }
  :endtc_parse_token(self, ')');
  tc_write_call_insn(self, function);
  return FROM_OTHER retv;
}
```

Parsing a primitive expression is done as before, but we must now return the origin of the expression value. This is `VARIABLE` for symbols referring to function parameter or local variable names, `OTHER` for `const` and `static` symbols or integer constants, or the origin returned by the function called to handle the other cases.

```
fn tc_parse_primitive_expr(self) {
  let origin = FROM_OTHER;
  let symbol = 0;
  if *(self+tc_next_token), == TC_INTEGER ifne not_integer;{
    tc_write_cst_insn(self, tc_parse_integer(self));
  goto end;} else :not_integerif *(self+tc_next_token), == TC_IDENTIFIER i⸮
                                                        ⸮fne parentheses;{
    symbol = tc_parse_symbol(self, *(self+tc_symbols)) set symbol;
    if *(self+tc_next_token), == '(' ifne identifier;{
      origin = tc_parse_fn_arguments(self, symbol) set origin;
    goto end;} else :identifier{
      if *(symbol+sym_kind), == SYM_VARIABLE ifne not_variable;{
        tc_write_get_insn(self, *(symbol+sym_value));
        origin = FROM_VARIABLE set origin;
      goto end;} else :not_variableif *(symbol+sym_kind), == SYM_CONST ifn⸮
                                                            ⸮e not_const;{
        tc_write_cst_insn(self, *(symbol+sym_value));
      goto end;} else :not_constif *(symbol+sym_kind), == SYM_STATIC ifne ⸮
                                                                ⸮error;{
        tc_write_static_insn(self, *(symbol+sym_value));
      goto end;} else :error{
        panic(35);
      }
    }
  } else :parentheses{
```

```
    tc_parse_token(self, '(');
    origin = tc_parse_expr(self) set origin;
    tc_parse_token(self, ')');
  }
  :endreturn origin retv;
}
```

Similarly, parsing a pointer expression is unchanged, but we must now return ADDRESS for dereference expressions "$*e$" and OTHER for address-of expressions "&x":

```
fn tc_parse_pointer_expr(self) {
  let symbol = 0;
  if *(self+tc_next_token), == TC_MUL ifne not_mul;{
    tc_read_token(self);
    tc_parse_pointer_expr(self) pop;
    tc_write_load_insn(self);
    return FROM_ADDRESS retv;
  } else :not_mulif *(self+tc_next_token), == TC_BIT_AND ifne not_bit_and;≀
                                                                        ≀{
    tc_read_token(self);
    symbol = tc_parse_symbol(self, *(self+tc_symbols)) set symbol;
    if *(symbol+sym_kind), == SYM_VARIABLE ifne error;{
      tc_write_ptr_insn(self, *(symbol+sym_value));
    goto end;} else :error{
      panic(36);
    }
  } else :not_bit_and{
    return tc_parse_primitive_expr(self) retv;
  }
  :endreturn FROM_OTHER retv;
}
```

The remaining expression parsing functions are also updated to return the correct origin. This is done with the same method for all of them: the origin of an expression $e_1\ op\ e_2\ op\ \ldots\ e_n$ is the origin of e_1 if $n = 1$, and OTHER otherwise. We also add a new function to parse shift expressions, very similar to the others:

```
fn tc_parse_mult_expr(self) {
  let origin = tc_parse_pointer_expr(self);
  let next_token = *(self+tc_next_token);
  :loopwhile next_token, == TC_MUL ifeq mul_or_div;|| next_token, == TC_DI≀
                                                              ≀V ifne end;{
    :mul_or_divtc_read_token(self);
    tc_parse_pointer_expr(self) pop;
    origin = FROM_OTHER set origin;
    tc_write_binary_insn(self, next_token);
    next_token = *(self+tc_next_token) set next_token;
  goto loop;}
  :endreturn origin retv;
```

```
}

fn tc_parse_add_expr(self) {
  let origin = tc_parse_mult_expr(self);
  let next_token = *(self+tc_next_token);
  :loopwhile next_token, == TC_ADD ifeq add_or_sub;|| next_token, == TC_SU⌇
                                                          ⌇B ifne end;{
    :add_or_subtc_read_token(self);
    tc_parse_mult_expr(self) pop;
    origin = FROM_OTHER set origin;
    tc_write_binary_insn(self, next_token);
    next_token = *(self+tc_next_token) set next_token;
  goto loop;}
  :endreturn origin retv;
}

fn tc_parse_shift_expr(self) {
  let origin = tc_parse_add_expr(self);
  let next_token = *(self+tc_next_token);
  if next_token, == TC_SHIFT_LEFT ifeq shift;|| next_token, == TC_SHIFT_RI⌇
                                                        ⌇GHT ifne end;{
    :shifttc_read_token(self);
    tc_parse_add_expr(self) pop;
    origin = FROM_OTHER set origin;
    tc_write_binary_insn(self, next_token);
  }
  :endreturn origin retv;
}

fn tc_parse_bit_and_expr(self) {
  let origin = tc_parse_shift_expr(self);
  :loopwhile *(self+tc_next_token), == TC_BIT_AND ifne end;{
    tc_read_token(self);
    tc_parse_shift_expr(self) pop;
    origin = FROM_OTHER set origin;
    tc_write_binary_insn(self, TC_BIT_AND);
  goto loop;}
  :endreturn origin retv;
}

fn tc_parse_expr(self) {
  let origin = tc_parse_bit_and_expr(self);
  :loopwhile *(self+tc_next_token), == TC_BIT_OR ifne end;{
    tc_read_token(self);
    tc_parse_bit_and_expr(self) pop;
    origin = FROM_OTHER set origin;
    tc_write_binary_insn(self, TC_BIT_OR);
```

```
  goto loop;}
  :endreturn origin retv;
}
```

The following functions are the main new part of the compiler. They implement the grammar rules for statements and for the new boolean expressions, as described in Section 18.2.2. Parsing a comparison expression is simple: we just need to parse the two subexpressions, check that the token in between is actually a comparison operator, and return it:

```
fn tc_parse_comparison_expr(self) {
  tc_parse_expr(self) pop;
  let token = *(self+tc_next_token);
  if token, < TC_LT iflt error;|| token, > TC_GE ifle ok;{ :errorpanic(25)⌇
                                                          ⌇; }
  :oktc_read_token(self);
  tc_parse_expr(self) pop;
  return token retv;
}
```

As explained in Section 18.2.2, the function parsing a conjunction c_1 && c_2 && ... c_n takes as parameter the label where to jump if this expression is false. This parameter is actually the address of some local variable containing the label's list of placeholders. It is named `else_refs_p` for brevity (for "*p*ointer to a list of placeholders for forward *ref*erences to an *else* label"). In order to generate a jump to this label when each c_i except the last is false, we write a jump instruction for the *opposite* operator of c_i when a "&&" token is read. Note that the opposite operators of "<", "==", ">", "<=", "!=", and ">=" are the same operators in reverse order. Since their $token$ value, equal to their opcode, are 12, 13, 14, 15, 16, and 17, respectively, the opposite of $token$ is simply $12 + 17 - token$. This gives the following function, returning c_n's $token$:

```
fn tc_parse_and_expr(self, else_refs_p) {
  let token = tc_parse_comparison_expr(self);
  :loopwhile *(self+tc_next_token), == TC_AND ifne end;{
    tc_read_token(self);
    tc_write_jump_insn(self, TC_LT + TC_GE - token, else_refs_p);
    token = tc_parse_comparison_expr(self) set token;
  goto loop;}
  :endreturn token retv;
}
```

The function parsing a disjunction c_1 || c_2 || ... c_n works in a similar way. It is a bit longer because it creates a label l_i where each c_i should jump if false, in the `else_refs` local variable. Resetting this variable to 0 in the loop effectively discards l_i, no longer needed, and "creates" l_{i+1}. Returning this variable at the end returns l_n, as described in Section 18.2.2:

```
fn tc_parse_boolean_expr(self, then_refs_p) {
  let else_refs = 0;
```

```
  let token = tc_parse_and_expr(self, &else_refs);
  :loopwhile *(self+tc_next_token), == TC_OR ifne end;{
    tc_read_token(self);
    tc_write_jump_insn(self, token, then_refs_p);
    tc_fill_label_placeholders(self, else_refs);
    else_refs = 0 set else_refs;
    token = tc_parse_and_expr(self, &else_refs) set token;
  goto loop;}
  :endtc_write_jump_insn(self, TC_LT + TC_GE - token, &else_refs);
  return else_refs retv;
}
```

We can now implement the functions parsing statements, one for each grammar rule. We implement them in reverse order, starting with "block_stmt". This rule uses "stmt" but since `tc_parse_stmt` is implemented last, we need to declare it first. The `tc_parse_stmt` function takes as parameter the label where any nested "`break`" statement should jump. It is equal to 0 when parsing a statement which is not inside a loop. Like all the statement parsing functions, it returns one of the following constants:

```
const END_UNREACHABLE = 0;
const END_REACHABLE = 1;
```

```
fn tc_parse_stmt(self, break_refs_p);
```

Parsing a "{ s_1; ... s_n; }" block is very simple. As described in Section 18.2.2, we return an error if any s_i is unreachable, and return the reachability of s_n's end:

```
fn tc_parse_block_stmt(self, break_refs_p) {
  let state = END_REACHABLE;
  tc_parse_token(self, '{');
  :loopwhile *(self+tc_next_token), != '}' ifeq end;{
    if state, == END_UNREACHABLE ifne ok;{ panic(37); }
    :okstate = tc_parse_stmt(self, break_refs_p) set state;
  goto loop;}
  :endtc_read_token(self);
  return state retv;
}
```

The following function parses and compiles the right hand side of an assignment statement. As described in Section 18.2.2, it detects invalid assignments, erases the unwanted code written while compiling the left hand side, compiles the right hand side, and finally writes the correct assignment instruction:

```
fn tc_parse_assignment(self, origin) {
  let location = 0;
  if origin, == FROM_ADDRESS ifne not_address;{
    tc_erase_load_insn(self);
  goto ok;} else :not_addressif origin, == FROM_VARIABLE ifne error;{
    location = tc_erase_get_insn(self) set location;
  goto ok;} else :error{
```

```
    panic(38);
  }
  :oktc_parse_token(self, '=');
  tc_parse_expr(self) pop;
  if origin, == FROM_ADDRESS ifne variable;{
    tc_write_store_insn(self);
  goto end;} else :variable{
    tc_write_set_insn(self, location);
  }
  :endreturn END_REACHABLE retv;
}
```

With this function, parsing an expression or assignment statement is simple. Parsing a return or a break statement is simple too. As explained in Section 18.1.2, a "`return;`" is actually compiled as a "`return 0;`" so that all expressions return a value (the pop instruction written below with `tc_write_pop_insn` requires this):

```
fn tc_parse_expr_or_assign_stmt(self) {
  let origin = tc_parse_expr(self);
  if *(self+tc_next_token), == '=' ifne not_assign;{
    tc_parse_assignment(self, origin) pop;
  goto end;} else :not_assign{
    tc_write_pop_insn(self);
  }
  :endtc_parse_token(self, ';');
  return END_REACHABLE retv;
}
```

```
fn tc_parse_return_stmt(self) {
  tc_parse_token(self, TC_RETURN);
  if *(self+tc_next_token), == ';' ifne not_semicolon;{
    tc_write_cst_insn(self, 0);
  goto end;} else :not_semicolon{
    tc_parse_expr(self) pop;
  }
  :endtc_parse_token(self, ';');
  tc_write_return_insn(self);
  return END_UNREACHABLE retv;
}
```

```
fn tc_parse_break_stmt(self, break_refs_p) {
  tc_parse_token(self, TC_BREAK);
  tc_parse_token(self, ';');
  tc_write_goto_insn(self, break_refs_p);
  return END_UNREACHABLE retv;
}
```

The next function parses a "`while`" or "`loop`" statement (the caller must make sure that the next token is either "`while`" or "`loop`"). It does this with two *body* and

end labels managed as described in Section 18.2.2. Note that the *end* label is passed as a `breaks_refs` argument to parse the loop's body, since this is where "`break`" statement inside this loop should jump. The end of a "`loop`" is unreachable if its body does not contain a "`break`", which is the case if the *end* label has no placeholder.

```
fn tc_parse_while_or_loop_stmt(self) {
  let loop_dst = *(self+tc_dst);
  let body_refs = 0;
  let end_refs = 0;
  let token = *(self+tc_next_token);
  tc_read_token(self);
  if token, == TC_WHILE ifne body;{
    end_refs = tc_parse_boolean_expr(self, &body_refs) set end_refs;
  }
  :bodytc_fill_label_placeholders(self, body_refs);
  tc_parse_block_stmt(self, &end_refs) pop;
  tc_write_loop_insn(self, loop_dst);
  :end_iftc_fill_label_placeholders(self, end_refs);
  if token, == TC_LOOP ifne end;&& end_refs, == 0 ifne end;{ return END_UN⸮
                                                          ⸮REACHABLE retv; }
  :endreturn END_REACHABLE retv;
}
```

Similary, parsing an "`if`" statement is done with 3 *then*, *else*, and *end* labels managed as described in Section 18.2.2. The "`state = state | ...`" pattern ensures the reachability rule for "`if`" statements (cf. Section 18.2.2) because `END_UNREACHABLE` and `END_REACHABLE` are 0 and 1, respectively.

```
fn tc_parse_if_stmt(self, break_refs_p) {
  tc_parse_token(self, TC_IF);
  let then_refs = 0;
  let else_refs = tc_parse_boolean_expr(self, &then_refs);
  tc_fill_label_placeholders(self, then_refs);
  let state = tc_parse_block_stmt(self, break_refs_p);
  let end_if_refs = 0;
  if *(self+tc_next_token), == TC_ELSE ifne not_else;{
    tc_read_token(self);
    if state, == END_REACHABLE ifne else;{
      tc_write_goto_insn(self, &end_if_refs);
    }
    :elsetc_fill_label_placeholders(self, else_refs);
    if *(self+tc_next_token), == '{' ifne not_block;{
      state = state | tc_parse_block_stmt(self, break_refs_p) set state;
    goto end_if;} else :not_block{
      state = state | tc_parse_if_stmt(self, break_refs_p) set state;
    }
    :end_iftc_fill_label_placeholders(self, end_if_refs);
  goto end;} else :not_else{
    tc_fill_label_placeholders(self, else_refs);
```

```
    state = END_REACHABLE set state;
  }
  :endreturn state retv;
}
```

Finally, parsing an arbitrary statement is done by calling one of the above functions, depending on the next token value:

```
fn tc_parse_stmt(self, break_refs_p) {
  if *(self+tc_next_token), == TC_IF ifne step2;{
    return tc_parse_if_stmt(self, break_refs_p) retv;
  } else :step2if *(self+tc_next_token), == TC_WHILE ifne step3;{
    return tc_parse_while_or_loop_stmt(self) retv;
  } else :step3if *(self+tc_next_token), == TC_LOOP ifne step4;{
    return tc_parse_while_or_loop_stmt(self) retv;
  } else :step4if *(self+tc_next_token), == TC_BREAK ifne step6;{
    if break_refs_p, == 0 ifne ok;{ panic(39); }
    :okreturn tc_parse_break_stmt(self, break_refs_p) retv;
  } else :step6if *(self+tc_next_token), == TC_RETURN ifne end;{
    return tc_parse_return_stmt(self) retv;
  }
  :endreturn tc_parse_expr_or_assign_stmt(self) retv;
}
```

The next 3 functions are essentially unchanged compared with the expressions compiler, besides an additional call in `tc_parse_let_stmt` to parse the "=" token now required after the identifier:

```
fn tc_parse_let_stmt(self, variable) {
  tc_parse_token(self, TC_LET);
  let length = 0;
  let name = tc_parse_identifier(self, &length);
  tc_parse_token(self, '=');
  tc_parse_expr(self) pop;
  tc_parse_token(self, ';');
  tc_add_symbol(self, name, length, SYM_VARIABLE, variable) pop;
  return variable + 1 retv;
}

fn tc_parse_fn_name(self) {
  let length = 0;
  let name = tc_parse_identifier(self, &length);
  let fn_dst = *(self+tc_dst);
  *(self+tc_fn_dst), = fn_dst store;
  let value = tc_get_fn_value(self, fn_dst);
  return tc_add_or_resolve_fn_symbol(self, name, length, value) retv;
}

fn tc_parse_fn_parameters(self) {
```

```
  let i = 0;
  let name = 0;
  let length = 0;
  tc_parse_token(self, '(');
  :loopwhile *(self+tc_next_token), != ')' ifeq end;{
    if i, > 0 ifle identifier;{ tc_parse_token(self, ','); }
    :identifiername = tc_parse_identifier(self, &length) set name;
    tc_add_symbol(self, name, length, SYM_VARIABLE, i) pop;
    i = i + 1 set i;
  goto loop;}
  :endtc_read_token(self);
  return i retv;
}
```

The function to parse a function body is extended to allow "`let`" statements, as defined by the new "fn_body" rule. It is also updated to check that all the statements are reachable, and to return an error if the end of the last one might be reachable.

```
fn tc_parse_fn_body(self, function, arity) {
  if *(self+tc_next_token), == ';' ifne body;{
    tc_read_token(self);
    *(function+sym_kind), = SYM_FORWARD_FN store;
    *(function+sym_value), = 0 store;
    retreturn;
  }
  :bodytc_parse_token(self, '{');
  tc_write_fn_insn(self, arity);
  let next_variable = arity + 4;
  let state = END_REACHABLE;
  :loopwhile *(self+tc_next_token), != '}' ifeq end;{
    if state, == END_UNREACHABLE ifne ok;{ panic(40); }
    :okif *(self+tc_next_token), == TC_LET ifne stmt;{
      next_variable = tc_parse_let_stmt(self, next_variable) set next_vari⟅
                                                                ⟅able;
    goto loop;} else :stmt{
      state = tc_parse_stmt(self, 0) set state;
    }
  goto loop;}
  :endif state, == END_REACHABLE ifne valid;{ panic(41); }
  :validtc_read_token(self);
  retreturn;
}
```

The next function is unchanged, but is no longer called from `tc_parse_fn`, otherwise unchanged too, since programs no longer use labels. `tc_parse_program` is also unchanged.

```
fn tc_check_symbols(symbol, end_symbol) {
  :loopwhile symbol, != end_symbol ifeq end;{
```

```
    if *(symbol+sym_kind), == SYM_FORWARD_FN ifne next;{ panic(32); }
    :nextsymbol = *(symbol+sym_next) set symbol;
  goto loop;}
  :endretreturn;
}

fn tc_parse_fn(self) {
  tc_parse_token(self, TC_FN);
  let function = tc_parse_fn_name(self);
  let heap = *(self+tc_heap);
  let symbols = *(self+tc_symbols);
  let arity = tc_parse_fn_parameters(self);
  tc_parse_fn_body(self, function, arity);
  *(self+tc_symbols), = symbols store;
  *(self+tc_heap), = heap store;
  retreturn;
}

fn tc_parse_program(self) {
  :looploop {
    if *(self+tc_next_token), == TC_FN ifne not_fn;{
      tc_parse_fn(self);
    goto loop;} else :not_fnif *(self+tc_next_token), == TC_STATIC ifne no⌇
                                                              ⌇t_static;{
      tc_parse_static(self);
    goto loop;} else :not_staticif *(self+tc_next_token), == TC_CONST ifne⌇
                                                                  ⌇ end;{
      tc_parse_const(self);
    goto loop;} else :end{
      if *(self+tc_next_token), != 0 ifeq ok;{ panic(23); }
      :oktc_check_symbols(*(self+tc_symbols), 0);
      retreturn;
    }
  }
}
```

Finally, the `tc_main` function is refactored to store the compiler variables between the *dst* buffer and the *heap* (instead of on the stack). The advantage of this method is that we can skip the initialization of the variables which don't need to be.

```
fn tc_main(src_buffer, dst_buffer, flash_buffer) {
  let error = 0;
  let compiler = dst_buffer + 12288;
  *(compiler+tc_src), = src_buffer + 3 store;
  *(compiler+tc_src_end), = src_buffer + 4 + *src_buffer store;
  *(compiler+tc_dst), = dst_buffer + 4 store;
  *(compiler+tc_heap), = compiler + sizeof_compiler store;
  *(compiler+tc_symbols), = 0 store;
  *(compiler+tc_flash_offset), = dst_buffer - flash_buffer store;
```

```
  let panic3 = 0;
  let panic2 = 0;
  let panic1 = 0;
  let panic0 = 0;
  error = panic_result(&panic0) set error;
  if error, != 0 ifeq ok;{
    *dst_buffer, = *(compiler+tc_src) - src_buffer - 4 store;
    return error retv;
  }
  :oktc_read_char(compiler) pop;
  tc_read_token(compiler);
  tc_parse_program(compiler);
  *dst_buffer, = *(compiler+tc_dst) - dst_buffer - 4 store;
  return 0 retv;
}
```

18.4 Compilation and tests

To compile the above source code proceed as follows (see also Figure 16.4).

Edit v1 In the command editor, type "F3"+"r" and "F4"+"r" to load and edit the current compiler version. Then update it to the 1^{st} version of the statements compiler. For convenience, we also provide this code in the `statements_compiler_v1.txt` file in https://ebruneton.github.io/toypc/sources.zip. When you are done, exit the text editor and type "F5"+"r" to save your work. Alternatively, you can "cheat" by running the following command on an external computer (see Section 16.4 for more details):

```
user@host:~$ python3 flash_helper.py < part3/statements_compiler_v1.txt
```

Compile v1 In the command editor, type "F6"+"r" to compile the code you typed. If all goes well, after about 3 seconds, you should get a result equal to 0 (meaning that no error was found). If this is not the case use Appendix D to get the error code meaning, fix this error, save the program and compile it again. Repeat this process until the compilation is successful. Then type "F7"+"r" to save the result.

Test v1 Type "F2"+"r" to create a new program, "F4"+"r" to edit it, and type the following small test program, which computes the factorial of 6:

```
fn factorial(n);
fn test() { return factorial(6); }
fn factorial(n) {
  if n == 0 { return 1; }
  return factorial(n - 1) * n;
}
```

Then type "F9"+"r" to run it. If the result is not 720 = $2D0_{16}$ this means that the compiler is wrong. In this case, type "F8"+"r" to restore the expressions compiler. Then repeat the previous steps and double check everything until this test passes.

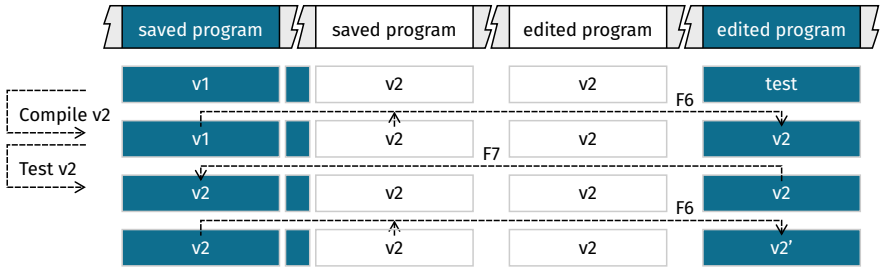


FIGURE 18.2 The memory content after each command in the "Compile v2" and "Test v2" steps. White, blue and gray areas represent source code, bytecode and unused memory, respectively (not to scale). See also Figure 16.4.

Edit v2 Type "F3"+"r" to load the 1^{st} version of the statements compiler and "F4"+"r" to edit it. Then update it to the 2^{nd} version. For convenience, we also provide this code in the `statements_compiler_v2.txt` file. Then save this new version with the F5 command. Alternatively, run the following command on an external computer:

```
user@host:~$ python3 flash_helper.py < part3/statements_compiler_v2.txt
```

Compile v2 Type"F6"+"r" to compile this new code. The result should be 0, meaning "no error". If this is not the case, repeat the "Edit v2" and "Compile v2" steps until all errors are fixed.

Test v2 Unlike in the previous chapters, the compiled code of the 2^{nd} version of the statements compiler is not identical to that of the 1^{st} version. One reason, in particular, is that we used both `ret` and `retv` instructions in the 1^{st} version, but the statements compiler only produces `retv` instructions for `return` statements (see Section 18.1.2).

However, the two versions should produce the same compiled code for the same input program, since they are supposed to be functionally equivalent. We can thus use this property to test the 2^{nd} version. For this we can use as input program the 2^{nd} version itself (see Figure 18.2). We just compiled it with the 1^{st} version in the previous step. Type "F7"+"r" to store the result in flash memory. Then type "F6"+"r" to compile the 2^{nd} version with itself. At this stage we have the code of our input program in flash memory (as compiled with the 1^{st} version), and in RAM (as compiled with the 2^{nd} version). Type "F10"+"r" to compare them. The result should be 0. If this is not the case, this means that the 2^{nd} version is wrong[1]. Type "F8"+"r" to restore the 1^{st} version and repeat the steps from "Edit v2" until this test passes.

[1]It might also happen that the 1^{st} version is wrong despite the "Test v1" step. In this case we are in trouble because we no longer have any backup of the expressions compiler (which is necessary to retry the "Compile v1" step). To avoid this we would need to do a backup of the backup compiler before "Test v2".

CHAPTER

19 Types Compiler

We can now write programs in a simple syntax, and automatically obtain the corresponding bytecode with our toy compiler. This avoids a lot of tedious and error-prone manual tasks which would otherwise be needed to convert statements and expressions into low level instructions, and to compute function addresses, instruction offsets, or stack frame slot indices. However, many errors can still be made when writing programs in our toy programming language. For instance, we may inadvertently use a wrong number of arguments in a function call. We may also pass a pointer to some value instead of the value itself, or vice versa. Or try to use the result of a function not returning any value. This chapter extends our toy programming language so that our compiler can detect the majority of these errors.

19.1 Requirements

19.1.1 Type declarations

The function parameters and local variables of our programs are all 32 bit values. However, these values represent different things depending on each parameter and local variable. For instance, a value can represent a character, a pointer to a character, a pointer to a local variable itself storing a pointer to a character, etc. Other values represent the address of a list of values, each with its own representation (*e.g.*, in our compiler, `*(self+tc_src)` is a pointer to a character, while `*(self+tc_next_char)` is a character).

Passing a character to a function expecting a pointer to a character will almost certainly lead to a crash. Storing a pointer to a character in a local variable representing a character will likely have the same result. Unfortunately this kind of error can easily happen if one forgets a "&" or "*" operator. And the compiler cannot detect them since it does not "know" what each function parameter or local variable represents. The solution is to "tell" it. For this we add a new requirement for our programming language, namely a syntax to specify what a value represents, which is called its *type*. We define one as follows.

The type of an integer value (such as a character, or a number of characters) is noted "`u32`", for "unsigned 32 bit value". By analogy with the address-of operator "&" the type of the address of an integer, *i.e.*, of a pointer to an integer, is noted "`&u32`".

The type of a pointer to an integer pointer is noted "&&u32". And so on. The type of a list of values is defined with the syntax illustrated on the following example:

```
struct Symbol {
  name: &u32,
  length: u32,
  next: &Symbol
}
```

This syntax defines a new type, here called `Symbol`. A value of this type is made of 3 consecutive words in memory, the first one representing an integer pointer, the second one an integer, and the third one a pointer to a `Symbol` value. `name`, `length` and `next` are called the *fields* of the `Symbol` *data structure*.

Thanks to this new syntax we can now require each function parameter and local variable to *declare* its type, as illustrated in the following example:

```
fn load8(ptr: &u32) -> u32 { let value:u32 = (*ptr) & 255; return value; }
```

The new elements, in green, specify that `load8` takes an integer pointer as parameter, and returns an integer. They also specify that "`value`" represents an integer. Finally, we require our compiler to use these declarations to keep track of the type of each expression, and to check that they are correctly used. For instance, knowing that `ptr` has type `&u32`, the compiler should deduce that `*ptr`, the value at address `ptr`, has type `u32`. It should then confirm that combining it with the integer constant 255 and storing the result in `value` is valid. On the other hand, it should detect that "`load8(1)`", for instance, is incorrect, since it passes an integer to a function expecting an integer pointer. We define these requirements in the next section.

Besides enabling the detection of more errors, the above requirements have two additional benefits:

- *Path expressions*. So far we used `structs` without knowing it, with manually defined constants such as `sym_length`, used in expressions such as `*(symbol+sym_length)`. We can now replace this with a new "`symbol.length`" syntax and require the compiler to compile it into the code corresponding to `*(symbol+4)`. Likewise, we can replace `*(symbol+sym_next)+sym_length` with the new syntax "`&symbol.next.length`" (a pointer to the `length` field of the `symbol`'s `next` symbol). Indeed, it is easy to deduce, from the definition of `Symbol`, that `name`, `length`, and `next` are *offset* by 0, 4, and 8 bytes, respectively, from the start of a symbol.
- *Implicit return*. Return type declarations, with the above "`->`" notation, enable the compiler to check that `return` statements return values of the correct type. They also enable it to check that functions without return type declaration, called *void* functions, do not return any value. But we can go further by combining this with the reachability analysis of the previous chapter. Indeed, we can require the compiler to automatically generate a `return;` at the end of void functions, if it is missing.

19.1.2 Type checking

We require our compiler to compute the type of an expression e, and whether it is correct or not, as follows:

- if e is an INTEGER it is correct and has type `u32`.
- if e is an IDENTIFIER:
  - referring to a `const` name: e is correct and has the constant's declared type (we require constants to declare their type with the "`const` X`:` *type* `=` x`;`" syntax).
  - referring to a `static` name: e is correct and has type `&u32`.
  - referring to a function parameter or a local variable: e is correct and has the declared type of this parameter or variable.
- if $e = e_1$`.field`: e_1 must have a `&`S type, where S is a `struct` name, and `field` must be a field of this struct. Then e's type is the declared type of `field`.
- if $e =$ `*`e_1: e_1 must have a pointer type `&`t, where t is a non-void type (see below). Then e's type is t, the type of the value at address e_1.
- if $e =$ `&`e_1: e_1 must have a non-void type t. Then e's type is the pointer type `&`t.
- if $e = e_1$`*`e_2, e_1`/`e_2, e_1`<<`e_2, e_1`>>`e_2, e_1`&`e_2, or e_1`|`e_2: e_1 and e_2 must have type `u32`, and e gets the same type.
- if $e = f$`(`$e_1, \ldots, e_n$`)`: f must have n parameters, and $e_1, \ldots, e_n$ must have the declared type of these parameters. Then e's type is the declared return type of f. If f does not return any value, this type is the *void* type (for which there is no syntax).

Additions e_1`+`e_2 and subtractions e_1`-`e_2 are a special case. If e_1 and e_2 have type `u32` then e is correct and has the same type. But this is not the only valid case. If e_1 has a pointer type `&`t and e_2 has type `u32` then e is correct too, and has type `&`t. This allows expressions computing a pointer by adding a byte offset to another pointer, such as `TC_KEYWORDS+i` (the address of the i^{th} byte of `TC_KEYWORDS`). Finally, the case where e_1 and e_2 have the same pointer type `&`t is also valid, *for the subtraction only*. Then e has type `u32`. This allows expressions computing the offset in bytes between two pointers.

The two sides of a comparison e_1`<`e_2, e_1`==`e_2, e_1`>`e_2, e_1`<=`e_2, e_1`!=`e_2, or e_1`>=`e_2 must have the same non-void type. This allows comparing integers but also addresses. This rule also applies to the two sides of an assignment. Finally, as described above, `return` statements must be consistent with the function's return type.

With these rules the compiler can compute the type of any valid expression, including the right hand side e of a "`let` x`:` t `=` e`;`" statement. It is therefore not necessary to declare the type of a local variable, and it should be possible to omit this declaration, as in "`let` x `=` e`;`". In this case x's "declared type" is e's type. However, if t is declared, then e's type should be equal to it.

In some cases a value represents different things depending on the context. For instance, in our compiler, the value of a `SYM_CONST` symbol is an integer, but the value of a `SYM_STATIC` symbol is a pointer. To handle these cases we add a new "e `as` t" syntax requirement. Such a *cast* expression has type t, whatever the type of e (provided

it is not void). In the previous example, we can then declare the symbol's `value` field with type `u32`, and cast it to `&u32` when dealing with a `SYM_STATIC` symbol.

In some other cases 0 is used as a special pointer value, for instance to represent the end of a linked list or a "not found" symbol. The above rules no longer allow this. For instance, passing 0 to a function expecting a `Symbol` pointer is incorrect. A workaround is to use "`0 as &Symbol`" instead, but this is not really satisfactory. Instead, we extend the above rules so that, anywhere an expression with a well-defined pointer type is required, a new "`null`" keyword can be used instead. This expression, compiled to `cst_0`, can thus be used in "`f(null)`" if `f` has a pointer parameter, "`x != null`" if `x` has a pointer type, "`return null;`" in a function returning a pointer, etc. However, we do not allow "`let x = null;`" since this does not provide a well-defined type for `x` (something like "`let x: &u32 = null;`" must be used instead).

Finally, in order to simplify the compiler, we add two important restrictions. The first is that `struct` types must be defined before they can be used. The second is that no function parameter, local variable or expression may have a `struct` type (only `struct` *pointer* types are allowed). This ensures that any value still fits in a word, as in the statements compiler.

19.1.3 Grammar

In order to support the above requirements we extend the grammar of our programming language as follows (unchanged parts are in gray or not shown):

```
program: (fn | struct | static | const)* END
...
fn_parameters: "(" (IDENTIFIER ":" type ("," IDENTIFIER ":" type)*)? ")"
               ("->" type)?
fn_body: "{" (const | let_stmt | stmt)* "}" | "=" INTEGER ";" | ";"
let_stmt: "let" IDENTIFIER (":" type)? "=" expr ";"
...
mult_expr: cast_expr (("*" | "/") cast_expr)*
cast_expr: pointer_expr ("as" type)?
pointer_expr: "*" pointer_expr | "&" path_expr | path_expr
path_expr: primitive_expr "." IDENTIFIER)*
primitive_expr: INTEGER | IDENTIFIER fn_arguments? | sizeof_expr | "null" |
                "(" expr ")"
sizeof_expr: "sizeof" "(" IDENTIFIER ")"
fn_arguments: "(" (expr ("," expr)*)? ")"
struct: "struct" IDENTIFIER
        "{" (IDENTIFIER ":" type ("," IDENTIFIER ":" type)*)? "}"
static: "static" IDENTIFIER "=" "[" INTEGER ("," INTEGER)* "]" ";"
const: "const" IDENTIFIER ":" type "=" INTEGER ";"
type: "&"* ("u32" | IDENTIFIER) ";"
```

The new "struct" and "type" rules correspond to type definitions. Expressions rules are updated to allow cast expressions "e `as` t", path expressions "$e.f$", and `null` expressions, but also a new "`sizeof(`S`)`" syntax. This expression evaluates to the size in bytes of S, which must be a `struct` type name (the size of S is 4 times its number of fields). The other rules are updated to take type declarations into account.

Note that the address-of operator "`&`" can now be followed by an arbitrary path expression (previously only identifiers could be used). This allows valid expressions such as "`&symbol.next.length`". But this rule now also allows invalid expressions such as "`&1`" or "`&(1+2)`" (numbers don't have an address). As for assignments, the compiler must then use another method than grammar rules to detect these errors.

Note also that two small changes unrelated to types are made in the "fn_body" rule. The first one allows the definition of constants inside a function. We call them local constants, because a constant defined in a function can only be used inside this function. The second change allows an external function to be *imported* in a program. For instance, "`fn delay(millis: u32) = 651;`" allows a program to call the `delay` function at address $\text{C0000}_{16}+651$ (see Table 9.3).

Finally, we also require the possibility to add comments inside programs, anywhere a space character is allowed. Comments must start with "`/*`" and end with "`*/`", as in "`/* This is a comment. */`". The compiler must simply skip them, like spaces.

19.2 Algorithms

19.2.1 Scanner

The scanner must be updated to support the new keywords ("`u32`", "`struct`", etc) and the new "`.`" and "`->`" tokens. It must also recognize comments and skip them, just like spaces, tabs and "new line" characters are skipped.

Adding the new keywords is trivial. However, to introduce a very useful algorithm, we define here a new method to find if an identifier x is equal to a keyword. The current algorithm compares x with each keyword, one by one, until it finds a match or all keywords have been tested. It thus takes more and more time, as new keywords are added. In fact, if x has n characters we only need to compare it with the keywords having n characters. Said otherwise, if $length(x) = n$ we only need to compare x with the keywords k for which $length(k) = n$. And this applies to any function h computing a number from an identifier. With a well chosen function, the keywords verifying $h(k) = h(x)$ can be reduced to at most one. Then we only need to compare x with a single keyword at most, which is much faster than the current algorithm. In practice h is called a *hashing* function, $h(x)$ is called the *hashcode* of x, and the map from keyword hashcodes $h(k)$ to the list of keywords having this hashcode is called a *hash map* or *hash table*.

In our compiler we compute $h(x)$ with a similar method as the one used to compute token values v and $lsb(v)$ in Chapter 15. More precisely, for x="$c_{n-1} \ldots c_1 c_0$", we initialize v to 0 and update v to $31v + c_i$ for each character c_i from left to right. We

then define $h(x)$ as v modulo 64, also equal to $v \wedge 63$ or "`v & 63`". With this choice all our keywords have a unique hashcode, and we can store them in a table with 64 rows. The i^{th} row is either empty, or contains the keyword k such that $h(k) = i$, and its associated $token$ value.

In order to support the new "`->`" token, that we associate with new $token$ value 20, we extend the `OPERATORS` table with a new row and a new column, as follows:

1^{st} character	!	&	<	=	>	|	-
`CHAR_TYPES`	10	11	12	13	14	15	16
any other case	1	and	iflt	'='	ifgt	or	sub
2^{nd} character = 1^{st}	1	18	lsl	ifeq	lsr	19	1
2^{nd} character = =	ifne	1	ifle	ifeq	ifge	1	1
2^{nd} character = >	1	1	1	1	lsr	1	20

Finally, in order to read comments and skip them, we extend the loop reading and skipping spacing characters in `tc_read_token`. If the next character is a "`/`", this loop now calls a new function which "looks ahead" at the second next character. If it is a "`*`" this function reads and skips a comment. Otherwise, it returns without reading any of these two characters, so that "`/`" is read as a normal token.

19.2.2 Backend

The only new grammar rules for which bytecode must be generated are the "`null`", "`sizeof`", and path expression rules (including in address-of expressions). The first two are trivial and can be compiled with the already existing `tc_write_cst_insn` backend function. The third case is less simple. To simplify the compiler, we want to compile all path expressions in the same way. As a consequence, when compiling "`&x`", for instance, compiling the path expression "`x`" produces a `get` instruction. To get a `ptr` instruction instead, we use the same solution as for assignments: we erase the last written instruction and write another instead (see Section 18.2.2). As for assignments, this last instruction must either be a `get` (rewritten to `ptr`) or a `load` (simply erased).

So far we reserved 12 KB in RAM for the code generated by the backend. Any larger code would override the compiler's variables and then the heap (see Figure 19.4), which would most probably lead to a crash. To avoid this we add a new dst_limit compiler variable, and we raise an error if any attempt is made to increase dst beyond that. We also add a similar $heap\_limit$ variable for the heap.

19.2.3 Parser

Type declarations

In order to compute the type of each expression, and to check that all expressions have a correct type, we first need a way to represent a type in memory. From the "type"

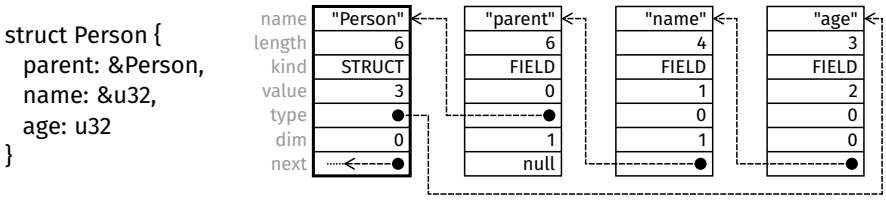


FIGURE 19.1 A `struct` with 3 fields, represented with 4 symbols. Symbols are represented more abstractly than in Figure 16.1 for clarity, but are actually stored one after the other as in Figure 16.1, with name values pointing to the source code (left).

grammar rule we see that a type is made of some number n of "`&`", followed either by "`u32`" or by the name of a `struct`. Assuming for now that each struct name has an associated symbol in the list of symbols (we define it below), this suggests a simple way to represent a type t. Namely with two words, one storing n and the other being either `null` (for `u32`) or a pointer to a symbol representing a struct. We call them the *dimension* and the *base type* of t, respectively.

We can then store the declared type of a function parameter or of a local variable by adding two new words in each symbol. Indeed, we already use one symbol per parameter and local variable, storing their name and stack frame slot index. Hence, storing their base type and dimension only requires two additional words. For brevity, we note them `type` and `dim`, respectively.

The above discussion assumes that each struct name has an associated symbol. This symbol should somehow contain an in-memory representation of each field of the struct, including their name, type, and index. Indeed, this is needed to check and compile a path expression such as "e`.field`". First to check that `field` is indeed a field of e, then to compute the type of this expression (equal to the `field`'s type), and finally to produce the bytecode equivalent to `*(`e`+`$4i$`)` (where i is the field index). In order to do this, we define the symbol of a struct as follows (see Figure 19.1):

- the symbol's name is the struct name.
- the symbol's kind is a new `STRUCT` kind value.
- the symbol's value is the number of fields of the struct (this is used for `sizeof`).
- the symbol's `type` is a list of symbols, one for each field, in reverse order[1]. Each symbol contains a field name and has a new `FIELD` kind. Its value is the field index, and its `type` and `dim` values are the base type and the dimension of the field's declared type.
- the symbol's `dim` is unused and set to 0.

Currently our compiler uses one symbol per function, containing the function name and the argument to use in `call` instructions to call it (or a list of placeholders). This is not sufficient to check that a function is called with the correct number of

[1]This list should in theory be stored in the symbol's value – a struct definition *is* a type, the type field is thus here a "metatype". In practice using the type field is easier because it avoids some casts.

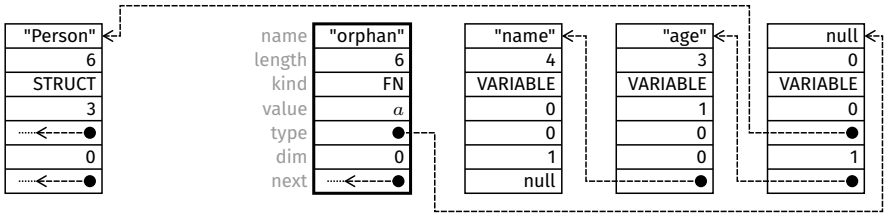


FIGURE 19.2 A "`fn orphan(name: &u32, age: u32) -> &Person`" function at address $\texttt{C0000}_{16}+a$ is represented with 4 symbols, for the function, its two parameters and its return type (from left to right), linked in reverse order.

arguments, each with a correct type. Nor to compute the type of a function call expression. In order to do this the symbol representing a function must contain an in-memory representation of the parameter and return types of this function. To this end, we store these types in the function's symbol `type`, as follows (see Figure 19.2):

- the `type` of a function's symbol is a symbol representing the function's return type. For functions returning a value it has a `null` name, a `VARIABLE` kind, a 0 value, and its `type` and `dim` values are the base type and the dimension of the function's return type. For functions returning no value a new `VOID` kind is used instead; `type` and `dim` are unused and set to 0.
- the `next` symbol of the above symbol is the start of a list of symbols, one per function parameter, in reverse order. Each symbol contains a parameter name and has the `VARIABLE` kind. Its value is the parameter index, and its `type` and `dim` values are the base type and the dimension of the parameters's declared type.

Type checking

We now have everything we need to compute the type of each expression, and to check if it is correct or not. Each type checking rule in Section 19.1.2 can be implemented in the corresponding expression parsing function. For instance, the rule stating that a shift expression e=e_1<<e_2 is correct and has type `u32` if and only if e_1 and e_2 have type `u32` can be implemented in the `tc_parse_shift_expr` function. For this we could make each expression parsing function return the type of the parsed expression. Then, for instance, `tc_parse_shift_expr` would get the types of e_1 and e_2 as a result of parsing them with `tc_parse_add_expr`. Unfortunately, each expression parsing function already returns a value, the origin of the expression's value. And a function cannot return several values. Our solution is to define a struct to store both the origin and the type of a parsed expression, and to store these structs in a stack:

- we define a `Value` struct with an `origin`, a `type` and a `dim` field (storing the origin of the expression value, and the base type and the dimension of its type, respectively). We also add a `slot` field, used for expressions whose value comes from a function parameter, a local variable, or a field. By definition, it contains the

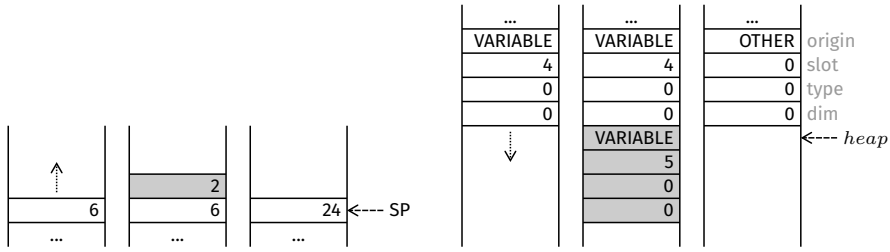


FIGURE 19.3 Evaluating `x<<y` with `x` and `y` two local variables equal to 6 and 2 pushes 6, then 2, then pops 6 and 2 and pushes the result $6 \ll 2 = 24$ (left). Compiling it pushes `x`'s `Value` on the *heap*, then `y`'s `Value`, pops these two values, checks them, and pushes a result `Value` (right). Addresses increase towards the bottom.

FIGURE 19.4 The compiler variables (in red) are between the compiled code (white) and the *heap* in RAM. The *heap* contains global `const`, `static`, `struct` and `fn` symbols (dark blue), local symbols (blue), and a stack of `Value`s (light blue). Global symbols are only added when there are no local symbols nor `Value`s. Local symbols are only added when the `Value` stack is empty. Unused memory is in gray. The native stack and its Stack Pointer SP are shown on the right.

stack frame slot index of the function parameter or local variable, or the index of the field in its struct.

- we store the `Value` struct data in the *heap*, after the symbols (see Figure 19.4), and organize them in a stack. When the compiled code for an expression runs it uses the "real" stack, managed with the Stack Pointer SP (see Section 7.2.3). For instance, e_1<<e_2 pushes the value of e_1, then the value of e_2, pops these two values, and pushes the result e_1<<e_2. We use a similar method for computing and checking the type of expressions. For instance, we push the `Value` of e_1 while compiling it with `tc_parse_add_expr`, and do the same for e_2. Then, in `tc_parse_shift_expr`, we pop these two `Value`s, check that their type is `u32`, and finally push a new `Value` with the `u32` type (see Figure 19.3).

null and void The `null` keyword is a special case in the type checking rules. To handle it we define a new `NULL` origin value (this is easier than using a special type, with some specific `type` and `dim` values). Similarly, we define a new `VOID` origin value for expressions calling functions without return type. Compiling such an expression pushes a `Value` with a `VOID` origin, unlike running the corresponding code (which leaves the stack unchanged). This simplifies all the expression parsing functions, which would otherwise have to check that the compilation of their subexpressions produces as many values.

19.3 Implementation

We can now extend the statements compiler in order to support types. As before, we need to write it in two steps, first without using types, then with them. To save space, we give the two compiler versions at the same time (without types in red, with in green). The start of the compiler does not change in the 1^{st} version, but can be rewritten with types and implicit `returns` in the 2^{nd}:

```
fn tc_main(src_buffer: &u32, dst_buffer: &u32, flash_buffer: &u32) -> u32;
fn main(src_buffer: &u32, dst_buffer: &u32, flash_buffer: &u32) -> u32 {
  return tc_main(src_buffer, dst_buffer, flash_buffer);
}

fn load8(ptr: &u32) -> u32 { return (*ptr) & 255; }
fn load16(ptr: &u32) -> u32 { return (*ptr) & 65535; }
fn store8(ptr: &u32, value: u32) { *ptr = (*ptr) & 4294967040 | value; ret
                                                                    urn; }
fn store16(ptr: &u32, value: u32) { *ptr = (*ptr) & 4294901760 | value; re
                                                                   turn; }

const PANIC_BUFFER: &&u32 = 1074666152;
fn panic_copy(src: &u32, dst: &u32) {
  *dst = *src;
  *(dst + 4) = *(src + 4);
  *(dst + 8) = *(src + 8);
  *(dst + 12) = *(src + 12);
  return;
}
fn panic_result(ptr: &u32) -> u32 {
  panic_copy(&ptr as &u32 - 16, ptr);
  *PANIC_BUFFER = ptr;
  return 0;
}
fn panic(error: u32) -> u32 {
  panic_copy(*PANIC_BUFFER, &error - 16);
  return error;
}
```

19.3.1 Shared constants and data structures

Here we add new constants for the new token values and the new symbol kinds. We also add the new symbol fields and compiler variables discussed above, plus a new `fn_return_type` variable, storing the symbol corresponding to the return type of the currently compiled function.

```
const TC_INTEGER: u32 = 2;
const TC_IDENTIFIER: u32 = 3;
```

```
const TC_ADD: u32 = 4;
const TC_SUB: u32 = 5;
const TC_MUL: u32 = 6;
const TC_DIV: u32 = 7;
const TC_BIT_AND: u32 = 8;
const TC_BIT_OR: u32 = 9;
const TC_SHIFT_LEFT: u32 = 10;
const TC_SHIFT_RIGHT: u32 = 11;
const TC_LT: u32 = 12;
const TC_GE: u32 = 17;
const TC_AND: u32 = 18;
const TC_OR: u32 = 19;
const TC_ARROW: u32 = 20;
const TC_AS: u32 = 138;
const TC_BREAK: u32 = 128;
const TC_CONST: u32 = 129;
const TC_ELSE: u32 = 130;
const TC_FN: u32 = 131;
const TC_IF: u32 = 132;
const TC_LET: u32 = 133;
const TC_LOOP: u32 = 134;
const TC_NULL: u32 = 139;
const TC_RETURN: u32 = 135;
const TC_SIZEOF: u32 = 140;
const TC_STATIC: u32 = 136;
const TC_STRUCT: u32 = 141;
const TC_U32: u32 = 142;
const TC_WHILE: u32 = 137;

const SYM_FN: u32 = 0;
const SYM_FORWARD_FN: u32 = 1;
const SYM_VARIABLE: u32 = 2;
const SYM_CONST: u32 = 3;
const SYM_STATIC: u32 = 4;
const SYM_STRUCT: u32 = 5;
const SYM_FIELD: u32 = 6;
const SYM_VOID: u32 = 7;

struct Symbol {
  const sym_name = 0;: &u32,
  const sym_length = 4;: u32,
  const sym_kind = 8;: u32,
  const sym_value = 12;: u32,
  const sym_type = 16;: &Symbol,
  const sym_dim = 20;: u32,
  const sym_next = 24;: &Symbol
}const sizeof_symbol = 28;
```

```
struct Compiler {
  const tc_src = 0;: &u32,
  const tc_src_end = 4;: &u32,
  const tc_next_char = 8;: u32,
  const tc_next_char_type = 12;: u32,
  const tc_next_token = 16;: u32,
  const tc_next_token_data = 20;: u32,
  const tc_next_token_length = 24;: u32,
  const tc_dst = 28;: &u32,
  const tc_dst_limit = 32;: &u32,
  const tc_heap = 36;: &u32,
  const tc_heap_limit = 40;: &u32,
  const tc_symbols = 44;: &Symbol,
  const tc_fn_dst = 48;: &u32,
  const tc_fn_return_type = 52;: &Symbol,
  const tc_flash_offset = 56;: u32
}const sizeof_compiler = 60;
```

19.3.2 Scanner

We start the scanner by setting the CHAR_TYPES of "." to its ASCII code 46, by changing the one of "-" to 16, and by updating the OPERATORS table (see Section 19.2.1):

```
static TC_CHAR_TYPES = [
  1,1,1,1,1,1,1,1,1,32,32,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  32,10,1,1,1,1,11,39,40,41,6,4,44,16,46,7,2,2,2,2,2,2,2,2,2,58,59,12,13,
  14,1,1,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,91,1,93,1,3,
  1,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,123,15,125,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1];

static TC_OPERATORS = [
  1,1,16,1,8,18,1,1,12,10,15,1,61,13,13,1,14,11,17,11,9,19,1,1,5,1,1,20];
```

We then add the new keywords, but we insert before them 64 bytes. If there is a keyword whose hashcode is i, then the i^{th} byte is the offset of this keyword in TC_KEYWORD. Otherwise it is 0. For instance, the 0^{th} byte is 0 because no keyword has hashcode 0. And the 8^{th} byte is 88 because the "fn" keyword has hashcode 8 ('f' $= 102$, 'n' $= 110$ and $102 * 31 + 110 \mod 64 = 8$) and starts at the 88^{th} byte (with its length, followed by its characters and its token value).

```
static TC_KEYWORDS = [
  0,0,0,0,0,0,0,107,88,0,0,0,0,0,0,0,0,0,0,145,137,
  0,0,121,0,0,96,0,92,0,0,0,0,0,75,101,0,0,0,0,0,0,0,
  0,0,129,0,113,150,64,0,0,0,0,0,0,82,0,0,0,0,0,68,
  2,'a','s',138,
```

```
  5,'b','r','e','a','k',128,
  5,'c','o','n','s','t',129,
  4,'e','l','s','e',130,
  2,'f','n',131,
  2,'i','f',132,
  3,'l','e','t',133,
  4,'l','o','o','p',134,
  4,'n','u','l','l',139,
  6,'r','e','t','u','r','n',135,
  6,'s','i','z','e','o','f',140,
  6,'s','t','a','t','i','c',136,
  6,'s','t','r','u','c','t',141,
  3,'u','3','2',142,
  5,'w','h','i','l','e',137];

fn mem_compare(ptr1: &u32, ptr2: &u32, size: u32) -> u32 {
  let i = 0;
  while i < size && load8(ptr1 + i) == load8(ptr2 + i) {
    i = i + 1;
  }
  return size - i;
}
```

Thanks to this hash table `tc_get_keyword` now only needs to compare the given identifier with at most one keyword, the one with the identifier's hashcode (which must be computed by the caller):

```
fn tc_get_keyword(start: &u32, length: u32, hashcode: u32) -> u32 {
  let keyword = load8(TC_KEYWORDS + hashcode);
  if keyword != 0 &&
    length == load8(TC_KEYWORDS + keyword) &&
    mem_compare(start, TC_KEYWORDS + keyword + 1, length) == 0 {
      return load8(TC_KEYWORDS + keyword + length + 1);
  }
  return TC_IDENTIFIER;
}

fn tc_read_char(self: &Compiler) -> u32 {
  let src = *(self+.tc_src);
  let src_end = *(self+.tc_src_end);
  if src >= src_end { panic(10); }
  src = src + 1;
  let c = 0;
  let type = 0;
  if src < src_end {
    c = load8(src);
    type = load8(TC_CHAR_TYPES + c);
  }
  *(self+.tc_src) = src;
```

```
  *(self+.tc_next_char) = c;
  *(self+.tc_next_char_type) = type;
  return type;
}
fn tc_read_integer(self: &Compiler) -> u32 {
  let type = *(self+.tc_next_char_type);
  let v = 0;
  while type == TC_INTEGER {
    v = v * 10 + (*(self+.tc_next_char) - '0');
    type = tc_read_char(self);
  }
  *(self+.tc_next_token_data) = v;
  return TC_INTEGER;
}
fn tc_read_quoted_char(self: &Compiler) -> u32 {
  tc_read_char(self);
  let value = *(self+.tc_next_char);
  if value < 32 || value >= 127 { panic(11); }
  if tc_read_char(self) != ''' { panic(12); }
  tc_read_char(self);
  *(self+.tc_next_token_data) = value;
  return TC_INTEGER;
}
```

We compute this hashcode with a few changes in the following function:

```
fn tc_read_identifier(self: &Compiler) -> u32 {
  let hashcode = 0;
  let start = *(self+.tc_src);
  let type = *(self+.tc_next_char_type);
  while type == TC_IDENTIFIER || type == TC_INTEGER {
    hashcode = 31 * hashcode + *(self+.tc_next_char);
    type = tc_read_char(self);
  }
  let length = *(self+.tc_src) - start;
  *(self+.tc_next_token_data) = start as u32;
  *(self+.tc_next_token_length) = length;
  return tc_get_keyword(start, length, hashcode & 63);
}
```

The next function is updated to use the new `OPERATORS` table:

```
fn tc_read_operator(self: &Compiler, first_char_type: u32) -> u32 {
  let second_char_type = tc_read_char(self);
  let index = 4 * (first_char_type - 10);
  if second_char_type == first_char_type {
    tc_read_char(self);
    index = index + 1;
  } else if *(self+.tc_next_char) == '=' {
    tc_read_char(self);
```

```
    index = index + 2;
  } else if *(self+.tc_next_char) == '>' {
    tc_read_char(self);
    index = index + 3;
  }
  return load8(TC_OPERATORS + index);
}
```

We finish the scanner with a new function to read comments, used in the loop skipping spacing characters in `tc_read_token`. This new function assumes that the next character is a "/". It returns 0 if it not followed by a "*". Otherwise it "reads" the comment and returns 1. In fact this function only increments *src*, without updating the other scanner variables. This is why it does not "read" the last "/", leaving this to the caller. Note also how the end of the comment is found: by comparing two characters at once with "*/" ('*' = 42, '/' = 47 and $42 + 47 * 256 = 12074$).

```
fn tc_read_comment(self: &Compiler, src: &u32) -> u32 {
  if src + 1 >= *(self+.tc_src_end) || load8(src + 1) != '*' { return 0; }
  while src + 3 < *(self+.tc_src_end) && load16(src + 2) != 12074 {
    src = src + 1;
  }
  *(self+.tc_src) = src + 3; /* The last '/' is NOT read. */
  return 1;
}

fn tc_read_token(self: &Compiler) {
  let type = *(self+.tc_next_char_type);
  while type == ' ' || type == TC_DIV {
    if type == TC_DIV && tc_read_comment(self, *(self+.tc_src)) == 0 { bre
                                                                       ak; }
    type = tc_read_char(self);
  }
  let token = type;
  if type == TC_INTEGER {
    token = tc_read_integer(self);
  } else if type == ''' {
    token = tc_read_quoted_char(self);
  } else if type == TC_IDENTIFIER {
    token = tc_read_identifier(self);
  } else if type >= 10 && type < 20 {
    token = tc_read_operator(self, type);
  } else if type != 0 {
    tc_read_char(self);
  }
  *(self+.tc_next_token) = token;
  return;
}
```

19.3.3 Backend

In order to check that dst and $heap$ do not grow beyond dst_limit and $heap\_limit$, respectively, we add a new ptr_limit parameter in the following function. We then check that $ptr + size \leq ptr\_limit$ and panic otherwise. In fact $ptr + size$ could overflow the capacity of a word and thus incorrectly appear as less than ptr_limit. The code below uses a reformulation of this test which avoids any possible overflow.

```
fn mem_allocate(size: u32, ptr_p: &&u32, ptr_limit: &u32) -> &u32 {
  let ptr = *ptr_p;
  if size > ptr_limit as u32 || ptr > ptr_limit - size { panic(1); }
  *ptr_p = ptr + size;
  return ptr;
}
fn tc_write8(self: &Compiler, value: u32) {
  store8(mem_allocate(1, &self+.tc_dst, *(self+.tc_dst_limit)), value);
  return;
}
fn tc_write16(self: &Compiler, value: u32) {
  store16(mem_allocate(2, &self+.tc_dst, *(self+.tc_dst_limit)), value);
  return;
}
fn tc_write32(self: &Compiler, value: u32) {
  *mem_allocate(4, &self+.tc_dst, *(self+.tc_dst_limit)) = value;
  return;
}
fn tc_write_insn(self: &Compiler, opcode: u32, argument: u32) {
  tc_write8(self, opcode);
  tc_write8(self, argument);
  return;
}
fn tc_write_placeholder(self: &Compiler, placeholder_p: &&u32) {
  let new_placeholder = *(self+.tc_dst);
  let last_placeholder = *placeholder_p;
  *placeholder_p = new_placeholder;
  if last_placeholder == 0null { last_placeholder = new_placeholder; }
  tc_write16(self, new_placeholder - last_placeholder);
  return;
}
fn tc_fill_placeholders(placeholder: &u32, value: u32) {
  let offset = 0;
  while placeholder != 0null {
    offset = load16(placeholder);
    store16(placeholder, value);
    if offset == 0 { break; }
    placeholder = placeholder - offset;
  }
  return;
```

```
}
fn tc_fill_label_placeholders(self: &Compiler, placeholder: &u32) {
  tc_fill_placeholders(placeholder, *(self+.tc_dst) - *(self+.tc_fn_dst));
  return;
}
fn tc_write_cst_insn(self: &Compiler, value: u32) {
  if value <= 1 {
    tc_write8(self, value);
  } else if value < 256 {
    tc_write_insn(self, /*cst8*/2, value);
  } else {
    tc_write8(self, /*cst*/3);
    tc_write32(self, value);
  }
  return;
}
fn tc_write_static_insn(self: &Compiler, dst: u32) {
  tc_write_cst_insn(self, dst - *(self+.tc_flash_offset));
  return;
}
fn tc_write_binary_insn(self: &Compiler, token: u32) {
  tc_write8(self, token);
  return;
}
fn tc_write_jump_insn(self: &Compiler, token: u32, placeholder_p: &&u32) {
  tc_write8(self, token);
  tc_write_placeholder(self, placeholder_p);
  return;
}
fn tc_write_goto_insn(self: &Compiler, placeholder_p: &&u32) {
  tc_write_jump_insn(self, /*goto*/18, placeholder_p);
  return;
}
fn tc_write_loop_insn(self: &Compiler, loop_dst: &u32) {
  tc_write8(self, /*goto*/18);
  tc_write16(self, loop_dst - *(self+.tc_fn_dst));
  return;
}
```

The next function now writes the code to load the field of a struct, given by its index. It loads the value starting $4 * \mathit{field}$ bytes after the struct's address.

```
fn tc_write_load_insn(self: &Compiler, field: u32) {
  if field > 0 {
    tc_write_cst_insn(self, field << 2);
    tc_write8(self, /*add*/4);
  }
  tc_write8(self, /*load*/19);
  return;
```

```
}
fn tc_erase_load_insn(self: &Compiler) {
  *(self+.tc_dst) = *(self+.tc_dst) - 1;
  return;
}
```

A new *field* parameter is also added to the next function. It is introduced to simplify the next chapter, although it is unused for now. For the same reason, a new function, empty for now, is added to write the code computing the address of a field.

```
fn tc_write_store_insn(self: &Compiler, field: u32) {
  tc_write8(self, /*store*/20);
  return;
}
fn tc_write_address_of_insn(self: &Compiler, field: u32) {
  return;
}
fn tc_write_ptr_insn(self: &Compiler, variable: u32) {
  tc_write_insn(self, /*ptr*/21, variable);
  return;
}
fn tc_write_get_insn(self: &Compiler, variable: u32) {
  tc_write_insn(self, /*get*/22, variable);
  return;
}
```

The following function no longer returns the argument of the erased instruction. This is no longer necessary since we now keep track of the stack frame slot indices in the `Value`'s `slot` field. In fact this field was added to simplify this function.

```
fn tc_erase_get_insn(self: &Compiler) {
  *(self+.tc_dst) = *(self+.tc_dst) - 2;
  return;
}
fn tc_write_set_insn(self: &Compiler, variable: u32) {
  tc_write_insn(self, /*set*/23, variable);
  return;
}
fn tc_write_pop_insn(self: &Compiler) {
  tc_write8(self, /*pop*/24);
  return;
}
fn tc_write_fn_insn(self: &Compiler, arity: u32) {
  tc_write_insn(self, /*fn*/25, arity);
  return;
}
fn tc_get_fn_value(self: &Compiler, fn_dst: u32) -> u32 {
  return fn_dst - *(self+.tc_flash_offset) - 786432;
}
```

```
fn tc_write_call_insn(self: &Compiler, function: &Symbol) {
  tc_write8(self, /*call*/26);
  if *(function+.sym_kind) == SYM_FORWARD_FN {
    tc_write_placeholder(self, &function+.sym_value as &&u32);
  } else {
    tc_write16(self, *(function+.sym_value));
  }
  return;
}
```

Finally, the last backend function is updated to generate either a `ret` or a `retv` instruction, depending on the return type of the currently compiled function.

```
fn tc_write_return_insn(self: &Compiler) {
  if *(*(self+.tc_fn_return_type)+.sym_kind) == SYM_VOID {
    tc_write8(self, /*ret*/29);
  } else {
    tc_write8(self, /*retv*/30);
  }
  return;
}
```

19.3.4 Parser

```
fn sym_lookup(symbol: &Symbol, name: &u32, length: u32) -> &Symbol {
  while symbol != 0null {
    if *(symbol+.sym_length) == length && mem_compare(*(symbol+.sym_name),↩
                                                      ↩ name, length) == 0 {
      return symbol;
    }
    symbol = *(symbol+.sym_next);
  }
  return 0null;
}
```

The symbols represented in Figures 19.1 and 19.2 are not in the *symbols* list. To create them we split `tc_add_symbol` in two, with new `type` and `dim` parameters:

```
fn tc_new_symbol(self: &Compiler, name: &u32, length: u32, kind: u32,
    value: u32, type: &Symbol, dim: u32, next: &Symbol) -> &Symbol {
  let symbol = mem_allocate(sizeof_symbol(Symbol), &self+.tc_heap, *(self+↩
                                                  ↩.tc_heap_limit)) as &Symbol;
  if sym_lookup(next, name, length) != 0null { panic(30); }
  *(symbol+.sym_name) = name;
  *(symbol+.sym_length) = length;
  *(symbol+.sym_kind) = kind;
  *(symbol+.sym_value) = value;
  *(symbol+.sym_type) = type;
  *(symbol+.sym_dim) = dim;
```

```
  *(symbol+.sym_next) = next;
  return symbol;
}
fn tc_add_symbol(self: &Compiler, name: &u32, length: u32, kind: u32,
    value: u32, type: &Symbol, dim: u32) -> &Symbol {
  *(self+.tc_symbols) = tc_new_symbol(
      self, name, length, kind, value, type, dim, *(self+.tc_symbols));
  return *(self+.tc_symbols);
}
fn tc_add_or_resolve_fn_symbol(self: &Compiler, name: &u32, length: u32, v⸮
                                                        ⸮alue: u32) -> &Symbol {
  let symbol = sym_lookup(*(self+.tc_symbols), name, length);
  if symbol == 0null {
    return tc_add_symbol(self, name, length, SYM_FN, value, 0null, 0);
  }
  if *(symbol+.sym_kind) != SYM_FORWARD_FN { panic(31); }
  tc_fill_placeholders(*(symbol+.sym_value) as &u32, value);
  *(symbol+.sym_kind) = SYM_FN;
  *(symbol+.sym_value) = value;
  return symbol;
}
fn tc_parse_token(self: &Compiler, token: u32) {
  if *(self+.tc_next_token) != token { panic(20); }
  tc_read_token(self);
  return;
}
fn tc_parse_integer(self: &Compiler) -> u32 {
  if *(self+.tc_next_token) != TC_INTEGER { panic(21); }
  let value = *(self+.tc_next_token_data);
  tc_read_token(self);
  return value;
}
fn tc_parse_identifier(self: &Compiler, length_p: &u32) -> &u32 {
  if *(self+.tc_next_token) != TC_IDENTIFIER { panic(22); }
  let name = *(self+.tc_next_token_data) as &u32;
  *length_p = *(self+.tc_next_token_length);
  tc_read_token(self);
  return name;
}
fn tc_parse_symbol(self: &Compiler, symbol: &Symbol) -> &Symbol {
  let length = 0;
  let name = tc_parse_identifier(self, &length);
  symbol = sym_lookup(symbol, name, length);
  if symbol == 0null { panic(33); }
  return symbol;
}
```

The following function implements a corrected version of the “type” grammar

rule. Indeed, a type declaration such as "&&&u32" is not read by the scanner as 3 "&" tokens followed by "u32", but as the 3 tokens "&&", "&", and "u32". It is used in `tc_parse_const` to parse the constant's declared type.

```
fn tc_parse_type(self: &Compiler, dim: &u32) -> &Symbol {
  *dim = 0;
  while *(self+.tc_next_token) == TC_BIT_AND || *(self+.tc_next_token) == ↩
                                                                ↩TC_AND {
    *dim = *dim + 1;
    if *(self+.tc_next_token) == TC_AND { *dim = *dim + 1; }
    tc_read_token(self);
  }
  if *(self+.tc_next_token) == TC_U32 {
    tc_read_token(self);
    return 0null;
  }
  let symbol = tc_parse_symbol(self, *(self+.tc_symbols));
  if *dim == 0 || *(symbol+.sym_kind) != SYM_STRUCT { panic(42); }
  return symbol;
}

fn tc_parse_const(self: &Compiler) {
  tc_parse_token(self, TC_CONST);
  let length = 0;
  let name = tc_parse_identifier(self, &length);
  tc_parse_token(self, ':');
  let dim = 0;
  let type = tc_parse_type(self, &dim);
  tc_parse_token(self, '=');
  tc_add_symbol(self, name, length, SYM_CONST, tc_parse_integer(self), typ↩
                                                                ↩e, dim);
  tc_parse_token(self, ';');
  return;
}

fn tc_parse_static(self: &Compiler) {
  tc_parse_token(self, TC_STATIC);
  let length = 0;
  let name = tc_parse_identifier(self, &length);
  tc_add_symbol(self, name, length, SYM_STATIC, *(self+.tc_dst) as u32, 0n↩
                                                                ↩ull, 1);
  tc_parse_token(self, '=');
  tc_parse_token(self, '[');
  tc_write8(self, tc_parse_integer(self));
  while *(self+.tc_next_token) == ',' {
    tc_read_token(self);
    tc_write8(self, tc_parse_integer(self));
  }
```

```
  tc_parse_token(self, ']');
  tc_parse_token(self, ';');
  return;
}
```

The next function implements the new "struct" rule. It first adds to *symbols* a symbol for the struct itself (without fields), so that the symbols for its fields, collected in `fields`, can use it as their type.

```
fn tc_parse_struct(self: &Compiler) {
  tc_parse_token(self, TC_STRUCT);
  let length = 0;
  let name = tc_parse_identifier(self, &length);
  let symbol = tc_add_symbol(self, name, length, SYM_STRUCT, 0, 0null, 0);
  tc_parse_token(self, '{');
  let dim = 0;
  let type: &Symbol = 0null;
  let fields: &Symbol = 0null;
  let value = 0;
  while *(self+.tc_next_token) != '}' {
    if value > 0 { tc_parse_token(self, ','); }
    name = tc_parse_identifier(self, &length);
    tc_parse_token(self, ':');
    type = tc_parse_type(self, &dim);
    fields = tc_new_symbol(self, name, length, SYM_FIELD, value, type, dim
                                                                  , fields);
    value = value + 1;
  }
  tc_read_token(self);
  *(symbol+.sym_value) = value;
  *(symbol+.sym_type) = fields;
  return;
}
```

We then define the new `NULL` and `VOID` origins, and the new `Value` struct. The next two functions check if the type of a value is equal to a required type, either given directly or via a `Symbol`, and panic otherwise. They take the exception for `null` into account (`null` can be used anywhere a pointer type is required).

```
const FROM_ADDRESS: u32 = 0;
const FROM_VARIABLE: u32 = 1;
const FROM_NULL: u32 = 2;
const FROM_VOID: u32 = 3;
const FROM_OTHER: u32 = 255;
```

```
struct Value {
  const val_origin = 0;: u32,
  const val_slot = 4;: u32,
  const val_type = 8;: &Symbol,
```

```
  const val_dim = 12;: u32
}const sizeof_value = 16;
```

```
fn value_type_check(self: &Value, type: &Symbol, dim: u32) {
  if *(self+.val_origin) == FROM_NULL && dim != 0 { return; }
  if *(self+.val_type) != type || *(self+.val_dim) != dim { panic(43); }
  return;
}
fn value_check(self: &Value, symbol: &Symbol) {
  value_type_check(self, *(symbol+.sym_type), *(symbol+.sym_dim));
  return;
}
```

The following new functions are used to manage the `Value` stack. The first one pushes a new value given explicitly. It panics if its type is a struct type (recall that only struct pointer types are allowed). The second one uses it to push a value corresponding to a symbol. It computes the value's origin from the symbol's kind. The third returns a pointer to the last pushed value. The last ones do the same but also pop this value. Their result must be used before pushing a new value!

```
fn tc_push_value(self: &Compiler, origin: u32, slot: u32, type: &Symbol, d↵
                                                        ↵im: u32) -> &Value {
  let value = mem_allocate(sizeof_value(Value), &self+.tc_heap, *(self+.tc↵
                                                    ↵_heap_limit)) as &Value;
  if dim == 0 && type != 0null { panic(44); }
  *(value+.val_origin) = origin;
  *(value+.val_slot) = slot;
  *(value+.val_type) = type;
  *(value+.val_dim) = dim;
  return value;
}
fn tc_push_symbol_value(self: &Compiler, symbol: &Symbol) -> &Value {
  let origin = FROM_OTHER;
  if *(symbol+.sym_kind) == SYM_FIELD { origin = FROM_ADDRESS; }
  else if *(symbol+.sym_kind) == SYM_VARIABLE { origin = FROM_VARIABLE; }
  else if *(symbol+.sym_kind) == SYM_VOID { origin = FROM_VOID; }
  return tc_push_value(self, origin, *(symbol+.sym_value), *(symbol+.sym_t↵
                                                    ↵ype), *(symbol+.sym_dim));
}
fn tc_top_value(self: &Compiler) -> &Value {
  return (*(self+.tc_heap) - sizeof_value(Value)) as &Value;
}
fn tc_pop_value_or_void(self: &Compiler) -> &Value {
  *(self+.tc_heap) = *(self+.tc_heap) - sizeof_value(Value);
  return *(self+.tc_heap) as &Value;
}
fn tc_pop_value(self: &Compiler) -> &Value {
  let value = tc_pop_value_or_void(self);
```

```
  if *(value+.val_origin) == FROM_VOID { panic(45); }
  return value;
}

fn tc_parse_expr(self: &Compiler);
```

The next function implements the type checking rule for function call expressions. It checks that the number of arguments (`argument_count`) is equal to the number of parameters, pops their types and check them against the parameter types of the callee (`function`). It then pushes a value corresponding to the function's return type. It is called in the next function, refactored to compute `argument_count`.

```
fn tc_check_fn_arguments(self: &Compiler, function: &Symbol, argument_coun⸮
                                                              ⸮t: u32) {
  let parameter = *(*(function+.sym_type)+.sym_next);
  while parameter != 0null && argument_count != 0 {
    value_check(tc_pop_value(self), parameter);
    parameter = *(parameter+.sym_next);
    argument_count = argument_count - 1;
  }
  if parameter != 0null || argument_count != 0 { panic(46); }
  tc_push_symbol_value(self, *(function+.sym_type));
  return;
}
fn tc_parse_fn_arguments(self: &Compiler, function: &Symbol) {
  if *(function+.sym_kind) != SYM_FN && *(function+.sym_kind) != SYM_FORWA⸮
                                                                ⸮RD_FN {
    panic(34);
  }
  tc_parse_token(self, '(');
  let argument_count = 0;
  while *(self+.tc_next_token) != ')' {
    if argument_count > 0 { tc_parse_token(self, ','); }
    tc_parse_expr(self);
    argument_count = argument_count + 1;
  }
  tc_read_token(self);
  tc_check_fn_arguments(self, function, argument_count);
  tc_write_call_insn(self, function);
  return;
}
```

Parsing and compiling a `sizeof` expression is very easy. The size of a struct is 4 times its number of fields, itself stored in the `value` field of the struct's symbol.

```
fn tc_parse_sizeof_expr(self: &Compiler) {
  tc_parse_token(self, TC_SIZEOF);
  tc_parse_token(self, '(');
  let symbol = tc_parse_symbol(self, *(self+.tc_symbols));
```

```
  if *(symbol+.sym_kind) != SYM_STRUCT { panic(47); }
  tc_push_value(self, FROM_OTHER, 0, 0null, 0);
  tc_write_cst_insn(self, *(symbol+.sym_value) << 2);
  tc_parse_token(self, ')');
  return;
}
```

The next function is updated to parse the new `sizeof` and `null` expressions, and to push a value instead of returning the origin of the expression's value.

```
fn tc_parse_primitive_expr(self: &Compiler) {
  let symbol: &Symbol = 0null;
  if *(self+.tc_next_token) == TC_INTEGER {
    tc_push_value(self, FROM_OTHER, 0, 0null, 0);
    tc_write_cst_insn(self, tc_parse_integer(self));
  } else if *(self+.tc_next_token) == TC_IDENTIFIER {
    symbol = tc_parse_symbol(self, *(self+.tc_symbols));
    if *(self+.tc_next_token) == '(' {
      tc_parse_fn_arguments(self, symbol);
    } else {
      tc_push_symbol_value(self, symbol);
      if *(symbol+.sym_kind) == SYM_VARIABLE {
        tc_write_get_insn(self, *(symbol+.sym_value));
      } else if *(symbol+.sym_kind) == SYM_CONST {
        tc_write_cst_insn(self, *(symbol+.sym_value));
      } else if *(symbol+.sym_kind) == SYM_STATIC {
        tc_write_static_insn(self, *(symbol+.sym_value));
      } else {
        panic(35);
      }
    }
  } else if *(self+.tc_next_token) == TC_SIZEOF {
    tc_parse_sizeof_expr(self);
  } else if *(self+.tc_next_token) == TC_NULL {
    tc_read_token(self);
    tc_push_value(self, FROM_NULL, 0, 0null, 0);
    tc_write_cst_insn(self, 0);
  } else {
    tc_parse_token(self, '(');
    tc_parse_expr(self);
    tc_parse_token(self, ')');
  }
  return;
}
```

Compiling a path expression e`.f` is done by checking that e's `value` has a struct pointer type, searching for the `field` symbol corresponding to `f` in the struct's list of fields (`value.type.type`), pushing a value corresponding to the type of `f`, and finally writing the code to load its value.

```
fn tc_parse_path_expr(self: &Compiler) {
  let value: &Value = 0null;
  let field: &Symbol = 0null;
  tc_parse_primitive_expr(self);
  while *(self+.tc_next_token) == '.' {
    tc_read_token(self);
    value = tc_pop_value(self);
    if *(value+.val_type) == 0null || *(value+.val_dim) != 1 { panic(48); }
    field = tc_parse_symbol(self, *(*(value+.val_type)+.sym_type));
    tc_push_symbol_value(self, field);
    tc_write_load_insn(self, *(field+.sym_value));
  }
  return;
}
```

The next function is updated to implement the type checking rules of dereference and address-of expressions, as well as the new grammar rule for address-of expressions (compiled by rewriting the last written instruction – see Section 19.2.2).

```
fn tc_parse_pointer_expr(self: &Compiler) {
  let value: &Value = 0null;
  if *(self+.tc_next_token) == TC_MUL {
    tc_read_token(self);
    tc_parse_pointer_expr(self);
    value = tc_pop_value(self);
    if *(value+.val_dim) == 0 { panic(49); }
    tc_push_value(self, FROM_ADDRESS, 0, *(value+.val_type), *(value+.val_dim) - 1);
    tc_write_load_insn(self, 0);
  } else if *(self+.tc_next_token) == TC_BIT_AND {
    tc_read_token(self);
    tc_parse_path_expr(self);
    value = tc_pop_value(self);
    if *(value+.val_origin) == FROM_ADDRESS {
      tc_erase_load_insn(self);
      tc_write_address_of_insn(self, *(value+.val_slot));
    } else if *(value+.val_origin) == FROM_VARIABLE {
      tc_erase_get_insn(self);
      tc_write_ptr_insn(self, *(value+.val_slot));
    } else {
      panic(36);
    }
    tc_push_value(self, FROM_OTHER, 0, *(value+.val_type), *(value+.val_dim) + 1);
  } else {
    tc_parse_path_expr(self);
  }
```

```
  return;
}
```

Compiling a cast expression only requires to pop a value and push it again with the parsed type. The value's origin stays the same, unless it was NULL (after a cast a null expression can no longer be used anywhere a pointer types is expected). No code needs to be generated.

```
fn tc_parse_cast_expr(self: &Compiler) {
  tc_parse_pointer_expr(self);
  if *(self+.tc_next_token) != TC_AS { return; }
  tc_read_token(self);
  let value = tc_pop_value(self);
  if *(value+.val_origin) == FROM_NULL { *(value+.val_origin) = FROM_OTHER;
  let dim = 0;
  let type = tc_parse_type(self, &dim);
  tc_push_value(self, *(value+.val_origin), 0, type, dim);
  return;
}
```

The next two functions implement the type checking rules for arithmetic and logic expressions, including the special cases for additions and subtractions. They are used in the next parsing functions, which are all updated in the same straightforward way.

```
fn tc_check_integer_expr(self: &Compiler) {
  let right_value = tc_pop_value(self);
  let left_value = tc_pop_value(self);
  if *(left_value+.val_type) != 0null || *(left_value+.val_dim) != 0 { panic(50); }
  if *(right_value+.val_type) != 0null || *(right_value+.val_dim) != 0 { panic(51); }
  tc_push_value(self, FROM_OTHER, 0, 0null, 0);
  return;
}
```

```
fn tc_check_add_or_sub_expr(self: &Compiler, token: u32) {
  let right_value = tc_pop_value(self);
  let left_value = tc_pop_value(self);
  if *(right_value+.val_dim) == 0 {
    tc_push_value(self, FROM_OTHER, 0, *(left_value+.val_type), *(left_value+.val_dim));
  } else if token == TC_SUB && *(left_value+.val_dim) != 0 {
    if *(left_value+.val_type) != *(right_value+.val_type) { panic(52); }
    if *(left_value+.val_dim) != *(right_value+.val_dim) { panic(52); }
    tc_push_value(self, FROM_OTHER, 0, 0null, 0);
  } else {
    panic(53);
  }
```

```
  return;
}

fn tc_parse_mult_expr(self: &Compiler) {
  tc_parse_cast_expr(self);
  let next_token = *(self+.tc_next_token);
  while next_token == TC_MUL || next_token == TC_DIV {
    tc_read_token(self);
    tc_parse_cast_expr(self);
    tc_check_integer_expr(self);
    tc_write_binary_insn(self, next_token);
    next_token = *(self+.tc_next_token);
  }
  return;
}
fn tc_parse_add_expr(self: &Compiler) {
  tc_parse_mult_expr(self);
  let next_token = *(self+.tc_next_token);
  while next_token == TC_ADD || next_token == TC_SUB {
    tc_read_token(self);
    tc_parse_mult_expr(self);
    tc_check_add_or_sub_expr(self, next_token);
    tc_write_binary_insn(self, next_token);
    next_token = *(self+.tc_next_token);
  }
  return;
}
fn tc_parse_shift_expr(self: &Compiler) {
  tc_parse_add_expr(self);
  let next_token = *(self+.tc_next_token);
  if next_token == TC_SHIFT_LEFT || next_token == TC_SHIFT_RIGHT {
    tc_read_token(self);
    tc_parse_add_expr(self);
    tc_check_integer_expr(self);
    tc_write_binary_insn(self, next_token);
  }
  return;
}
fn tc_parse_bit_and_expr(self: &Compiler) {
  tc_parse_shift_expr(self);
  while *(self+.tc_next_token) == TC_BIT_AND {
    tc_read_token(self);
    tc_parse_shift_expr(self);
    tc_check_integer_expr(self);
    tc_write_binary_insn(self, TC_BIT_AND);
  }
  return;
```

```
}
fn tc_parse_expr(self: &Compiler) {
  tc_parse_bit_and_expr(self);
  while *(self+.tc_next_token) == TC_BIT_OR {
    tc_read_token(self);
    tc_parse_bit_and_expr(self);
    tc_check_integer_expr(self);
    tc_write_binary_insn(self, TC_BIT_OR);
  }
  return;
}
```

The following function checks that the two sides of a comparison expression have the same non-void type, unless the right hand side is `null` (in which case the left hand side must have a pointer type). It is used to check comparison expressions and assignment statements. Most of the following statement parsing functions don't use expressions directly and are thus unchanged.

```
fn tc_check_comparison_expr(self: &Compiler) {
  let right_value = tc_pop_value(self);
  let left_value = tc_pop_value(self);
  if *(left_value+.val_dim) != 0 && *(right_value+.val_origin) == FROM_NUL⸮
                                                              ⸮L { return; }
  if *(left_value+.val_type) != *(right_value+.val_type) ||
      *(left_value+.val_dim) != *(right_value+.val_dim) {
    panic(54);
  }
  return;
}

fn tc_parse_comparison_expr(self: &Compiler) -> u32 {
  tc_parse_expr(self);
  let token = *(self+.tc_next_token);
  if token < TC_LT || token > TC_GE { panic(25); }
  tc_read_token(self);
  tc_parse_expr(self);
  tc_check_comparison_expr(self);
  return token;
}
fn tc_parse_and_expr(self: &Compiler, else_refs_p: &&u32) -> u32 {
  let token = tc_parse_comparison_expr(self);
  while *(self+.tc_next_token) == TC_AND {
    tc_read_token(self);
    tc_write_jump_insn(self, TC_LT + TC_GE - token, else_refs_p);
    token = tc_parse_comparison_expr(self);
  }
  return token;
}
```

```
fn tc_parse_boolean_expr(self: &Compiler, then_refs_p: &&u32) -> &u32 {
  let else_refs: &u32 = 0null;
  let token = tc_parse_and_expr(self, &else_refs);
  while *(self+.tc_next_token) == TC_OR {
    tc_read_token(self);
    tc_write_jump_insn(self, token, then_refs_p);
    tc_fill_label_placeholders(self, else_refs);
    else_refs = 0null;
    token = tc_parse_and_expr(self, &else_refs);
  }
  tc_write_jump_insn(self, TC_LT + TC_GE - token, &else_refs);
  return else_refs;
}

const END_UNREACHABLE: u32 = 0;
const END_REACHABLE: u32 = 1;
fn tc_parse_stmt(self: &Compiler, break_refs_p: &&u32) -> u32;

fn tc_parse_block_stmt(self: &Compiler, break_refs_p: &&u32) -> u32 {
  let state = END_REACHABLE;
  tc_parse_token(self, '{');
  while *(self+.tc_next_token) != '}' {
    if state == END_UNREACHABLE { panic(37); }
    state = tc_parse_stmt(self, break_refs_p);
  }
  tc_read_token(self);
  return state;
}
fn tc_parse_assignment(self: &Compiler) -> u32 {
  let value = tc_top_value(self);
  let origin = *(value+.val_origin);
  let slot = *(value+.val_slot);
  if origin == FROM_ADDRESS {
    tc_erase_load_insn(self);
  } else if origin == FROM_VARIABLE {
    tc_erase_get_insn(self);
  } else {
    panic(38);
  }
  tc_parse_token(self, '=');
  tc_parse_expr(self);
  tc_check_comparison_expr(self);
  if origin == FROM_ADDRESS {
    tc_write_store_insn(self, slot);
  } else {
    tc_write_set_insn(self, slot);
  }
```

```
  return END_REACHABLE;
}
fn tc_parse_expr_or_assign_stmt(self: &Compiler) -> u32 {
  tc_parse_expr(self);
  if *(self+.tc_next_token) == '=' {
    tc_parse_assignment(self);
  } else if *(tc_pop_value_or_void(self)+.val_origin) != FROM_VOID {
    tc_write_pop_insn(self);
  }
  tc_parse_token(self, ';');
  return END_REACHABLE;
}
fn tc_parse_return_stmt(self: &Compiler) -> u32 {
  tc_parse_token(self, TC_RETURN);
  if *(self+.tc_next_token) == ';' {
    if *(*(self+.tc_fn_return_type)+.sym_kind) != SYM_VOID { panic(55); }
  } else {
    if *(*(self+.tc_fn_return_type)+.sym_kind) == SYM_VOID { panic(56); }
    tc_parse_expr(self);
    value_check(tc_pop_value(self), *(self+.tc_fn_return_type));
  }
  tc_parse_token(self, ';');
  tc_write_return_insn(self);
  return END_UNREACHABLE;
}
fn tc_parse_break_stmt(self: &Compiler, break_refs_p: &&u32) -> u32 {
  tc_parse_token(self, TC_BREAK);
  tc_parse_token(self, ';');
  tc_write_goto_insn(self, break_refs_p);
  return END_UNREACHABLE;
}
fn tc_parse_while_or_loop_stmt(self: &Compiler) -> u32 {
  let loop_dst = *(self+.tc_dst);
  let body_refs: &u32 = 0null;
  let end_refs: &u32 = 0null;
  let token = *(self+.tc_next_token);
  tc_read_token(self);
  if token == TC_WHILE {
    end_refs = tc_parse_boolean_expr(self, &body_refs);
  }
  tc_fill_label_placeholders(self, body_refs);
  tc_parse_block_stmt(self, &end_refs);
  tc_write_loop_insn(self, loop_dst);
  tc_fill_label_placeholders(self, end_refs);
  if token == TC_LOOP && end_refs == 0null { return END_UNREACHABLE; }
  return END_REACHABLE;
}
```

```
fn tc_parse_if_stmt(self: &Compiler, break_refs_p: &&u32) -> u32 {
  tc_parse_token(self, TC_IF);
  let then_refs: &u32 = 0null;
  let else_refs = tc_parse_boolean_expr(self, &then_refs);
  tc_fill_label_placeholders(self, then_refs);
  let state = tc_parse_block_stmt(self, break_refs_p);
  let end_if_refs: &u32 = 0null;
  if *(self+.tc_next_token) == TC_ELSE {
    tc_read_token(self);
    if state == END_REACHABLE {
      tc_write_goto_insn(self, &end_if_refs);
    }
    tc_fill_label_placeholders(self, else_refs);
    if *(self+.tc_next_token) == '{' {
      state = state | tc_parse_block_stmt(self, break_refs_p);
    } else {
      state = state | tc_parse_if_stmt(self, break_refs_p);
    }
    tc_fill_label_placeholders(self, end_if_refs);
  } else {
    tc_fill_label_placeholders(self, else_refs);
    state = END_REACHABLE;
  }
  return state;
}
fn tc_parse_stmt(self: &Compiler, break_refs_p: &&u32) -> u32 {
  if *(self+.tc_next_token) == TC_IF {
    return tc_parse_if_stmt(self, break_refs_p);
  } else if *(self+.tc_next_token) == TC_WHILE || *(self+.tc_next_token) =≀
                                                          ≀= TC_LOOP {
    return tc_parse_while_or_loop_stmt(self);
  } else if *(self+.tc_next_token) == TC_BREAK {
    if break_refs_p == 0null { panic(39); }
    return tc_parse_break_stmt(self, break_refs_p);
  } else if *(self+.tc_next_token) == TC_RETURN {
    return tc_parse_return_stmt(self);
  }
  return tc_parse_expr_or_assign_stmt(self);
}
```

A `let` statement can have an optional type declaration. If this is the case, the type of the right hand side is checked with `value_type_check` against this declared type (which becomes the type of the variable). Otherwise the variable gets the type of the right hand side, which must not be `null`.

```
fn tc_parse_let_stmt(self: &Compiler, variable: u32) -> u32 {
  tc_parse_token(self, TC_LET);
  let length = 0;
  let name = tc_parse_identifier(self, &length);
```

```
  let separator = *(self+.tc_next_token);
  let type: &Symbol = 0null;
  let dim = 0;
  if separator == ':' {
    tc_read_token(self);
    type = tc_parse_type(self, &dim);
  }
  tc_parse_token(self, '=');
  tc_parse_expr(self);
  tc_parse_token(self, ';');
  let value = tc_pop_value(self);
  if separator == ':' {
    value_type_check(value, type, dim);
  } else {
    if *(value+.val_origin) == FROM_NULL { panic(57); }
    type = *(value+.val_type);
    dim = *(value+.val_dim);
  }
  tc_add_symbol(self, name, length, SYM_VARIABLE, variable, type, dim);
  return variable + 1;
}

fn tc_parse_fn_name(self: &Compiler) -> &Symbol {
  let length = 0;
  let name = tc_parse_identifier(self, &length);
  let fn_dst = *(self+.tc_dst);
  *(self+.tc_fn_dst) = fn_dst;
  let value = tc_get_fn_value(self, fn_dst as u32);
  return tc_add_or_resolve_fn_symbol(self, name, length, value);
}
```

When a function is declared before it is implemented, such as the `tc_main` function, its parameter and return types are declared twice. These two type declarations must be identical (the parameter names can differ). The following function checks this:

```
fn tc_check_fn_parameters(forward_parameters: &Symbol, parameters: &Symbol
                                                                    ) {
  if forward_parameters == 0null { return; }
  while forward_parameters != 0null && parameters != 0null {
    if *(forward_parameters+.sym_kind) != *(parameters+.sym_kind) ||
        *(forward_parameters+.sym_type) != *(parameters+.sym_type) ||
        *(forward_parameters+.sym_dim) != *(parameters+.sym_dim) {
      panic(58);
    }
    forward_parameters = *(forward_parameters+.sym_next);
    parameters = *(parameters+.sym_next);
  }
  if forward_parameters != 0null || parameters != 0null { panic(59); }
```

```
  return;
}
```

It is used in the next function, which is updated to parse the function parameter and return types, and to build a corresponding list of symbols in `symbols` (as shown in Figure 19.2). If the function was previously declared its `function` symbol's `type` already contains a list of parameter and return types, which are checked with the above function.

```
fn tc_parse_fn_parameters(self: &Compiler, function: &Symbol) -> u32 {
  let i = 0;
  let name: &u32 = 0null;
  let length = 0;
  let type: &Symbol = 0null;
  let dim = 0;
  let symbols: &Symbol = 0null;
  tc_parse_token(self, '(');
  while *(self+.tc_next_token) != ')' {
    if i > 0 { tc_parse_token(self, ','); }
    name = tc_parse_identifier(self, &length);
    tc_parse_token(self, ':');
    type = tc_parse_type(self, &dim);
    symbols = tc_new_symbol(self, name, length, SYM_VARIABLE, i, type, dim
                                                                , symbols);
    i = i + 1;
  }
  tc_read_token(self);
  if *(self+.tc_next_token) == TC_ARROW {
    tc_read_token(self);
    type = tc_parse_type(self, &dim);
    symbols = tc_new_symbol(self, 0null, 0, SYM_VARIABLE, 0, type, dim, sy
                                                                mbols);
  } else {
    symbols = tc_new_symbol(self, 0null, 0, SYM_VOID, 0, 0null, 0, symbols
                                                                );
  }
  tc_check_fn_parameters(*(function+.sym_type), symbols);
  *(function+.sym_type) = symbols;
  *(self+.tc_fn_return_type) = symbols;
  return i;
}
```

The following function is updated to parse the new syntax for imported functions and for local constants, and to generate a `return` at the end of void functions if it might be reachable. Finally, since `tc_parse_fn_parameters` no longer adds symbols for the function parameters in the *symbols* list (they are added to the `function`'s `type` instead), a new loop is added to add a copy of these symbols in this list.

```
fn tc_parse_fn_body(self: &Compiler, function: &Symbol, arity: u32) {
  if *(self+.tc_next_token) == ';' {
```

```
    tc_read_token(self);
    *(function+.sym_kind) = SYM_FORWARD_FN;
    *(function+.sym_value) = 0;
    return;
  }
  if *(self+.tc_next_token) == '=' {
    tc_read_token(self);
    *(function+.sym_value) = tc_parse_integer(self) + 786432;
    tc_parse_token(self, ';');
    return;
  }
  let parameter = *(*(function+.sym_type)+.sym_next);
  while parameter != 0null {
    tc_add_symbol(self, *(parameter+.sym_name), *(parameter+.sym_length),
        SYM_VARIABLE, *(parameter+.sym_value), *(parameter+.sym_type), *(p
                                                              arameter+.sym_dim));
    parameter = *(parameter+.sym_next);
  }
  tc_parse_token(self, '{');
  tc_write_fn_insn(self, arity);
  let next_variable = arity + 4;
  let state = END_REACHABLE;
  while *(self+.tc_next_token) != '}' {
    if state == END_UNREACHABLE { panic(40); }
    if *(self+.tc_next_token) == TC_CONST {
      tc_parse_const(self);
    } else if *(self+.tc_next_token) == TC_LET {
      next_variable = tc_parse_let_stmt(self, next_variable);
    } else {
      state = tc_parse_stmt(self, 0null);
    }
  }
  if state == END_REACHABLE {
    if *(*(self+.tc_fn_return_type)+.sym_kind) != SYM_VOID { panic(41); }
    tc_write_return_insn(self);
  }
  tc_read_token(self);
  return;
}

fn tc_check_symbols(symbol: &Symbol, end_symbol: &Symbol) {
  while symbol != end_symbol {
    if *(symbol+.sym_kind) == SYM_FORWARD_FN { panic(32); }
    symbol = *(symbol+.sym_next);
  }
  return;
}
```

The next function is updated to save the *heap* and *symbols* variables *after* parsing the function parameters. Indeed this step now adds symbols for the parameter and return types, which must not be deleted after the function has been compiled.

```
fn tc_parse_fn(self: &Compiler) {
  tc_parse_token(self, TC_FN);
  let function = tc_parse_fn_name(self);
  let arity = tc_parse_fn_parameters(self, function);
  let heap = *(self+.tc_heap);
  let symbols = *(self+.tc_symbols);
  tc_parse_fn_body(self, function, arity);
  *(self+.tc_symbols) = symbols;
  *(self+.tc_heap) = heap;
  return;
}
```

Finally, the last two functions are updated to take the new "program" rule into account, and to initialize the new *dst_limit* and *heap_limit* variables (*fn_return_type* is set in `tc_parse_fn_parameters` and thus does not need to be initialized).

```
fn tc_parse_program(self: &Compiler) {
  loop {
    if *(self+.tc_next_token) == TC_FN {
      tc_parse_fn(self);
    } else if *(self+.tc_next_token) == TC_STRUCT {
      tc_parse_struct(self);
    } else if *(self+.tc_next_token) == TC_STATIC {
      tc_parse_static(self);
    } else if *(self+.tc_next_token) == TC_CONST {
      tc_parse_const(self);
    } else {
      if *(self+.tc_next_token) != 0 { panic(23); }
      tc_check_symbols(*(self+.tc_symbols), 0null);
      return;
    }
  }
}

fn tc_main(src_buffer: &u32, dst_buffer: &u32, flash_buffer: &u32) -> u32 ≀
                                                                      ≀{
  let error = 0;
  let compiler = (dst_buffer + 12288) as &Compiler;
  *(compiler+.tc_src) = src_buffer + 3;
  *(compiler+.tc_src_end) = src_buffer + 4 + *src_buffer;
  *(compiler+.tc_dst) = dst_buffer + 4;
  *(compiler+.tc_dst_limit) = compiler as &u32;
  *(compiler+.tc_heap) = *(compiler+.tc_dst_limit) + sizeof_compiler(Compi≀
                                                                  ≀ler);
  *(compiler+.tc_heap_limit) = *(compiler+.tc_heap) + 18432;
```

```
  *(compiler+.tc_symbols) = 0null;
  *(compiler+.tc_flash_offset) = dst_buffer - flash_buffer;
  let panic3 = 0;
  let panic2 = 0;
  let panic1 = 0;
  let panic0 = 0;
  error = panic_result(&panic0);
  if error != 0 {
    *dst_buffer = *(compiler+.tc_src) - src_buffer - 4;
    return error;
  }
  tc_read_char(compiler);
  tc_read_token(compiler);
  tc_parse_program(compiler);
  *dst_buffer = *(compiler+.tc_dst) - dst_buffer - 4;
  return 0;
}
```

19.4 Compilation and tests

To compile the above source code proceed as follows (see also Figure 16.4).

Edit v1 In the command editor, type "F3"+"r" and "F4"+"r" to load and edit the current compiler version. Then update it to the 1^{st} version of the types compiler. For convenience, we also provide this code in the `types_compiler_v1.txt` file in https://ebruneton.github.io/toypc/sources.zip. When you are done, exit the text editor and type "F5"+"r" to save your work. Alternatively, you can "cheat" by running the following command on an external computer (see Section 16.4 for more details):

```
user@host:~$ python3 flash_helper.py < part3/types_compiler_v1.txt
```

Compile v1 In the command editor, type "F6"+"r" to compile the code you typed. If all goes well, after about 4 seconds, you should get a result equal to 0 (meaning that no error was found). If this is not the case use Appendix D to get the error code meaning, fix this error, save the program and compile it again. Repeat this process until the compilation is successful. Then type "F7"+"r" to save the result.

Test v1 Type "F2"+"r" to create a new program, "F4"+"r" to edit it, and type the following small test program, which computes the factorial of 6:

```
fn factorial(n: u32) -> u32;
fn test() -> u32 { return factorial(6); }
fn factorial(n: u32) -> u32 {
  if n == 0 { return 1; }
  return factorial(n - 1) * n;
}
```

Then type "F9"+"r" to run it. If the result is not 720 = $2D0_{16}$ this means that the compiler is wrong. In this case, type "F8"+"r" to restore the statements compiler. Then repeat the previous steps and double check everything until this test passes.

Edit v2 Type "F3"+"r" to load the 1^{st} version of the types compiler and "F4"+"r" to edit it. Then update it to the 2^{nd} version. For convenience, we also provide this code in the `types_compiler_v2.txt` file. Then save this new version with the F5 command. Alternatively, run the following command on an external computer:

```
user@host:~$ python3 flash_helper.py < part3/types_compiler_v2.txt
```

Compile v2 Type"F6"+"r" to compile this new code. The result should be 0, meaning "no error". If this is not the case, repeat the "Edit v2" and "Compile v2" steps until all errors are fixed.

Test v2 As in the previous chapter, the compiled code of the 2^{nd} version of the types compiler is not identical to that of the 1^{st} version. One reason, in particular, is that the code of the 1^{st} version, obtained with the statements compiler, does not use `ret` instructions. Whereas the 2^{nd} version uses such instructions for void functions.

However, the two versions should produce the same compiled code for the same input program, since they are supposed to be functionally equivalent. We can thus use this property to test the 2^{nd} version, as in the previous chapter (see Section 18.4):

- Type "F7"+"r" to store the bytecode of 2^{nd} version, produced by the 1^{st} version.
- Type "F6"+"r" to compile the 2^{nd} version with itself.
- Type "F10"+"r" to compare the results of the previous commands. If the result is not 0 this means that the 2^{nd} version is wrong[2]. Type "F8"+"r" to restore the 1^{st} version and repeat the steps from "Edit v2" until this test passes.

[2]As noted in the previous chapter, the 1^{st} version might also be wrong despite the "Test v1" step.

CHAPTER

20 Native Compiler

Our toy programming language is now quite usable. We could continue to add new features to make it even more usable but at some point it would stop being a toy language, and this is not our goal. Instead of improving the language, this chapter improves its compiler in order to generate more efficient code.

All the programs we wrote before having the first version of our compiler use bytecode instructions, because writing native Cortex M3 code manually is very hard. Now that we have a compiler, we no longer need to write any bytecode or native code manually. Thus, there is no reason to continue using bytecode instructions, in particular for the code generated by our compiler. On the contrary, since bytecode programs are about 10 times slower than native ones when run with our bytecode interpreter, it would be better to generate native code. This chapter refactors our compiler in order to do this.

20.1 Requirements

In this chapter we require our compiler to compile programs into native Cortex M3 instructions (see Section 7.3). We also require the generated code to be *position independent*. As explained in Section 13.3, this means that this code should work whatever the address at which it is run. This is not the case of the bytecode produced by the current compiler, which only works at the $flash\_buffer$ address used to obtain it (in particular because of the `call` instructions).

The above requirements can be achieved by using only a subset of the instructions presented in Section 7.3. In some rare cases however, in particular in the next part, some special Cortex M3 instructions are needed. To enable this we add a last requirement, namely the possibility to write functions directly in native code. For this we could define a syntax for Cortex M3 instructions, such as "`MOV PC LR;`" to denote the instruction copying the Link Register into the Program Counter. But this would greatly increase the size of our compiler, for a rare use case. Instead, to simplify, we require these instructions to be given directly in encoded form, between square brackets. For instance, compiling "`fn do_nothing() [ 18167; ]`" should produce a function reduced to the single "`MOV PC LR`" instruction (whose encoding is 18167). This syntax is not user friendly but it should rarely be used. Comments can also make it easier to understand: "`fn do_nothing() [ /*MOV_PC_LR*/18167; ]`".

20.2 Algorithms

The above requirements do not introduce any new token. Hence no changes are needed in the scanner. The new syntax can be specified with a simple grammar rule, used in `fn_body` (unchanged parts of the grammar are in gray or not shown):

fn_body: "{" (const | let_stmt | stmt)* "}" | fn_asm_body | ";"
fn_asm_body: "[" (INTEGER ";")* "]"

Note that we removed the syntax for imported functions. Indeed, it is no longer possible to import, with a fixed address, a function from a program using position independent code, which is not guaranteed to be at a fixed location.

The algorithm to parse and compile the `fn_asm_body` rule is trivial (by definition, each instruction simply needs to written, as is, in the dst buffer). Hence, the only new algorithms which are needed are in the backend part. They are presented below.

20.2.1 Register stack

The Cortex M3 instructions use registers to perform computations. Unfortunately there is only a small number of registers. They can only store a very small subset of all the values computed by a program. Hence, a native program needs to repeatedly copy values from memory to registers, do some computation with them, and copy the results back to memory. For instance, our bytecode interpreter does this *for each bytecode instruction*. But copying values between memory and the registers is slow, and this is a reason why our interpreter is inefficient. To reduce the number of copies we need to keep values in registers as long as possible, and to use all the registers (our interpreter uses only 2 registers for actual computations, namely R0 and R1). Unfortunately, finding an optimal way to use registers in a function is a complex problem. The best known algorithms to solve it take a time which increases exponentially with the size of the function. This makes them impossible to use in practice. Instead, compilers use fast algorithms, called *heuristics*, which compute a "good enough" solution.

In this section we define a very simple strategy to use the registers. It gives a solution which is far from the optimal, and from the results of the best compiler heuristics, but better than copying values back and forth for each arithmetic or logic operation. The idea is to use the registers, other than the SP, LR, and PC, as a stack:

- at any point in time, only the first n registers, R0 to R$n-1$, contain actual values.
- new values are "pushed" by storing them in Rn, and incrementing n by 1.
- values which are no longer needed are "popped" by decrementing n by 1.

For instance, to compute $(1+2)*(3+4)$ we can push 1 in R0, push 2 in R1, pop two values and push their sum in R0, push 3 in R1, 4 in R2, pop two values and push their sum in R1, and finally pop two values and push their product in R0 (see Figure 20.1). This idea works well for expressions, because intermediate results are discarded in the reverse order of their creation. But this is not the case, in general, of

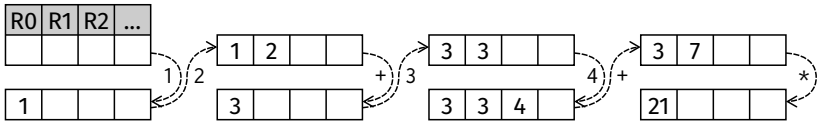


FIGURE 20.1 Using the registers as a stack to compute $(1 + 2) * (3 + 4)$.

function parameters or local variables. We thus use the register stack only to compile expressions. And we keep function parameters and local variables in the real stack. As a consequence, the register stack is always empty at the start of each statement.

20.2.2 Stack frames

With the above strategy, a statement such as "`let x = f(1, 2);`" is compiled into code "pushing" 1 and 2 in R0 and R1, before calling "f". And, by hypothesis, the effect of this call should be to "pop" R0 and R1 and to "push" the result in R0. On the other hand, during `f`'s execution, its parameters must be on the stack, as defined above. We deduce from this that the function arguments should be pushed on the real stack at some point. Either the caller or the callee can do this, but doing this in the callee saves code (there can be several call expressions for a single function).

In order to allow the callee to return at the correct instruction, the caller should use a Branch with Link instruction, as explained in Section 7.2. And the callee should copy the LR into the PC to return into the caller. As indicated in Section 7.2.4, this can be done by pushing the LR on the stack at the beginning of each function, and by popping it into the PC to return from this function.

We implement the above choices with a single PUSH instruction at the beginning of each function. This instruction pushes the registers R0 to R$a - 1$, and the LR, onto the stack, where a is the function's arity. This has several consequences:

- each stack frame contains, from bottom to top, the saved Link Register value, the function parameters *in reverse order*, and the local variables in the order they are declared (see Figure 20.2). The function parameters are in reverse order due to the way the PUSH instruction works (see Figure 7.4).
- the stack frame slot index of the i^{th} function parameter is $a - i$, instead of i in the bytecode interpreter. And the stack frame slot index of the i^{th} local variable is $a + 1 + i$, instead of $a + 4 + i$ in the interpreter (see Figure 20.2).
- function call expressions must always "push" their arguments in R0 to R$a - 1$ and not, for instance, in R1 to Ra. This is discussed in Section 20.2.4.

20.2.3 Function parameters and local variables

In order to use a function parameter or a local variable with stack frame slot index s we first need to compute its address on the stack. For this two methods can be used. We can either subtract $4s$ from the Frame Pointer, or add $4(frame\_size - s)$ to the

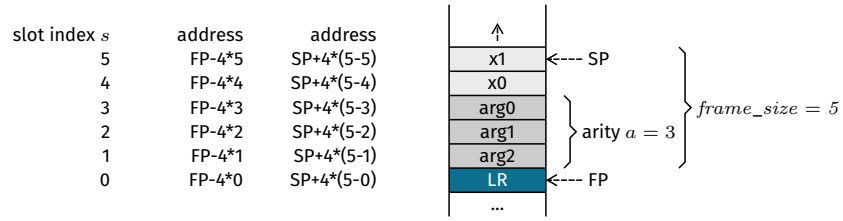


FIGURE 20.2 Each stack frame contains the saved Link Register (blue), the function parameters in reverse order (gray), and the local variables (light gray). The address of each slot s can be computed with FP$-4s$ or SP$+4(frame_size - s)$ (left).

Stack Pointer, where $frame\_size$ is the number of slots in the stack frame, minus 1 (see Figure 20.2). The bytecode interpreter uses the first method because this avoids updating $frame\_size$ each time a value is pushed or popped from the stack. Here however we don't need to update it *during execution*. Instead, the compiler can keep track of its value *during compilation*. Indeed, $frame\_size$ is initially equal to the function's arity a, and increases by one after each `let` statement. We thus use the second method, which has the advantage of not requiring a Frame Pointer register.

20.2.4 Function calls

As explained above, a function call must be compiled into code pushing the arguments in the R0 to R$a-1$ registers, where a is the function's arity. This is what naturally happens for a statement such as "`f(1, 2);`", but not for "`f(1, g(2, 3));`". In this case, before the call to g, 2 and 3 would be R1 and R2 instead of R0 and R1 (because R0 contains 1). To solve this issue we emit code saving the register stack on the real stack before compiling a call expression. And we emit code restoring these registers after the call returns.

A consequence of the register stack algorithm is that a "`return` e`;`" statement is always compiled into code pushing e's value in R0. This works fine in a case such as "`let x = f(1, 2);`", which is supposed to pop 1 and 2 from R0 and R1, and to push the result in R0. But this does not work for the nested call in "`f(1, g(2, 3));`". In this case, we want the result of g in R1, because the arguments of `f` must be in R0 and R1. To solve this issue we emit code copying R0 into R1, before restoring R0's value from the real stack.

In summary, a function call is compiled as follows, where n is the size of the register stack, t is the function's return type, and a its arity (see Figure 20.3):

1. set n_0 to n
2. if $n > 0$ emit code to PUSH R0...R$n-1$; increase $frame\_size$ by n; set n to 0
3. emit code to compute and "push" the arguments in R0...R$a-1$
4. emit a Branch with Link instruction to call the function
5. if t is void set n to n_0. Otherwise write code to copy R0 in Rn_0; set n to $n_0 + 1$
6. if $n_0 > 0$ generate code to POP R0...R$n_0 - 1$; decrease $frame\_size$ by n_0

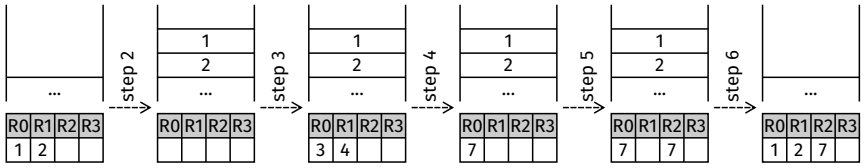


FIGURE 20.3 From left to right, the call to `g` in `f(1, 2, g(3, 4))` saves `f`'s arguments on the stack, evaluates `g`'s arguments in R0 and R1, calls `g`, copies its result from R0 to R2, and restores `f`'s arguments from the stack.

Note that this algorithm does not work if a is larger than the number of available registers. More generally, the register stack algorithm does not work for complex expressions requiring more registers than available. In such cases, to simplify the compiler, we simply raise an error. The user should then refactor their code to avoid it.

In theory there are 13 available registers (the other 3 are used for the SP, LR, and PC). In practice, however, we restrict ourselves to only 8, because using registers R8 to R12 generally requires 32 bits instructions with complex encodings.

20.2.5 Label and function placeholders and symbols

An `ifeq` instruction is encoded as one byte for the opcode, and 2 bytes for the jump offset. When this offset is not yet known, a 16 bits placeholder is used instead. The equivalent native code requires a "CMP IT B" instruction sequence, the last one containing an 11 bits offset. To simplify we keep 16 bits placeholders, which are thus now filled with a complete B instruction (instead of its offset only).

Similarly, a `call` instruction is encoded as one byte for the opcode, and 2 bytes for the argument. When this argument is not yet known, a 16 bits placeholder is used instead. The equivalent native code requires a 32 bits BL instruction, containing a 22 bits offset split in two parts. Here again, to simplify, we use 32 bits placeholders for complete BL instructions. This allows us to store, in each placeholder, the previous placeholder's address, instead of its offset.

Currently the value of a function name symbol is the argument to use in a `call` instruction to call this function. This cannot be generalized to the argument to use in a BL instruction, because this argument depends on the instruction's address (see Section 7.3.3). Hence we redefine the value of a function name symbol as the dst address of the function's first instruction.

20.2.6 Error handling

The current implementation of the `panic` and `panic_result` functions assumes that each stack frame contains 4 saved register values. Since this is not the case of the layout depicted in Figure 20.2, we need to re-implement these functions. The general idea stays the same: `panic_result` should save everything which is needed for `panic`

to return directly in the `panic_result` caller. This includes the LR, the SP (which replaces the FP), and nothing else. Saving the LR and the SP values is easy, but restoring the SP is not possible unless `panic` is written directly in native code. To simplify we implement both functions in native code, and we save the SP and LR registers in R11 and R12, respectively (they are not used anywhere else since the register stack is restricted to R0. . .R7).

20.3 Implementation

We can now extend our compiler in order to generate native code. As usual, this must be done in two steps, first without using native code, then with it. To save space, we give the two compiler versions at the same time.

Since the compiler must generate position independent code it no longer needs the *flash_buffer* parameter, which is thus removed from the main function (we keep it, unused, in the 1^{st} version, to simplify the compilation steps in Section 20.4):

```
fn tc_main(src_buffer: &u32, dst_buffer: &u32) -> u32;
fn main(src_buffer: &u32, dst_buffer: &u32, flash_buffer: &u32) -> u32 {
  return tc_main(src_buffer, dst_buffer);
}
```

The following functions are rewritten in native code in the 2^{nd} version. This is not necessary but it illustrates how *native functions* can be implemented. An important thing to note is that native functions do not have a stack frame by default. This is because they don't have a PUSH instruction to create this stack frame (see Section 20.2.2), unless one is added explicitly. Most of the time this is not necessary. Consider for instance the `load8` function. When it is called its argument is in R0 by hypothesis, and the LR contains the return address. The result must be put in R0, which can be done with a single LDRB instruction (see Section 7.3.2). Returning to the caller is then done by copying the LR into the PC. The 3 other functions are implemented in the same way.

```
fn load8(ptr: &u32) -> u32 { return (*ptr) & 255; }
fn load16(ptr: &u32) -> u32 { return (*ptr) & 65535; }
fn store8(ptr: &u32, value: u32) { *ptr = (*ptr) & 4294967040 | value; }
fn store16(ptr: &u32, value: u32) { *ptr = (*ptr) & 4294901760 | value; }
fn load8(ptr: &u32) -> u32 [ /*LDRB_R0_R0_0*/30720; /*MOV_PC_LR*/18167; ]
fn load16(ptr: &u32) -> u32 [ /*LDRH_R0_R0_0*/34816; /*MOV_PC_LR*/18167; ]
fn store8(ptr: &u32, value: u32) [ /*STRB_R1_R0_0*/28673; /*MOV_PC_LR*/181⌇
                                                                    ⌇67; ]
fn store16(ptr: &u32, value: u32) [ /*STRH_R1_R0_0*/32769; /*MOV_PC_LR*/18⌇
                                                                   ⌇167; ]
```

As explained above, `panic_result` copies the SP and the LR in R11 and R12. Since it does not have its own stack frame, this copies the correct SP value, *i.e.*, the one from the caller. It then returns 0 in R0. When `panic` is called *error* is in R0, which is where the `panic_result` caller expects it. Hence `panic` only needs to restore the SP from R11, and to return in the `panic_result` caller by copying R12 into the PC.

```
const PANIC_BUFFER: &&u32 = 1074666152;
fn panic_copy(src: &u32, dst: &u32) {
  *dst = *src;
  *(dst + 4) = *(src + 4);
  *(dst + 8) = *(src + 8);
  *(dst + 12) = *(src + 12);
}
fn panic_result(ptr: &u32) -> u32 {
  panic_copy(&ptr as &u32 - 16, ptr);
  *PANIC_BUFFER = ptr;
  return 0;
}
fn panic(error: u32) -> u32 {
  panic_copy(*PANIC_BUFFER, &error - 16);
  return error;
}
fn panic_result() -> u32 [
  /*MOV_R11_SP*/18155;
  /*MOV_R12_LR*/18164;
  /*MOV_R0_0*/8192;
  /*MOV_PC_LR*/18167;
]
fn panic(error: u32) [ /*MOV_SP_R11*/18141; /*MOV_PC_R12*/18151; ]
```

20.3.1 Shared constants and data structures

This part is unchanged except at the very end, where `flash_offset` is removed from the `Compiler` struct, and replaced with `next_register` and `frame_size`. The former contains the register stack size noted n in Section 20.2.1. The latter holds the *frame_size* variable defined in Section 20.2.3. Unchanged parts are not shown.

```
  fn_return_type: &Symbol,
  next_register: u32,
  frame_size: u32
}
```

20.3.2 Backend

The scanner is completely unchanged and is not shown. The first 4 backend functions, from `mem_allocate` to `tc_write32`, are unchanged and not shown. But the rest of the backend is almost entirely rewritten. We start with low level functions to encode and write individual instructions. Native instructions must always be stored at even addresses. The following function writes a 0 byte if *dst* is odd, to make it even:

```
fn tc_write_padding(self: &Compiler) {
  if self.dst as u32 & 1 == 1 { tc_write8(self, 0); }
}
```

If $top = 0$ the following function writes a MOVW: $\mathrm{R}z \leftarrow c_3{:}c_2{:}c_1{:}c_0$ instruction to store the 16 least significant bits of c in $\mathrm{R}z$. If $top = 1$ is writes a MOVT instruction instead. It computes the subparts c_0, c_1, c_2, and c_3 of c as described in Section 7.3.1, and shifts them to their correct location. For instance, "`(c >> 8) & 7`" computes c_1, which is then shifted by 28 bits ("`& 1`", "`& 7`", "`& 15`", and "`& 255`" keep only the 1, 3, 4 and 8 least significant bits of the left hand side, respectively).

```
fn tc_write_move16_insn(self: &Compiler, z: u32, c: u32, top: u32) {
  tc_write32(self, /*MOVW_RZ_C*/62016 | (((c >> 8) & 7) << 28) |
      (z << 24) | ((c & 255) << 16) | (((c >> 11) & 1) << 10) |
      (top << 7) | ((c >> 12) & 15));
}
```

The next function writes an ADD: $\mathrm{R}z \leftarrow \mathrm{R}x + \mathrm{R}y$ or SUB: $\mathrm{R}z \leftarrow \mathrm{R}x - \mathrm{R}y$ instruction with $x = z$. op must be the instruction's encoding for $x = y = z = 0$.

```
fn tc_write_add_or_sub_insn(self: &Compiler, op: u32, z: u32, y: u32) {
  tc_write16(self, op | (y << 6) | (z << 3) | z);
}
```

This function writes an ADD: $\mathrm{R}z \leftarrow \mathrm{R}z + c$ or ADD: $\mathrm{R}z \leftarrow \mathrm{SP} + 4 * c$ instruction. op must be the instruction's encoding for $z = c = 0$.

```
fn tc_write_add_const_insn(self: &Compiler, op: u32, z: u32, c: u32) {
  if c > 255 { panic(100); }
  tc_write16(self, op | (z << 8) | c);
}
```

The following function is similar. It can be used to write a MUL: $\mathrm{R}z \leftarrow \mathrm{R}z * \mathrm{R}x$, AND: $\mathrm{R}z \leftarrow \mathrm{R}z \wedge \mathrm{R}x$, ORR: $\mathrm{R}z \leftarrow \mathrm{R}z \vee \mathrm{R}x$, LSL: $\mathrm{R}z \leftarrow \mathrm{R}z \ll (\mathrm{R}x \ mod \ 32)$ or LSR: $\mathrm{R}z \leftarrow \mathrm{R}z \gg (\mathrm{R}x \ mod \ 32)$ instruction:

```
fn tc_write_math_insn(self: &Compiler, op: u32, z: u32, x: u32) {
  tc_write16(self, op | (x << 3) | z);
}
```

The next function is intended to be used for LDR: $\mathrm{R}z \leftarrow \mathrm{mem32}[\mathrm{SP} + 4 * c]$ and STR: $\mathrm{R}z \rightarrow \mathrm{mem32}[\mathrm{SP} + 4 * c]$. The following one is designed for the LDR: $\mathrm{R}z \leftarrow \mathrm{mem32}[\mathrm{R}x + 4 * c]$ and STR: $\mathrm{R}z \rightarrow \mathrm{mem32}[\mathrm{R}x + 4 * c]$ instructions.

```
fn tc_write_stack_insn(self: &Compiler, op: u32, z: u32, c: u32) {
  if c > 255 { panic(101); }
  tc_write16(self, op | (z << 8) | c);
}
fn tc_write_heap_insn(self: &Compiler, op: u32, z: u32, x: u32, c: u32) {
  if c > 31 { panic(102); }
  tc_write16(self, op | (c << 6) | (x << 3) | z);
}
```

The following function returns the encoding of a B: $\mathrm{PC} \leftarrow \mathrm{PC} + signed_{12}(2 * c)$ instruction jumping to $target$, to be written at dst. For this it uses $c = (target - pc)/2$,

where $pc = dst + 4$ is the PC value when the instruction is executed. The next one does the same for a BL: $\mathrm{PC} \leftarrow \mathrm{PC} + signed_{23}(2 * c_1{:}c_0), \mathrm{LR} \leftarrow a + 5$ instruction. Note that these functions return position independent code, because the offset between dst and $target$ does not change if the code is moved elsewhere.

```
fn tc_get_branch_insn(dst: &u32, target: &u32) -> u32 {
  let pc = dst + 4;
  if target >= pc + 2048 || pc > target + 2048 { panic(103); }
  return /*B_2C*/57344 | (((target - pc) & /*(1<<12)-1*/4095) >> 1);
}
fn tc_get_branch_with_link_insn(dst: &u32, target: &u32) -> u32 {
  let pc = dst + 4;
  if target >= pc + 4194304 || pc > target + 4194304 { panic(104); }
  let offset = (target - pc) & /*(1<<23)-1*/8388607;
  return /*BL_2C*/4160811008 | ((offset & 4095) << 15) | (offset >> 12);
}
```

The next function is unchanged, but `tc_fill_placeholders` is rewritten for function call placeholders only. As described above, these placeholders are now 32 bits wide, contain the address of the previous placeholder (or `null` for the first one), and are filled with a complete BL instruction.

```
fn tc_write_placeholder(self: &Compiler, placeholder_p: &&u32) {
  let new_placeholder = self.dst;
  let last_placeholder = *placeholder_p;
  *placeholder_p = new_placeholder;
  if last_placeholder == null { last_placeholder = new_placeholder; }
  tc_write16(self, new_placeholder - last_placeholder);
}
fn tc_fill_placeholders(placeholder: &u32, value: u32) {
  let previous: &u32 = null;
  while placeholder != null {
    previous = *placeholder as &u32;
    *placeholder = tc_get_branch_with_link_insn(placeholder, value as &u32
                                                              );
    placeholder = previous;
  }
}
```

As a consequence, `tc_fill_label_placeholders`, which was calling the above function, is rewritten to keep 16 bits placeholders. However, it now fills them with complete B instructions, as explained above.

```
fn tc_fill_label_placeholders(self: &Compiler, placeholder: &u32) {
  let offset = 0;
  while placeholder != null {
    offset = load16(placeholder);
    store16(placeholder, tc_get_branch_insn(placeholder, self.dst));
    if offset == 0 { break; }
```

```
    placeholder = placeholder - offset;
  }
}
```

We then add two functions related to the register stack. The first one increments the register stack size and returns its previous value. The second one writes code to push on the stack the first n registers, R0 to R$n-1$, and the LR if `link_register` $= 1$. It then increments $frame\_size$ by n and clears the register stack.

```
const MAX_REGISTERS: u32 = 8;
fn tc_new_register(self: &Compiler) -> u32 {
  let register = self.next_register;
  if register >= MAX_REGISTERS { panic(105); }
  self.next_register = register + 1;
  return register;
}
fn tc_save_registers(self: &Compiler, n: u32, link_register: u32) {
  tc_write16(self, /*PUSH*/46080 | (link_register << 8) | ((1 << n) - 1));
  self.frame_size = self.frame_size + n;
  self.next_register = 0;
}
```

The remaining functions are not new but are rewritten in order to generate native code, thanks to the above functions. The first one writes code to push $value$ on the register stack, while trying to minimize the size of the generated code.

```
fn tc_write_cst_insn(self: &Compiler, value: u32) {
  let register = tc_new_register(self);
  if value < 256 {
    tc_write16(self, /*MOV_RZ_C*/8192 | (register << 8) | value);
  } else {
    tc_write_move16_insn(self, register, value, 0);
    if (value >> 16) != 0 {
      tc_write_move16_insn(self, register, value >> 16, 1);
    }
  }
}
```

The next function generates code to push dst on the register stack. In order to get position independent code, an ADR: R$z \leftarrow \lfloor \mathrm{PC} \rfloor_4 - c_2{:}c_1{:}c_0$ instruction is used with the $offset$ between itself and dst as argument (which does not change if the code is moved). As required by ADR, $offset$ is computed as $\lfloor \mathrm{PC} \rfloor_4 - dst$, where PC is the instruction's address plus 4.

```
fn tc_write_static_insn(self: &Compiler, dst: u32) {
  let z = tc_new_register(self);
  let offset = (((self.dst as u32 + 4) >> 2) << 2) - dst;
  if offset > 4095 { panic(106); }
  tc_write32(self, /*ADR_RZ_C*/62127 | (((offset >> 8) & 7) << 28) |
      (z << 24) | ((offset & 255) << 16) | (((offset >> 11) & 1) << 10));
}
```

The next function writes code which pops the top 2 registers Rz and Ry, combines them with the operation corresponding to $token$, and pushes the result back in Rz.

```
fn tc_write_binary_insn(self: &Compiler, token: u32) {
  let y = self.next_register - 1;
  let z = y - 1;
  if token == TC_ADD {
    tc_write_add_or_sub_insn(self, /*ADD_RZ_RX_RY*/6144, z, y);
  } else if token == TC_SUB {
    tc_write_add_or_sub_insn(self, /*SUB_RZ_RX_RY*/6656, z, y);
  } else if token == TC_MUL {
    tc_write_math_insn(self, /*MUL_RZ_RZ_RX*/17216, z, y);
  } else if token == TC_DIV {
    tc_write32(self, /*UDIV_RZ_RX_RY*/4042324912 | z << 24 | y << 16 | z);
  } else if token == TC_BIT_AND {
    tc_write_math_insn(self, /*AND_RZ_RZ_RX*/16384, z, y);
  } else if token == TC_BIT_OR {
    tc_write_math_insn(self, /*ORR_RZ_RZ_RX*/17152, z, y);
  } else if token == TC_SHIFT_LEFT {
    tc_write_math_insn(self, /*LSL_RZ_RZ_RX*/16512, z, y);
  } else {
    tc_write_math_insn(self, /*LSR_RZ_RZ_RX*/16576, z, y);
  }
  self.next_register = y;
}
```

A conditional jump requires 3 instructions in native code. A CMP and an IT instruction are needed to skip a B instruction (here a placeholder) if the top 2 registers do not meet the jump condition (these registers are necessarily R0 and R1 since boolean expressions can't be used in other expressions). `IF_THEN_CONDITION` contains the IT arguments corresponding to the $<$, $=$, $>$, $\leq$, $\neq$, and $\geq$ conditions, respectively (see Section 7.3.4).

```
static IF_THEN_CONDITION = [48, 0, 128, 144, 16, 32];
fn tc_write_jump_insn(self: &Compiler, token: u32, placeholder_p: &&u32) {
  tc_write16(self, /*CMP_R0_R1*/17032);
  tc_write16(self, /*IT*/48904 | load8(IF_THEN_CONDITION + token - TC_LT));
  tc_write_placeholder(self, placeholder_p);
  self.next_register = 0;
}
fn tc_write_goto_insn(self: &Compiler, placeholder_p: &&u32) {
  tc_write_placeholder(self, placeholder_p);
}
fn tc_write_loop_insn(self: &Compiler, loop_dst: &u32) {
  tc_write16(self, tc_get_branch_insn(self.dst, loop_dst));
}
```

On the other hand, loading the value of a field can be done with a single native instruction, instead of 3 bytecode instructions. Indeed a single LDR instruction can

add $4.field$ to the struct's address, popped from the register stack, and push back the value at the resulting address.

```
fn tc_write_load_insn(self: &Compiler, field: u32) {
  let z = self.next_register - 1;
  tc_write_heap_insn(self, /*LDR_RZ_RX_4C*/26624, z, z, field);
}
```

Erasing a load instruction was previously erasing only the 3^{rd} instruction written by `tc_write_load_insn`. This was leaving the field's address on the stack which, consequently, didn't have to be recomputed to store a value in it. Erasing a load instruction now deletes it completely. The field address must thus be recomputed in the following function. This is done with a STR instruction, adding $4.field$ to the struct's address in Rx, and storing Rz at the resulting address. Since `tc_write_store_insn` is only used to compile assignments, and since the register stack is always empty at the start of a statement, we necessarily have $x = 0$ and $z = 1$.

```
fn tc_erase_load_insn(self: &Compiler) { self.dst = self.dst - 2; }
fn tc_write_store_insn(self: &Compiler, field: u32) {
  tc_write_heap_insn(self, /*STR_RZ_RX_4C*/24576, 1, 0, field);
  self.next_register = 0;
}
```

For the same reason the following function, called after `tc_erase_load_insn`, and which previously had nothing to do, must now write an instruction adding $4.field$ to the top register:

```
fn tc_write_address_of_insn(self: &Compiler, field: u32) {
  let z = self.next_register - 1;
  if field != 0 {
    tc_write_add_const_insn(self, /*ADD_RZ_RZ_C*/12288, z, field << 2);
  }
}
```

As explained in Section 20.2.3 a function parameter or local variable is now accessed by adding $4(frame\_size - variable)$ to the SP, where $variable$ is its stack frame slot index. Note that the register stack size is always 1 when `tc_write_set_insn` and `tc_write_pop_insn` are called (they are only used to compile assignment and expression *statements*, respectively).

```
fn tc_write_ptr_insn(self: &Compiler, variable: u32) {
  tc_write_add_const_insn(self, /*ADD_RZ_SP_4C*/43008,
      tc_new_register(self), self.frame_size - variable);
}
fn tc_write_get_insn(self: &Compiler, variable: u32) {
  tc_write_stack_insn(self, /*LDR_RZ_SP_4C*/38912,
      tc_new_register(self), self.frame_size - variable);
}
fn tc_erase_get_insn(self: &Compiler) {
  self.dst = self.dst - 2;
```

```
  self.next_register = self.next_register - 1;
}
fn tc_write_set_insn(self: &Compiler, variable: u32) {
  tc_write_stack_insn(self, /*STR_RZ_SP_4C*/36864, 0, self.frame_size - va⸘
                                                                  ⸘riable);
  self.next_register = 0;
}
fn tc_write_pop_insn(self: &Compiler) { self.next_register = 0; }
```

The next function writes a PUSH instruction at the beginning of a function to create its stack frame, initially containing the function parameters and the saved Link Register value. The value of a function name symbol is now *fn_dst* (see Section 20.2.5).

```
fn tc_write_fn_insn(self: &Compiler, arity: u32) {
  if arity > MAX_REGISTERS { panic(107); }
  self.frame_size = 0;
  tc_save_registers(self, arity, 1);
}
fn tc_get_fn_value(self: &Compiler, fn_dst: u32) -> u32 { return fn_dst; }
```

The next function implements steps 4 to 6 of the algorithm in Section 20.2.4 (the new *saved_registers* parameter corresponds to n_0; steps 1 to 3 are done by the caller).

```
fn tc_write_call_insn(self: &Compiler, function: &Symbol, saved_registers: ⸘
                                                                  ⸘ u32)
{
  let dst = self.dst;
  if function.kind == SYM_FORWARD_FN {
    tc_write32(self, function.value);
    function.value = dst as u32;
  } else {
    tc_write32(self, tc_get_branch_with_link_insn(dst, function.value as &⸘
                                                                  ⸘u32));
  }
  if function.type.kind != SYM_VOID {
    if saved_registers != 0 {
      tc_write16(self, /*MOV_RZ_R0*/17920 | saved_registers);
    }
    if saved_registers >= MAX_REGISTERS { panic(108); }
    self.next_register = saved_registers + 1;
  } else {
    self.next_register = saved_registers;
  }
  if saved_registers != 0 {
    tc_write16(self, /*POP*/48128 | ((1 << saved_registers) - 1));
    self.frame_size = self.frame_size - saved_registers;
  }
}
```

In order to return from a function the first step is to adjust the SP to point to the saved LR value, at the bottom of the function's stack frame. Popping this value into the PC then returns to the caller.

```
fn tc_write_return_insn(self: &Compiler) {
  let offset = self.frame_size;
  if offset > 127 { panic(109); }
  if offset != 0 { tc_write16(self, /*ADD_SP_SP_4C*/45056 | offset); }
  tc_write16(self, /*POP_PC*/48384);
  self.next_register = 0;
}
```

20.3.3 Parser

The parser is extended in order to support native functions. We also add the possibility to use function names in expressions, which evaluate to their address. Finally, a few parser functions need to be updated due to the changes in the backend. The first one is `tc_parse_static`. Native instructions must be stored at even addresses, but a `static` block may add an odd number of bytes between instructions. To fix this we add some padding if necessary (unchanged functions are not shown; an ellipsis replaces some unchanged statements):

```
fn tc_parse_static(self: &Compiler) {
  tc_parse_token(self, TC_STATIC);
...
  tc_parse_token(self, ';');
  tc_write_padding(self);
}
```

In order to support function names as `u32` expressions we add a special case for `SYM_FN` symbols in the following function:

```
fn tc_push_symbol_value(self: &Compiler, symbol: &Symbol) -> &Value {
...
  if symbol.kind == SYM_FN {
    return tc_push_value(self, origin, symbol.value, null, 0);
  }
  return tc_push_value(self, origin, symbol.value, symbol.type, symbol.dim
                                                                      );
}
```

The function below is updated to write code saving the register stack on the real stack (steps 1 and 2 in Section 20.2.4) before compiling the function arguments:

```
fn tc_parse_fn_arguments(self: &Compiler, function: &Symbol) {
...
  tc_parse_token(self, '(');
  let next_register = self.next_register;
  if next_register != 0 {
```

```
    tc_save_registers(self, next_register, 0);
  }
  let argument_count = 0;
  while self.next_token != ')' {
    if argument_count > 0 { tc_parse_token(self, ','); }
    tc_parse_expr(self);
    argument_count = argument_count + 1;
  }
  tc_read_token(self);
  tc_check_fn_arguments(self, function, argument_count);
  tc_write_call_insn(self, function, next_register);
}
```

Another change is needed in the following function in order allow SYM_FN symbols in expressions (which can be compiled in the same way as SYM_STATIC symbols).

```
fn tc_parse_primitive_expr(self: &Compiler) {
  let symbol: &Symbol = null;
...
      } else if symbol.kind == SYM_STATIC || symbol.kind == SYM_FN {
        tc_write_static_insn(self, symbol.value);
...
}
```

The function compiling a `let` statement must also be updated, in order to generate code pushing on the stack the right hand side value, in R0 (this value was previously produced directly on the stack).

```
fn tc_parse_let_stmt(self: &Compiler, variable: u32) -> u32 {
  tc_parse_token(self, TC_LET);
...
  tc_add_symbol(self, name, length, SYM_VARIABLE, variable, type, dim);
  tc_save_registers(self, 1, 0);
  return variable + 1;
}
```

A new function is added just before `tc_parse_fn_body`. It parses and compiles the new syntax for native functions. Each integer simply needs to be written in the *dst* buffer, using either 16 or 32 bits depending on its value (32 bits instructions have bits 11 to 15 greater than $11100_2 = 28$ – see Section A5.3 pA5-137 of [17]).

```
fn tc_parse_fn_asm_body(self: &Compiler) {
  let value = 0;
  tc_parse_token(self, '[');
  while self.next_token != ']' {
    value = tc_parse_integer(self);
    if (value >> 11) & 31 > 28 {
      tc_write32(self, value);
    } else {
      tc_write16(self, value);
```

```
    }
    tc_parse_token(self, ';');
  }
  tc_read_token(self);
}
```

It is used in the next function, which is also updated to compute the function parameter and local variable slot indices as described in Section 20.2.2.

```
fn tc_parse_fn_body(self: &Compiler, function: &Symbol, arity: u32) {
  if self.next_token == ';' {
...
  }
  if self.next_token == '[' {
    tc_parse_fn_asm_body(self);
    return;
  }
  let parameter = function.type.next;
  while parameter != null {
    tc_add_symbol(self, parameter.name, parameter.length, SYM_VARIABLE,
        arity - parameter.value, parameter.type, parameter.dim);
    parameter = parameter.next;
  }
  tc_parse_token(self, '{');
  tc_write_fn_insn(self, arity);
  let next_variable = arity + 1;
...
}
```

Finally, the 4 *panic* variables are removed from the main function, since they are no longer needed by the native `panic_result` function. The *flash_buffer* parameter is also removed, and a test is added to make sure that *dst_buffer* is *word aligned*, *i.e.*, is a multiple of 4. Indeed the generated code is not fully position independent: it can only be moved by $\pm 4n$ bytes, due to the ADR instruction. We assume here that it will be run at a word aligned address, which requires it to generated at such an address.

```
fn tc_main(src_buffer: &u32, dst_buffer: &u32) -> u32 {
...
  compiler.symbols = null;
  let panic3 = 0;
  let panic2 = 0;
  let panic1 = 0;
  let panic0 = 0;
  error = panic_result(&panic0);
...
  if (dst_buffer as u32) & 3 != 0 { panic(110); }
  tc_read_char(compiler);
...
}
```

20.4 Compilation and tests

To compile the above source code proceed as follows (see also Figure 16.4).

Edit v1 In the command editor, type "F3"+"r" and "F4"+"r" to load and edit the current compiler version. Then update it to the 1^{st} version of the native compiler. For convenience, we also provide this code in the `native_compiler_v1.txt` file in https://ebruneton.github.io/toypc/sources.zip. When you are done, exit the text editor and type "F5"+"r" to save your work. Alternatively, you can "cheat" by running the following command on an external computer (see Section 16.4 for more details):

```
user@host:~$ python3 flash_helper.py < part3/native_compiler_v1.txt
```

Compile v1 In the command editor, type "F6"+"r" to compile the code you typed. If all goes well, after about 4 seconds, you should get a result equal to 0 (meaning that no error was found). If this is not the case use Appendix D to get the error code meaning, fix this error, save the program and compile it again. Repeat this process until the compilation is successful. Then type "F7"+"r" to save the result.

Test v1 Type "F2"+"r" to create a new program, "F4"+"r" to edit it, and type the following small test program, which computes the factorial of 6:

```
fn factorial(n: u32) -> u32;
fn test() -> u32 { return factorial(6); }
fn factorial(n: u32) -> u32 {
  if n == 0 { return 1; }
  return factorial(n - 1) * n;
}
```

Compiling this program with the native compiler is easy, but running it is harder since its compiled code is using native instructions. One issue is that this code uses some registers also used by the bytecode interpreter. Another is that it returns the result in R0, but a command must return it on the stack. To solve these issues we need a small adapter subroutine using native code:

PUSH	R**0**..R**6** $LR \rightarrow$ stack	1011010*1***01111111**	B57F	000
LDR	R**0** $\leftarrow$ mem32[$\lfloor$PC$\rfloor_4 + 4 * 2$]	01001**000***00000010*	4802	002
BLX	PC $\leftarrow$ R**0** $- 1$, LR $\leftarrow a + 3$	010001111**0000**000	4780	004
LDR	R**1** $\leftarrow$ mem32[$\lfloor$PC$\rfloor_4 + 4 * 2$]	01001**001***00000010*	4902	006
STR	R**0** $\rightarrow$ mem32[R$1 + 4 * 0$]	01100**00000***001***000**	6008	008
POP	R**0**..R**6** $PC \leftarrow$ stack	1011110*1***01111111**	BD7F	00A
data	(program address)		00000000	00C
data	(result address)		00000000	010

```
00000000 00000000 BD7F6008 49024780 4802B57F   000
```

This subroutine saves the registers used by the bytecode interpreter (and the LR), loads the interworking address of the test program in R0, calls it, loads an address

where to store the result in R1, and stores it here. Finally, it restores the saved registers and returns. It is followed by two placeholders for the program and result addresses. To run it we can store it on the stack, as we did for the `page_flash` function. For this, type "F9"+"e" to edit this command and change its code to the following:

```
fn 0
  cst 537330176 cst 537379328 cst 803328 call 4612
  get 4 cst_0 ifne 53
  ptr 4 cst 537379333 cst 3179241480 cst 1224886144 cst 1208137087
  ptr 9 cst_1 add blx
  get 4 retv
d TEST_COMPILER
```

Then type Escape and "s" to save it. The 3^{rd} line of the function's body pushes each word of the subroutine code on the stack, in reverse order (*i.e.*, starting with the result address, and the test program interworking address $2007C200_{16}$+4+1). The next line computes the interworking address of this subroutine and calls it. Note that we run the test program directly in RAM, instead of at the *flash_buffer* address used to compile it (which is in fact ignored). This is possible because its native code is position-independent.

Finally, type "r" to run this command. If the result is not 720 = $2D0_{16}$ this means that the compiler is wrong. In this case, type "F8"+"r" to restore the types compiler. Then repeat the previous steps and double check everything until this test passes.

Edit v2 Type "F3"+"r" to load the 1^{st} version of the native compiler and "F4"+"r" to edit it. Then update it to the 2^{nd} version. For convenience, we also provide this code in the `native_compiler_v2.txt` file. Then save this new version with the F5 command. Alternatively, run the following command on an external computer:

```
user@host:~$ python3 flash_helper.py < part3/native_compiler_v2.txt
```

Compile v2 Type"F6"+"r" to compile this new code. The result should be 0, meaning "no error". If this is not the case, repeat the "Edit v2" and "Compile v2" steps until all errors are fixed.

Test v2 As in the previous chapter, the compiled code of the 2^{nd} version of the native compiler is not identical to that of the 1^{st} version. The obvious reason is that the former (produced by the native compiler) uses native instructions, while the latter (produced by the types compiler) uses bytecode instructions.

To test the 2^{nd} version we can check that it produces the same native code as the 1^{st} for the same input program, namely the 2^{nd} version of the native compiler itself. We already have the 2^{nd} version compiled by the 1^{st}. To get the 2^{nd} version compiled by itself we need a small adapter subroutine, as in the "Test v1" step:

PUSH	R**0**..R**6** $LR \rightarrow$ stack	1011010*1***01111111**	B57F	000
LDR	R**0** $\leftarrow$ mem32[$\lfloor$PC$\rfloor_4 + 4 * 4$]	01001**000***00000100*	4804	002
LDR	R**1** $\leftarrow$ mem32[$\lfloor$PC$\rfloor_4 + 4 * 4$]	01001**001***00000100*	4904	004
LDR	R**2** $\leftarrow$ mem32[$\lfloor$PC$\rfloor_4 + 4 * 2$]	01001**010***00000010*	4A02	006

BLX	$\mathrm{PC} \leftarrow \mathrm{R}\mathbf{2} - 1, \mathrm{LR} \leftarrow a + 3$	0100011111**0010**000	4790	008
LDR	$\mathrm{R}\mathbf{1} \leftarrow \mathrm{mem32}[\lfloor \mathrm{PC} \rfloor_4 + 4 * \mathit{4}]$	01001**001***00000100*	4904	00A
STR	$\mathrm{R}\mathbf{0} \rightarrow \mathrm{mem32}[\mathrm{R}\mathit{1} + 4 * 0]$	01100**00000***001***000**	6008	00C
POP	$\mathrm{R}\mathbf{0}..\mathrm{R}\mathbf{6}\ \mathit{PC} \leftarrow \mathrm{stack}$	1011110*1***01111111**	BD7F	00E
data	(program address)		00000000	010
data	(src buffer)		00000000	014
data	(dst buffer)		00000000	018
data	(result address)		00000000	01C

```
00000000 00000000 BD7F6008 49044790 4A024904 4804B57F   000
                                    00000000 00000000   018
```

This subroutine saves the registers used by the bytecode interpreter (and the LR). It then loads the native compiler's parameters in R0 and R1, and its interworking address in R2, calls this compiler, loads an address where to store its result in R1, and stores it here. Finally, it restores the saved registers and returns. It is followed by placeholders for the program address, its arguments, and a result address. To run it type "F9"+"e" to edit this command and change its code to the following:

```
fn 0
  cst_0
  ptr 4 cst 537391616 cst 537330176 cst 537379333
  cst 3179241480 cst 1225017232 cst 1241663748 cst 1208268159
  ptr 12 cst_1 add blx
  get 4 retv
d TEST_COMPILER
```

Then type Escape and "s" to save it. The first line of the function's body initializes the result value to 0. The next two lines push the subroutine data and instructions, in reverse order. We use as program address the interworking address of the 2^{nd} version of the native compiler ($2007C200_{16}$+4+1), as *src_buffer* its source code (20070200_{16}), and as *dst_buffer* a RAM region 12 KB after the program address ($2007F200_{16}$).

Finally, type "r" to run this command, *i.e.*, to compile the 2^{nd} version with itself. The result should be zero. You should also get it in a fraction of a second, instead of about 4 seconds in the "Compile v2" step. This is because native code is much faster than our bytecode interpreter. If the result is not 0 repeat the previous steps from "Edit v2" and double check everything.

To check if the generated code, at address $2007F200_{16}$, is identical to that produced with the 1^{st} version (at address $2007C200_{16}$) we need to update the "F10" command first (this command currently compares the buffers at addresses $C1200_{16}$ and $2007C200_{16}$). Type "F10"+"e" to edit it and change its second line as follows:

```
fn 0
  cst 537379328 cst 537391616
  get 4 load cst8 4 add
```

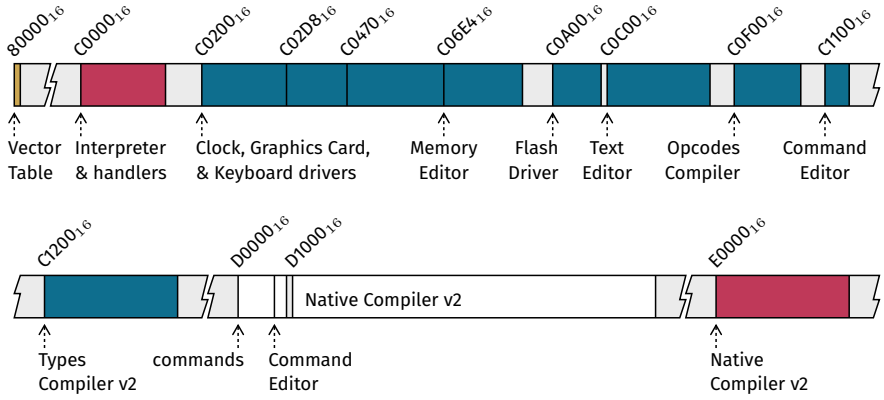


FIGURE 20.4 The flash memory content at the end of Part 3. White, blue, red, and gray areas represent source code, bytecode, native code, and unused memory, respectively (not to scale).

```
  cst_0
  get 7 get 6 ifge 50
  get 4 get 7 add call 960 get 5 get 7 add call 960 ifne 50
  cst_1 add goto 19
  get 6 get 7 sub retv
d COMPARE_COMPILED_CODE
```

Then type Escape and "s" to save it. Finally, type "r" to run it. The result should be 0, indicating that the two compiler versions produced the same compiled code. If this is not the case, repeat the steps from "Edit v2" until this test passes. It might also happen that the 1^{st} version is wrong despite the "Test v1" step. In this case type "F8"+"r" to restore the types compiler and restart from the "Edit v1" step.

In the next part we need both the types compiler and the native compiler, to produce either bytecode or native code. We currently have a backup of the former at address $E0000_{16}$, and two copies of the latter at addresses $2007C200_{16}$ and $2007F200_{16}$. However, the types compiler code is not position independent and cannot be used from address $E0000_{16}$. To copy it to its original address, restore its backup by typing "F8"+"r". We can then replace this backup, starting at page 512, with the native compiler. For this type "F8"+"e" to edit this command and replace it with the following:

```
fn 0
  cst 537379328 cst 512 call 2836
  cst_0 retv
d STORE_COMPILED_CODE
```

The type Escape and "r" to run it. At this stage the flash memory content is the one depicted in Figure 20.4.

Conclusion

A compiler transforms a program written in a well defined programming language into instructions for a real or virtual machine. For this it decomposes the program into a sequence of tokens, checks that these tokens follow the language's grammar, and finally generates instructions based on the program's grammatical structure. In this part we built a very basic compiler, for a very simple language, via successive improvements. Starting from a simple "text to binary converter" written in binary form[1], we progressively added support for labels, expressions, statements, and types.

The resulting language is much easier to use than machine code instructions, but can still be improved in many ways. For instance, we could introduce new `u8` and `u16` types to represent 8 and 16 bit numbers. Then the compiler could automatically use LDRB, STRB, LDRH, and STRH instructions to load and store values of these types in memory. This would be more efficient, and more practical for the user, than the current `load8`, `store8`, `load16`, and `store16` functions. As an another example, we could introduce a new type to represent a sequence of characters, called a *string*. This would be more practical than using a pointer to the first character, plus a variable containing the total number of characters.

Even without improving the language, the compiler itself can be improved in many ways, to produce smaller and more efficient code. For instance, an expression such as `(1 << 12) - 1` is currently compiled into instructions to load 1 and 12 into registers, to shift the former with the latter, to load 1 again in a register, and to subtract it from the previous result. Instead, the compiler could compute all this during compilation, and produce a single instruction to load the result (4095) in a register. This would give both smaller and more efficient code. As another example, already mentioned in the previous chapter, the compiler could make better use of the registers, to avoid many instructions copying values between registers and the memory.

Further readings

To learn how these improvements can be implemented, as well as many others, you can read one of the following books:

- "Compilers: Principles, Techniques, and Tools (2nd Edition)" [2]. This is a classic textbook about compilers for students in computer science. It covers all parts of a

[1] We used this starting point because the goal was to program a compiler from scratch. This is more or less how the very first assembler was written. Nowadays, a new compiler for a new language is written in an existing language, and compiled with an existing compiler (at least for its first version; it can then be rewritten in this new language and compiled with itself).

compiler (scanner, parser, type checking, code generation, code optimization, etc) and presents for each part the relevant theory and algorithms.
- "Engineering a Compiler" [6]. This book is similar but more recent. Some methods, such as the Static Single Assignment form, are presented in more details in this book than in [2]. Others, on the hand, are presented more extensively in [2] (such as interprocedural analysis).

In addition to these books, it is useful to have some basic knowledge about some generic algorithms and data structures, and their computational complexity (*i.e.*, how much time and space they need to run). For this "Introduction to Algorithms, Third Edition" [7] is a great book.

PART

4 A Toy Operating System

Introduction

We now have a toy programming language, hereafter called Toy, which makes it much easier to program our computer (compared to what it was at the end of Part 2). In particular, we no longer need to manually keep track of function addresses, instruction offsets, local variable indices, etc. However, we still need to manually keep track of the flash memory content with maps such as the one in Figure 20.4. This is necessary to find unused memory regions where new programs can be stored, without overriding existing programs or data. Likewise, we need to manually keep track of the RAM content with maps such as the one in Figure 15.3. This is necessary to find unused RAM regions where programs can store their data, without overriding the data of other programs. This manual work is quite tedious and error prone. The main goal of this part is thus to implement a program which can do it for us. Such a program is called an *operating system*.

In the previous part we removed the need of function addresses and local variable indices by using symbolic names for functions and local variables, and by letting the compiler maintain a map between the two. We can do the same for programs or other data stored in flash memory. More precisely, we can use *file names* to refer to pieces of data stored in flash memory, hereafter called *files*, and let the operating system maintain a map between file names and flash memory addresses. This map must obviously be stored in flash memory too, otherwise it would be lost after a reset. The precise way of storing it, as well as the files themselves, is called a *file system*.

In summary, the main goal of this part is to implement a toy operating system, based on a file system. Another goal is to eventually get rid of our bytecode interpreter, since we can now compile programs into native code, which is much faster than bytecode. This implies rewriting the drivers and programs from the previous parts, written directly in binary bytecode form, into Toy source code. We achieve these goals in seven steps, presented in as many chapters:

- Chapter 21 defines of a toy file system and provides corresponding functions to create, read, write, and delete files. We use them at the end to initialize a new, empty file system in the Flash0 memory bank (so far we only used the Flash1 bank – see Figures 6.2 and 6.3).
- Chapter 22 provides a new implementation, in Toy, of the clock, graphics card, and keyboard drivers from Part 2. We test them at the end with a small program which simply displays on the screen each key typed on the keyboard, as in Chapter 11. This time, however, we compile it in native code, and we store it in our file system. We also implement a *boot loader* to start it directly after reset, without going through the memory editor.

- Chapter 23 provides functions to start and stop programs stored in the file system, while keeping track of the RAM used by each running program, called *processes*. Together with the file system and driver functions, they constitute the first version of a program called the operating system *kernel*. This chapter also provides a way for processes to use services provided by the kernel.
- Chapter 24 extends the kernel with new services providing a simple, unified, and safe way to use the computer resources. These include the keyboard, the graphics card, and the files. All these resources are used via byte sequences called *streams*.
- Chapter 25 provides a better and easier to use version of the command editor, called a *command-line interpreter*, or *shell*. This program is automatically started by the kernel after a reset. Its role is to start other programs, with commands typed by the user. This chapter also provides a new implementation of the text editor from Chapter 14, in Toy. Together with the Toy compiler and the shell, stored in the file system, this gives an autonomous, *self-hosted* operating system. This means that we can edit and recompile its entire source code with itself, without needing the bytecode interpreter, the memory editor, or any other program written in the previous parts.
- Chapter 26 illustrates this self-hosting property by using the shell, the text editor, and the compiler, running with the operating system, to replace its kernel with a new version. This new version uses the Memory Protection Unit from the microcontroller to protect the kernel and each program from bugs in other programs.
- Chapter 27 completes our operating system with a few small utility programs, in particular to list, copy, and delete files. It also provides a better shell version.

Finally, just for fun, we conclude this book with a small game implemented with our toy computer, in Chapter 28.

CHAPTER

21 File System

As explained in introduction, the first step to build our toy operating system is to implement a file system. The goal is to remove the need to manually keep track of the flash memory content. This chapter defines the requirements for this file system, presents the data structures used to achieve them, and provides some functions based on these data structures to create, read, write, and delete files. We test them at the end with a small program and a test file in a previously initialized file system.

21.1 Requirements

The basic requirements of our file system are the ability to create an empty file with a given name, to write data in an existing file, to read data from a file, and to delete a file. We also require the possibility to get the size of a file, and to get a list of all the existing files.

Flash memory is larger than the RAM, and it should be possible to use files which fit in flash but not in RAM. For this we require the possibility to read only a small piece of a file, and to write data in a file piece by piece. More precisely, it should be possible to append new data at the end of an existing file. To simplify, we do not require the possibility to insert new data in the middle of a file, nor even to partially override existing data.

Standard file systems generally support grouping files into a hierarchy of folders. To simplify, and since we expect to have less than a hundred files in our final system, we require a *flat* file system (*i.e.*, non hierarchical). A consequence is that each file must have a globally unique name. For instance, we cannot have two files named `help.txt`. Instead, we need to use names such as `text_editor_help.txt` and `compiler_help.txt`. For this we require the possibility to use "long" file names, up to 128 characters at least. However, to simplify the implementation:

- file names should only contain printable ASCII characters other than Space.
- file names cannot be changed. To rename a file from x to y, one must thus create a new file y, copy x's data into y, and finally delete x.

Standard file systems also generally keep track of the creation and modification dates of each file, and of which users can read, write or execute it. Here again, to simplify, we do not require any of these features.

FIGURE 21.1 A file system with 5 files stored as contiguous sequences of bytes, one after the other (top). Adding new bytes to C requires moving it to a new address (bottom). Gray areas represent unused memory.

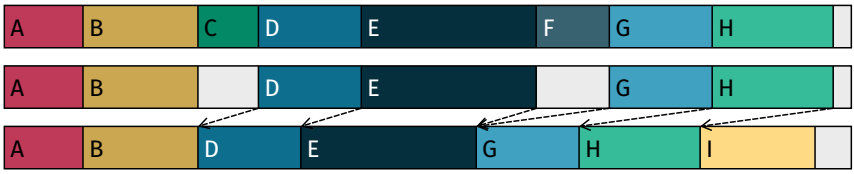


FIGURE 21.2 When files are stored as contiguous sequences of bytes, next to each other (top), and some files are deleted (C, F, middle), the unused memory (gray) can be too fragmented to store a new file (I). In this case files must be moved to merge these regions (bottom).

21.2 Data structures

So far each program that we saved in flash memory was stored as a contiguous sequence of bytes. This is the easiest method when one has to manually store data while keeping track of the memory content. However, this method has several drawbacks:

- when programs are stored compactly, one after the other, a program cannot be updated to a larger version without moving it elsewhere (see Figure 21.1).
- if some unused memory is left between each program, to mitigate the above problem, then this memory is wasted and reduces the total storage capacity.
- even if programs are never updated, and no space is left between them, after some programs are deleted we can get a lot of small unused memory regions between them. This can prevent the storage of a new large program, unless the existing ones are moved next to each other to merge these small regions (see Figure 21.2).

To avoid these issues, most file systems divide the storage into many small blocks of equal capacity, split each file into "chunks", and store each chunk in a block (see Figure 21.3). With this method, contiguous chunks of a file do not need to be stored in contiguous blocks, nor even in blocks in increasing address order. This solves the above problems. Indeed:

- as long as there are enough *free blocks* (*i.e.*, not used by any files), files can be extended, and new files can be created, without having to move existing data.
- no memory is wasted between files. The only wasted memory is in the last block of each file whose size is not an exact multiple of the block size. For instance, if each block contains 200 bytes, a 1250 bytes file uses 6 full blocks and 50 bytes in a

FIGURE 21.3 The same files as in Figure 21.1, stored in a block based file system (top). Each file is divided in chunks which can be stored in any block, in any order (*e.g.*, A is divided in 3 chunks A_0, A_1 and A_2). Bottom: adding bytes to a file can be done by putting them in any free block (in gray), without moving existing data (*e.g.*, chunks C_2 and C_3 are added in free blocks on the right).

seventh block, leaving 150 unused bytes (these bytes are not used to store data from another file, to avoid re-introducing the above issues). On average, we thus get only one half block of wasted memory per file.

However, this method creates a new problem. Indeed, while an address and a size were sufficient to keep track of the location of a file (with the first method), we now need a list of the blocks containing its data. This is a bit more complex, but the pros outweigh the cons. We therefore use blocks for our toy file system.

21.2.1 File blocks

Block capacity

Our file system is designed to store files in flash memory, which is divided in pages of 256 bytes each. Moreover, writing into flash memory can only be done one full page at a time. The most convenient way to build our file system is thus to use blocks whose capacity is a multiple of 256 bytes. To minimize the wasted memory per file (one half block on average), we use blocks of exactly 256 bytes.

Block structure

In order to know which blocks contain the data of a file, and in which order, we use a linked list *stored in the blocks themselves*. For this we divide each block in two parts called *header* and *payload*, used as follows (see Figure 21.4, where these parts are shown in white and blue, respectively):

- in each block of a file, except the last one, the first header word stores the address of the block containing the next file chunk, noted `next_block`. The payload contains a $256 - h$ bytes chunk of the file, where h is the header size.
- in the last block the first header word contains the *block size* s, defined as the header size plus the payload size. The payload contains the last file chunk ($s - h$ bytes by definition). The remaining $256 - s$ bytes are unused.

Note that block addresses are larger than or equal to 80000_{16}, the start address of the flash memory (see Figure 6.3). On the other hand, the size s of a block is less than or equal to 256. Hence, the value v of the first word of a block unambiguously indicates if this block has a next block ($v > 256$) or is the last one ($v \leq 256$).

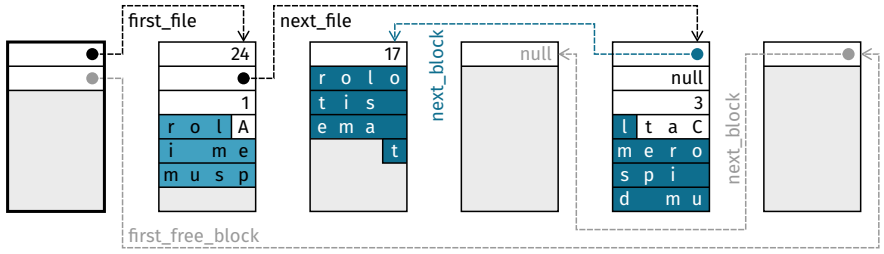


FIGURE 21.4 The data structures of our file system, illustrated with 6 blocks of 7 words each. The superblock (left) points to the first file and the first free block. The first block of each file starts with a pointer to the next block (blue arrows), a pointer to the next file (black arrows), the length of its name, and the name itself (white background). Here there are 2 files, "A" containing "lorem ipsum" in one block (light blue), and "Cat" containing "lorem ipsum dolor sit amet" in two blocks (blue). The last block of each file starts with the total number of bytes used in this block (here 24 and 17). Each free block points to the next one (gray arrows).

File list

In order to maintain the map between file names and flash memory addresses, which is the main goal of our file system, we use another linked list, also stored in the blocks themselves. More precisely, we use the header of the first block of each file to store the following data (after the `next_block` address, see Figure 21.4):

- the address of the next file's first block, noted `next_file`.
- the length of the file name, noted `name_length`.
- the `name_length` bytes of the name itself.

With this data structure, finding the address of the first block of a file can be done by following the `next_file` links from one file to the next, until the block's name matches the one we are looking for. The other blocks of the file can then be found by following the `next_block` links.

By definition, the first block of a file has a header size h equal to $12 +$ `name_length`: 4 bytes each for `next_block`, `next_file`, and `name_length`, and `name_length` bytes for the file name. The header of the other file blocks only contain a `next_block` link and thus has size $h = 4$.

Note that we implicitly assumed above that file names fit in the first file block. To ensure this we restrict them to $256 - 12 = 244$ characters at most.

21.2.2 Free blocks

In order to create new files, or to append data to an existing file, we need to find blocks not yet used by any file, called free blocks. For this we use a linked list of all the free blocks, once again stored in the free blocks themselves. More precisely, we store in

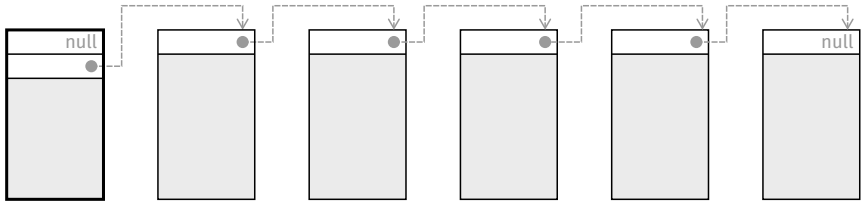


FIGURE 21.5 The initial state of the file system. All blocks are free and simply point to the next block in increasing address order. The super block's `first_file` pointer is `null` because there are initially no files.

the first word of each free block the address of the next free block (or 0 if this is the last free block – see Figure 21.4).

21.2.3 Super block

In order to use the above linked list of files we need to know where it starts. More precisely, we need the address of the first block of the first file in this list. Similarly, for the same reason, we need the address of the first free block in the list of free blocks. We store these two addresses, noted `first_file` and `first_free_block` respectively, in a special block called the *super block* (see Figure 21.4). This block only contains these two addresses, and is stored at a fixed, predefined address.

21.2.4 Initial state

Initially each bit of the flash memory is equal to 1. The first word of each page is thus initially equal to FFFFFFFF_{16}, which is not a valid `next_block` or `first_file` address. In other words, the flash memory does not initially contain a valid file system data structure. To solve this we need to explicitly initialize it to a valid state. For this the simplest is to use a state without any file, where all blocks are free (see Figure 21.5). This initialization process is called *disk formatting*.

21.3 Implementation

We can now implement some functions to create, read, write and delete files stored as described above. These functions have several goals. In this chapter we use some of them to format the Flash0 memory bank, hereafter called the "disk". We also test them by creating a small file, writing data in it, reading it back, and deleting the file. In the next chapters we use these functions to copy some programs edited and compiled in the Flash1 memory bank as files on the disk. Finally, we also copy them in our operating system kernel to implement a *disk driver* (in addition to the clock, graphics card and keyboard drivers).

In order to use the above functions we need to put them in some program. We thus

start a new program here. As in our compiler, we start by declaring a `main` function, implemented at the very end, but called from the first function, here called `entry` (both return 0 if and only if no error occurred):

```
fn main() -> u32;
fn entry() -> u32 {
  const NVIC_INTERRUPT_SET_ENABLE_REGISTER: &u32 = 3758153984;
  const NVIC_INTERRUPT_CLEAR_ENABLE_REGISTER: &u32 = 3758154112;
  const USART_ID: u32 = 131072;
  *NVIC_INTERRUPT_CLEAR_ENABLE_REGISTER = USART_ID;
  let result = main();
  *NVIC_INTERRUPT_SET_ENABLE_REGISTER = USART_ID;
  return result;
}
```

Since this program will write data in the Flash0 memory bank, which cannot be read while writing in it (see Section 6.5.1), the `entry` function temporarily disables USART interrupts before calling `main` (see Section 11.3 and Table 11.2). Otherwise, a key press during this time would read the Vector Table to find the USART handler address, which would trigger a Hard Fault since our Vector Table is at address 80000_{16} in the Flash0 memory bank (see Section 9.6)[1].

We continue with two small functions to load and store bytes, copied from the types compiler, which are needed later on:

```
fn load8(ptr: &u32) -> u32 { return (*ptr) & 255; }
fn store8(ptr: &u32, value: u32) { *ptr = (*ptr) & 4294967040 | value; }
```

21.3.1 Data structures

The following types correspond to the data structures defined in the previous section:

```
struct DiskBlock {
  next_block: &DiskBlock
}
struct FileBlock {
  next_block: &DiskBlock,
  next_file: &FileBlock,
  name_length: u32,
  name: u32
}
struct SuperBlock {
  first_file: &FileBlock,
  first_free_block: &DiskBlock
}
```

A `DiskBlock` represents either a free block or a file block other than a first file block. In the first case `next_block` is either `null` or points to the next free block. In

[1] Another solution is to move the Vector Table in RAM. We do this in the next chapter.

the second case `next_block` points to the next file block, if its value is larger than 256. Otherwise it contains the block size (header size plus payload size). The following functions make it easier to work with this rule:

```
fn block_next(self: &DiskBlock) -> &DiskBlock {
  if self.next_block as u32 <= 256 { return null; }
  return self.next_block;
}
fn block_size(self: &DiskBlock) -> &u32 {
  return &self.next_block as &u32;
}
```

The first one returns the next block of a block, or `null` if there is none. The second one returns a pointer to the size of a block, assuming that it does not have a next block.

A `FileBlock` represents the first block of a file. Its `name` field contains the first 4 characters of the file name (at most). If `file` is a `FileBlock` pointer, then the file name's i^{th} character is at address `&file.name` $+ i$.

The `SuperBlock` points to the first file and the first free block. We store it in page 1 of the Flash0 memory bank, at address 80100_{16} = 524544 (page 0 must contain a valid Vector Table in order to boot from flash memory – cf. Section 7.4):

```
const SUPER_BLOCK: &SuperBlock = 524544;
```

21.3.2 Reading functions

We can now implement some functions using these block data structures. We start in this section with functions which only read them, because they are simpler than those modifying blocks.

We first need a function to find the `FileBlock` of a file, given its name. Here we assume that the linked list of files is sorted in alphabetical order, because it is more convenient for users than an unsorted list. We can thus find a file with a given name n by comparing it with the name of each file in this list, until a match is found or the file name is greater than n (by hypothesis the next file names are greater than n too). This requires the ability to compare two names. The following function does this:

```
const EQUAL: u32 = 0;
const SMALLER: u32 = 1;
const GREATER: u32 = 2;
fn disk_compare_file_name(file: &FileBlock, name: &u32, length: u32) -> u32 {
  let file_name_length = file.name_length;
  let file_name = &file.name;
  let i = 0;
  while i < file_name_length && i < length {
    if load8(file_name + i) < load8(name + i) { return SMALLER; }
    if load8(file_name + i) > load8(name + i) { return GREATER; }
    i = i + 1;
```

```
  }
  if file_name_length < length { return SMALLER; }
  if file_name_length > length { return GREATER; }
  return EQUAL;
}
```

It returns `EQUAL`, `SMALLER`, or `GREATER` if the name of `file` is equal, smaller, or greater than `name`, respectively (more precisely than the name starting at address `name` and with `length` characters). Its `while` loop compares the characters of these names from left to right, and stops when a difference is found or the end of one name is reached. If no difference is found names are compared based on their length.

The following function uses it to find a file, with the algorithm described above:

```
fn disk_find_file(name: &u32, length: u32, previous_file: &&FileBlock) -> ⸨
                                                                    ⸩&FileBlock {
  let file = SUPER_BLOCK.first_file;
  let file_name = SMALLER;
  while file != null {
    file_name = disk_compare_file_name(file, name, length);
    if file_name == EQUAL { return file; }
    if file_name == GREATER { return null; }
    if previous_file != null { *previous_file = file; }
    file = file.next_file;
  }
  return null;
}
```

If returns the file's `FileBlock`, or `null` if no file was found. If `previous_file` is not `null` it also stores at this address a pointer to the `FileBlock` immediately preceding the returned file in the linked list of files (we assume that `*previous_file` is initialized to `null` by the caller).

Once we have a `FileBlock` we can use it to compute the size of this file, as follows:

```
fn disk_get_file_size(file: &FileBlock) -> u32 {
  let file_size = 0;
  let block = file as &DiskBlock;
  let header_size = 12 + file.name_length;
  loop {
    if block_next(block) == null {
      return file_size + *block_size(block) - header_size;
    }
    file_size = file_size + 256 - header_size;
    block = block.next_block;
    header_size = 4;
  }
}
```

This function simply adds the payload size of each block of the file, found by following the `next_block` links. As described in Section 21.2.1, the payload size is

$256 - h$ for all blocks except the last one, where the header size h is 12 plus the file name length for the first block, and 4 otherwise. The payload size of the last block is the block size minus h.

To read the content of a file into some buffer we need to copy memory. We already have a `mem_copy` function in our flash memory driver (see Section 13.2), but only in bytecode form. Since we want to eventually get rid of the bytecode interpreter we re-implement it here. Actually we only need it to copy data between memory regions which do not overlap. We thus implement a simpler version instead:

```
fn mem_copy_non_overlapping(src: &u32, dst: &u32, size: u32) {
  let i = 0;
  while i < size {
    store8(dst + i, load8(src + i));
    i = i + 1;
  }
}
```

We use it to implement the following function, which copies up to `size` bytes, starting `*offset` bytes from the file block `*block`, into `dst`:

```
fn disk_read_file(block: &&DiskBlock, offset: &u32, dst: &u32, size: u32) ≀
                                                                  ≀-> u32 {
  let available = 0;
  loop {
    if block_next(*block) == null {
      available = *block_size(*block) - *offset;
    } else {
      available = 256 - *offset;
    }
    if available >= size {
      mem_copy_non_overlapping((*block) as &u32 + *offset, dst, size);
      *offset = *offset + size;
      return 0;
    }
    mem_copy_non_overlapping((*block) as &u32 + *offset, dst, available);
    if block_next(*block) == null {
      return size - available;
    }
    *block = (*block).next_block;
    *offset = 4;
    dst = dst + available;
    size = size - available;
  }
}
```

This function first computes the number of available bytes in `*block`, after `*offset`, which depends on whether it is the last block or not. If there are more bytes available than requested, it copies `size` bytes from this block into `dst` and returns 0, meaning that all the requested bytes have been copied. Otherwise it copies all the

available bytes into `dst`, and then checks if there is a next block. If not it returns the number of bytes that were requested but could not be copied because the end of the file was reached. If there is a next block it repeats the above steps after updating `*block` to this next block (whose header size is necessarily 4), and after updating the destination address and the number of bytes to copy (since `available` bytes have just been copied).

Note that if this function returns 0, meaning that the file may have more bytes that those just read, then `*block` and `*offset` contain the correct values to read these additional bytes with a new call to this function.

21.3.3 Writing functions

This section provides functions to create a file, append data to a file, and delete a file. This requires the ability to modify blocks, *i.e.*, to modify data in flash memory pages. We therefore start with some low level functions to do this.

Low level functions

As explained in Section 6.5.1, writing a flash memory page requires storing a value in the 64 words of this page first, even if we only need to change a single word. Hopefully the value of these words can be overridden several times before actually writing the page. This suggests the following strategy to change some words in a page:

1. load the current value of each page word and store it back unchanged.
2. store new value(s) in the desired page word(s).
3. send a "write page" command to flash the page.

The following functions implement steps 1 and 3 of this process (`block_write` works as explained in Section 6.5.1; `address` must be in $[80000_{16}, C0000_{16}[$ = [524288,786432[):

```
fn block_read(ptr: &u32) {
  let end = ptr + 256;
  while ptr < end {
    *ptr = *ptr;
    ptr = ptr + 4;
  }
}
fn block_write(address: u32) {
  const EEFC0_COMMAND_REGISTER: &u32 = 1074661892;
  const EEFC0_STATUS_REGISTER: &u32 = 1074661896;
  const WRITE_PAGE_COMMAND: u32 = 1509949443;
  let page = (address - 524288) >> 8;
  *EEFC0_COMMAND_REGISTER = WRITE_PAGE_COMMAND | (page << 8);
  while *EEFC0_STATUS_REGISTER & 1 != 1 {}
}
```

Step 2 depends on each use case, but the important point is that only 32-bit values can be stored in the page, and only at addresses which are a multiple of 4 (see Section 6.5.1). In particular, the native STRH and STRB instructions cannot be used here. The `store8` function does not use them but cannot be used either, because loads do no "see" the effect of stores until the page is flashed (see Section 6.5.1). If it was called several times to store the bytes of a word, only the last call would have any effect.

Unfortunately, the payload of a block does not always start and end at a word-aligned address, as shown in Figure 21.4. We thus sometimes need to store individual bytes in a block. The following function solves this issue by using an intermediate `buffer` word to copy `size` bytes from `src` to `dst`:

```
fn block_copy_bytes(src: &u32, dst: &u32, size: u32) {
  let buffer_size = ((dst as u32) & 3) << 3;
  dst = dst - (buffer_size >> 3);
  let buffer = *dst & ((1 << buffer_size) - 1);
  let i = 0;
  while i < size {
    if buffer_size == 32 {
      *dst = buffer;
      dst = dst + 4;
      buffer = 0;
      buffer_size = 0;
    }
    buffer = buffer | (load8(src + i) << buffer_size);
    buffer_size = buffer_size + 8;
    i = i + 1;
  }
  if buffer_size != 0 { *dst = buffer; }
}
```

The `while` loop copies each source byte in `buffer`, one by one. When this buffer is full (*i.e.*, when `buffer_size` is equal to 32 bits), it is stored at `dst` and cleared. Otherwise each byte is copied in `buffer` after the `buffer_size` bits already used, and the buffer size is incremented by 8. If the buffer is not empty when the `while` loop ends, it is stored at `dst` too.

The beginning of the function is the most complex part. It initializes the buffer and rounds `dst` down to a multiple of 4, as required by the rest of the function. More precisely, if `dst` is equal to $4k + r$ with $0 \leq r < 4$ then this part rounds `dst` down to $4k$ and initializes `buffer` with the first r bytes of the word at $4k$:

- the 1^{st} line computes r with `(dst as u32) & 3`, and `buffer_size` as $8r = r \ll 3$.
- the 2^{nd} line rounds `dst` down to $4k$ (`buffer_size >> 3` = `buffer_size / 8`).
- the 3^{rd} line copies the least significant `buffer_size` bits of the word at `dst` into `buffer` ($1 \ll n = 2^n$ and $2^n - 1 =$ a number with n ones in base 2).

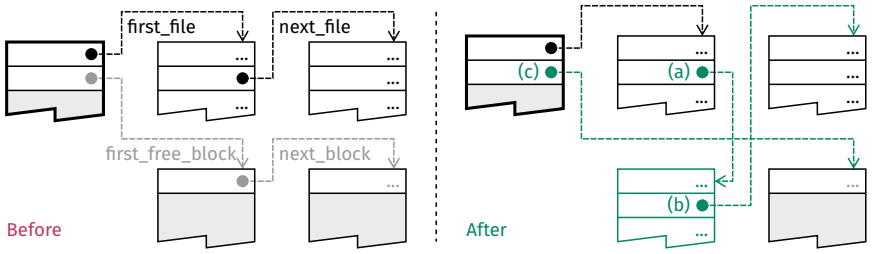


FIGURE 21.6 To create a file the first free block (left) is turned into a `FileBlock` (right, in green), removed from the list of free blocks (c), and inserted in the alphabetically sorted list of files (a, b).

Create file function

In order to create an empty file we need to find a free block, turn it into a properly initialized `FileBlock`, insert it in the linked list of files, and remove it from the linked list of free blocks (see Figure 21.6). If the file name is invalid, if a file with this name already exists, or if there is no free block the creation fails. We define some error codes for these cases:

```
const OK: u32 = 0;
const INVALID_ARGUMENT: u32 = 1;
const ALREADY_EXISTS: u32 = 2;
const OUT_OF_MEMORY: u32 = 3;
```

and we start a function to create a file by returning them if one of these cases happens:

```
fn disk_create_file(name: &u32, length: u32, result: &&FileBlock) -> u32 {
  if length == 0 || length > 244 { return INVALID_ARGUMENT; }
  let i = 0;
  while i < length {
    if load8(name + i) <= 32 || load8(name + i) >= 127 {
      return INVALID_ARGUMENT;
    }
    i = i + 1;
  }
  let super_block = SUPER_BLOCK;
  let previous_file: &FileBlock = null;
  let file = super_block.first_file;
  let file_name = SMALLER;
  while file != null {
    file_name = disk_compare_file_name(file, name, length);
    if file_name == EQUAL { return ALREADY_EXISTS; }
    if file_name == GREATER { break; }
    previous_file = file;
    file = file.next_file;
  }
```

```
  let new_file = super_block.first_free_block as &FileBlock;
  if new_file == null { return OUT_OF_MEMORY; }
```

At this point we have a block for the new file, in `new_file`, and we know that it must be inserted between `previous_file` and `file` to keep the list in alphabetic order. We start by updating the `next_file` link of `previous_file` (link (a) in Figure 21.6 – if `previous_file` is `null` we update the `first_file` link instead):

```
  let new_first_file = super_block.first_file;
  if previous_file != null {
    block_read(previous_file as &u32);
    previous_file.next_file = new_file;
    block_write(previous_file as u32);
  } else {
    new_first_file = new_file;
  }
```

We continue by initializing the new file block, after saving in a temporary variable the address of the second free block, which will become the new first free block:

```
  let new_first_free_block = new_file.next_block;
  block_read(new_file as &u32);
  *block_size(new_file as &DiskBlock) = 12 + length;
  new_file.next_file = file;
  new_file.name_length = length;
  block_copy_bytes(name, &new_file.name, length);
  block_write(new_file as u32);
```

Finally, we update the super block and store the address of the new file in `*result`:

```
  block_read(super_block as &u32);
  super_block.first_file = new_first_file;
  super_block.first_free_block = new_first_free_block;
  block_write(super_block as u32);
  *result = new_file;
  return OK;
}
```

Write file function

The next function appends `size` bytes from `src` to the file whose last block is `*block`. It returns `OK` if the operation succeeds, or `OUT_OF_MEMORY` if there are not enough free blocks for this:

```
fn disk_write_file(block: &&DiskBlock, src: &u32, size: u32) -> u32 {
  let super_block = SUPER_BLOCK;
  let new_first_free_block = super_block.first_free_block;
  let used = *block_size(*block);
  let free = 256 - used;
  let result = OK;
```

```
  loop {
    if size <= free {
      block_read((*block) as &u32);
      *block_size(*block) = used + size;
      block_copy_bytes(src, (*block) as &u32 + used, size);
      block_write((*block) as u32);
      break;
    }
    if new_first_free_block == null {
      result = OUT_OF_MEMORY;
      break;
    }
    block_read((*block) as &u32);
    (*block).next_block = new_first_free_block;
    block_copy_bytes(src, (*block) as &u32 + used, free);
    block_write((*block) as u32);
    *block = new_first_free_block;
    new_first_free_block = new_first_free_block.next_block;
    src = src + free;
    size = size - free;
    used = 4;
    free = 252;
  }
  if new_first_free_block != super_block.first_free_block {
    block_read(super_block as &u32);
    super_block.first_free_block = new_first_free_block;
    block_write(super_block as u32);
  }
  return result;
}
```

It first computes the number of bytes already `used` in `*block`, and the number of `free` bytes to store new data. If the number of bytes to copy is less than `free` then it appends them to `*block`, updates its header to reflect its new payload size, and returns `OK`. Otherwise, at least one free block b is necessary to store the bytes which cannot be stored in `*block`. If there is none then it returns `OUT_OF_MEMORY`. Otherwise it appends as many bytes as possible to `*block` (*i.e.*, `free` bytes), and uses b for the `next_block` of `*block`. It then repeats the above steps with this next block (which has 4 used bytes and 252 free bytes), after updating the source address and the number of bytes to write (since `free` bytes have just been written).

The function always ends by updating the super block, if some free blocks were used. Note that if it returns `OK`, then `*block` contains the new last block of the file, *i.e.*, the correct value to append more bytes to this file (with a new call to this function).

Clear and delete file functions

The last disk functions delete a file, or clear all its data without deleting it (yielding an empty file). Both need a way to recycle some file blocks into free blocks. The

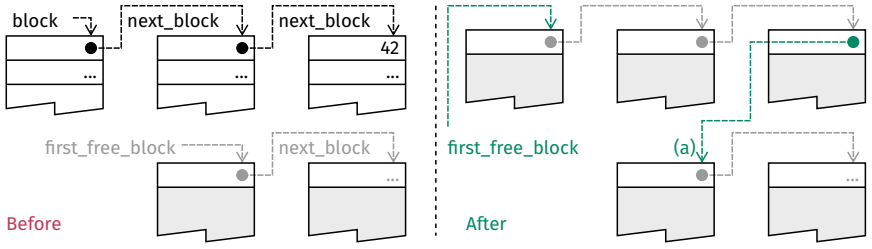


FIGURE 21.7 To delete the list of blocks starting at `block` (left) we just need to link the last one to the first free block (a), and to update `first_free_block` to `block` (right).

following function does this. It adds all the file blocks starting at `block` (included) to the list of free blocks, and returns the new first free block (see Figure 21.7):

```
fn disk_delete_file_blocks(block: &DiskBlock) -> &DiskBlock {
  let last_block = block;
  while block_next(last_block) != null {
    last_block = last_block.next_block;
  }
  block_read(last_block as &u32);
  last_block.next_block = SUPER_BLOCK.first_free_block;
  block_write(last_block as u32);
  return block;
}
```

For this it finds the last block of the list starting at `block`, and sets its `next_block` to the current first free block. The first free block then becomes `block`, but the super block is *not* updated to reflect this (the caller must do this instead).

With this function it is easy to clear a file: we just need to delete all its block except the first one (if there is more than one) and to update the super block accordingly. In any case we also need to reset the payload size of the first block to 0:

```
fn disk_clear_file(file: &FileBlock) {
  let super_block = SUPER_BLOCK;
  let new_first_free_block: &DiskBlock = null;
  if block_next(file as &DiskBlock) != null {
    new_first_free_block = disk_delete_file_blocks(file.next_block);
    block_read(super_block as &u32);
    super_block.first_free_block = new_first_free_block;
    block_write(super_block as u32);
  }
  block_read(file as &u32);
  *block_size(file as &DiskBlock) = 12 + file.name_length;
  block_write(file as u32);
}
```

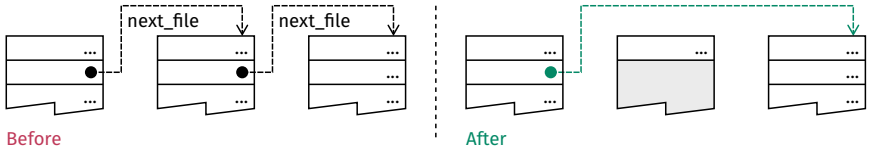


FIGURE 21.8 To remove a file from the list of files (middle left) we just need to change the `next_file` link of the previous file to the next file (right, green arrow).

To completely delete a file we need to delete all its blocks, including the first one, and to remove it from the list of files. For this we need to set the `next_file` link of the previous file (given as parameter) to the next file of `file` (see Figure 21.8 – if there is no previous file then the super block's `first_file` link must be updated instead):

```
fn disk_delete_file(file: &FileBlock, previous_file: &FileBlock) {
  let next_file = file.next_file;
  let super_block = SUPER_BLOCK;
  let new_first_file = super_block.first_file;
  let new_first_free_block = disk_delete_file_blocks(file as &DiskBlock);
  if previous_file != null {
    block_read(previous_file as &u32);
    previous_file.next_file = next_file;
    block_write(previous_file as u32);
  } else {
    new_first_file = next_file;
  }
  block_read(super_block as &u32);
  super_block.first_file = new_first_file;
  super_block.first_free_block = new_first_free_block;
  block_write(super_block as u32);
}
```

21.3.4 Disk formatting

In order to use the above functions we first need to format the disk, as illustrated in Figure 21.5. For this we finally implement the `main` function as follows:

```
fn gpu_draw_char(c: u32) = 1125;
fn main() -> u32 {
  let super_block = SUPER_BLOCK;
  let free_block = (super_block + 256) as &DiskBlock;
  let last_free_block = (super_block + 1022 * 256) as &DiskBlock;
  block_read(super_block as &u32);
  super_block.first_file = null;
  super_block.first_free_block = free_block;
  block_write(super_block as u32);
  while free_block <= last_free_block {
    block_read(free_block as &u32);
```

```
    if free_block == last_free_block {
      free_block.next_block = null;
    } else {
      free_block.next_block = free_block + 256;
    }
    block_write(free_block as u32);
    free_block = free_block + 256;
    gpu_draw_char('.');
  }
  return 0;
}
```

After the super block initialization, the `while` loop initializes 1022 free blocks by setting their `next_block` link to the next block, or to `null` for the last one (the disk contains 1024 blocks but the first one is reserved for the Vector Table, and the second one is the super block). Since writing a page takes a few milliseconds this process takes some time. To monitor its progress we draw a dot on the screen for each free block, with the gpu_draw_char function.

21.4 Compilation and tests

To type this program we first need to decide where to save it. Indeed, we don't have a file system yet. We thus still need to manually keep track of the flash memory content. We choose to store the source code at address $F0000_{16}$ = 983040 = page 768, and the compiled code at address $C3200_{16}$ = 799232 = page 50 (8 KB after the start of the types compiler, which is about 6 KB – see Figure 20.4).

We then need to update the command editor commands to use these new addresses. In the memory editor, type "w000c1172"+Enter, followed by "r", to start the command editor. Then type "F3" and "e" to edit the "load source code" command and update its code to the following (we call our new program the "builder" since its purpose is to build our operating system; see also Section 15.4.4):

```
fn 0
  cst 983040 cst 537330176 call 2708
  cst_0 retv
d LOAD_BUILDER_SOURCE_CODE
```

When done, type Escape and "s" to save it. In the same way, update the "save source code" command (F5) to:

```
fn 0
  cst 537330176 cst 768 call 2836
  cst_0 retv
d SAVE_BUILDER_SOURCE_CODE
```

the "compile source code" command (F6) to:

```
fn 0
  cst 983040 cst 537379328 cst 799232 call 4612 retv
d COMPILE_BUILDER_SOURCE_CODE
```

and the "store compiled code" command (F7) to:

```
fn 0
  cst 537379328 cst 50 call 2836
  cst 799236 calld retv
d STORE_AND_RUN_COMPILED_BUILDER_CODE
```

The first line stores the compiled code of the builder in flash memory. The second line runs it and returns its result (the `entry` function starts after the 4 bytes header of the compiled code data buffer).

21.4.1 Disk formatting

We now have everything we need to type our program, compile it and run it. Type "F2"+"r" to initialize a new text buffer, and "F4"+"r" to edit it. Then type the source code listed in the previous section. For reference, we also provide this code in the `disk_formatter.txt` file in https://ebruneton.github.io/toypc/sources.zip. When done, type "F5"+"r" to save it, and "F6"+"r" to compile it. The result should be 0, meaning that the compilation was successful. If not, type "F4"+"r" to fix the error. The text editor should open right at the error location. Fix the error indicated by the error code returned by the compiler (see Appendix D), save the program and compile it again. Repeat this process until the compilation is successful.

Finally, type "F7"+"r" to run it, *i.e.*, to format the disk. You should see dots appearing on the screen, eventually replaced with 0, meaning that the formatting is done. To check this you can type "Escape" to exit the command editor and then type "w00080100"+Enter, "w00080200"+Enter, "w00080300"+Enter, etc to have a look at the super block and the first few free blocks. The super block should start with 0 and 80200_{16}, and each free block should start with the address of the next one.

21.4.2 Tests

The above steps have only tested that the low level block reading and writing functions actually work. To test that the other functions work too, we can try to create a file in our new file system, append data to it, read it back, and finally delete the file. This can be done with the following code, which starts by creating a file, and returns 1 if the creation fails:

```
static NAME = ['m','y','_','f','i','l','e'];
const NAME_LENGTH: u32 = 7;

fn main() -> u32 {
  let file: &FileBlock = null;
  if disk_create_file(NAME, NAME_LENGTH, &file) != OK { return 1; }
```

It then initializes 300 bytes in RAM to some known values, starting at address 20080000_{16} = 537395200, and appends them to the file in two steps. If any step fails it returns a non-zero value:

```
  let block = file as &DiskBlock;
  let src = 537395200 as &u32;
  const SIZE: u32 = 300;
  let i = 0;
  while i < SIZE {
    store8(src + i, i);
    i = i + 1;
  }
  if disk_write_file(&block, src, 150) != OK { return 2; }
  if disk_write_file(&block, src + 150, 150) != OK { return 3; }
```

The following code checks that the file has the expected size, and reads its data back in RAM, starting 300 bytes after `src`, in chunks of 64 bytes:

```
  if disk_get_file_size(file) != SIZE { return 4; }
  block = file as &DiskBlock;
  let offset = 12 + NAME_LENGTH;
  let dst = src + SIZE;
  while disk_read_file(&block, &offset, dst, 64) == 0 {
    dst = dst + 64;
  }
```

It then tests that the data that was read is equal to the data that was written, and returns a non-zero result if this is not the case:

```
  i = 0;
  while i < SIZE {
    if *(src + SIZE + i) != *(src + i) {
      return 5;
    }
    i = i + 4;
  }
```

The rest of the code does a few more tests to check that creating another file with the same name fails, but that deleting it and recreating it works. It finally checks that the file can be found by its name, and deletes it:

```
  let unused: &FileBlock = null;
  if disk_create_file(NAME, NAME_LENGTH, &unused) == OK { return 6; }
  disk_delete_file(file, null);
  if disk_create_file(NAME, NAME_LENGTH, &unused) != OK { return 7; }
  let previous_file: &FileBlock = null;
  file = disk_find_file(NAME, NAME_LENGTH, &previous_file);
  if file == null { return 8; }
  disk_delete_file(file, previous_file);
  return 0;
}
```

To run this test:

- start the command editor, if it is not already running, by typing “w000c1172”+Enter, followed by “r” in the memory editor.

- type “F3”+“r” to load the builder source code and “F4”+“r” to edit it. Then delete all the code after `fn gpu_draw_char` (included), and type the above code instead.
- when done, type “F5”+“r” to save it, “F6”+“r” to compile it, and finally “F7”+“r” to run it. The result should be 0, meaning that all tests passed.

CHAPTER

22 Boot Loader and Drivers

Thanks to our file system we can now store data in flash memory without having to manually keep track of used and unused regions. Unfortunately, programs stored in the file system can no longer be run directly from flash memory. Indeed, a program split in several chunks stored in any order is completely broken, for many reasons. For instance, execution could "fall through" the end of a chunk, instructions might be split across chunks, and jump instruction targets might no longer be at their expected address. We thus need to reassemble a program into a contiguous sequence of bytes before we can run it. This is called *loading* a program.

A program can be loaded with `disk_read_file`, but this means that we need a "program loader" using this function to run a program. This loader could be stored in the file system, but then we would need another loader to load and run it. Ultimately, we need a loader which is not stored in the file system, or which can fit in a single block. This is called a *boot loader*, and we provide one in the first part of this chapter. We use it in the second part to load a very first version of our operating system kernel, containing only some "foundations". Namely the clock, graphics card and keyboard drivers, re-implemented in Toy, as well as the "disk driver" functions from the previous chapter.

22.1 Boot loader

22.1.1 Requirements

Our boot loader must be able to load and run a program stored in the file system under a predefined name. It should fit in a page and should run automatically upon reset.

Several errors could occur in the boot loader, for instance because the program to load does not exist or is invalid, or because the file system is corrupted. In such cases restarting the boot loader would likely trigger the same error again. The only solution would then be to fully erase the flash memory, in order to reboot with the Boot Assistant in ROM. To avoid this potential loss of data, we require our boot loader to change the boot mode to "boot from ROM" if an error occurs. A simple reboot is then sufficient to enter the Boot Assistant, which can then be used to try to repair the system, or at least to backup important data.

22.1.2 Implementation

To meet the above requirements we write a small program in Toy, compile it to native code, and store it in page 0 of the Flash0 memory bank, after the Vector Table.

Vector Table As described in Section 7.4, words 0, 1, and 3 of page 0 define the initial Stack Pointer, the Reset handler address, and the Hard Fault handler address, respectively (when booting from flash memory). The next words are used for exception and interrupt handlers, but they are not needed until the corresponding exceptions and interrupts are enabled. We thus store the boot loader code just after word 3.

The boot loader should provide a Reset handler, and a Hard Fault handler to change the boot mode in case of error. These handlers could be implemented in separate functions, but then we would need two function addresses to setup the Vector Table. Unfortunately, there is no easy and robust way to get the address of a function other than the first one. To solve this issue we implement both handlers in the same function. But this introduces a new problem: how can we know if this function is called because an error occurred? For this we take advantage of the fact that page 0, which starts at address 80000_{16}, is also mapped at address 0 (see Figure 6.4). Hence the main boot loader function, which by hypothesis starts after the 16 bytes of the Vector Table, is available both at interworking address 17 and $80000_{16} + 17$. And, as shown below, it can easily detect at which address it is running. We thus set the Reset handler to 17, and the Hard Fault handler to $80000_{16} + 17$.

To complete the Vector Table we set the initial Stack Pointer to 20088000_{16} (whose 4 bytes are 00, 80_{16} = 128, 08, and 20_{16} = 32 in little endian order), as in Section 9.6.1. Finally, to save space, we store in the 3^{rd} entry, otherwise unused, the name of the program to load, *i.e.*, of the kernel. We use the name "`toys`" because we are building a toy operating system, and because this name has exactly 4 characters:

```
static VECTOR_TABLE = [
  0, 128, 8, 32,        /* Initial Stack Pointer */
  17, 0, 0, 0,          /* Reset handler */
  't', 'o', 'y', 's',
  17, 0, 8, 0           /* Hard Fault handler */
];
```

Main function To implement the main boot loader function we first need to some definitions related to the file system, copied from the previous chapter:

```
struct DiskBlock {
  next_block: &DiskBlock
}
struct FileBlock {
  next_block: &DiskBlock,
  next_file: &FileBlock,
  name_length: u32,
  name: u32
}
```

```
struct SuperBlock {
  first_file: &FileBlock,
  first_free_block: &DiskBlock
}
const SUPER_BLOCK: &SuperBlock = 524544;
```

We also need the following functions, implemented later on. `copy_block` copies `size` bytes from `src` to `dst` and returns `dst + size`. `run` calls a function with the same parameters at the interworking address `code`:

```
fn copy_block(src: &u32, dst: &u32, size: u32) -> &u32;
fn run(code: &u32, heap: &u32, stack: &u32);
```

We start the main function by checking if it is called because an error occurred. In this case it is running from address $80000_{16} + 16$, and the `VECTOR_TABLE` expression, which computes the address of this data by subtracting an offset from the Program Counter, thus evaluates to 80000_{16}. Otherwise it is running from address 16, and the same expression thus evaluates to 0, *i.e.*, to `null`. If an error occurred we change the boot mode as described in Section 9.6.2, and then loop forever:

```
fn main() {
  const EEFC0_COMMAND_REGISTER: &u32 = 1074661892;
  if VECTOR_TABLE != null {
    *EEFC0_COMMAND_REGISTER = 1509949708; /*Clear Boot Mode Selection Bit*/
    loop {}
  }
```

Otherwise we iterate over the list of files, until we find the one named "`toys`". Thanks to the fact that "`toys`" has exactly 4 bytes (stored at address 8), we can compare it to another name with a single comparison between two 32 bit words:

```
  const KERNEL_NAME: &u32 = 8;
  let file = SUPER_BLOCK.first_file;
  while file.name_length != 4 || file.name != *KERNEL_NAME {
    file = file.next_file;
  }
```

To reduce code size we do not check if the file exists. If not the code will most likely trigger a Hard Fault at some point, which is fine thanks to the above code. Once the file is found we use another loop to copy all its blocks into a contiguous sequence of bytes, starting at address 20070100_{16} = 537329920 (we reserve the range [20070000_{16},20070100_{16}[for a new Vector Table, with proper handlers for the enabled exceptions and interrupts – see Figure 22.2):

```
  const KERNEL_ADDRESS: &u32 = 537329920;
  let dst = KERNEL_ADDRESS;
  let block = file as &DiskBlock;
  let offset = sizeof(FileBlock);
  while block as u32 > 256 {
    dst = copy_block(block as &u32 + offset, dst, 256 - offset);
```

```
    block = block.next_block;
    offset = 4;
  }
```

Note that the `while` loop does not check if `block` has a next block, but whether `block` is a valid block address. In the former case a separate `copy_block` call would be needed to copy the last block, which would increase code size. The above code avoids this but copies the unused bytes at the end of the last block. This is actually fine, and it guarantees that `copy_block` is always used to copy a number of bytes which is a multiple of 4.

Once the kernel is loaded we run its main function, with the initial Stack Pointer $20088000_{16} = 537427968$, with the following call (see below and Figure 22.2):

```
  const STACK_POINTER: &u32 = 537427968;
  run(KERNEL_ADDRESS + 1, dst, STACK_POINTER);
}
```

The `copy_block` function is a simple variant of the "memory copy" functions already implemented. It copies memory word by word, and assumes that the source and destination regions do not overlap (which is true):

```
fn copy_block(src: &u32, dst: &u32, size: u32) -> &u32 {
  let end = src + size;
  while src < end {
    *dst = *src;
    src = src + 4;
    dst = dst + 4;
  }
  return dst;
}
```

The last boot loader function is used to run the kernel. Here we assume that its main function has the same parameters as the `run` function. Namely the interworking address of the main function itself, the start address of a RAM region where the kernel can store its data, and its initial Stack Pointer. Since there is no way to call a function at a given address with a Toy expression, we implement `run` with native instructions.

When `run` starts executing, its arguments are in registers R0, R1 and R2. The function we want to call, at interworking address `code`, also expects its arguments in these registers. Hence, to call it, we just need to jump to it. This can be done with a native BX instruction (the called function is not expected to return, hence a BLX instruction is not needed). Before this, we set the Stack Pointer to `stack` with a MOV instruction (since the called function is not expected to return, we don't need to save the current Stack Pointer to restore it later):

```
fn run(code: &u32, heap: &u32, stack: &u32) [
  /*MOV_SP_R2*/18069;
  /*BX_R0*/18176;
]
```

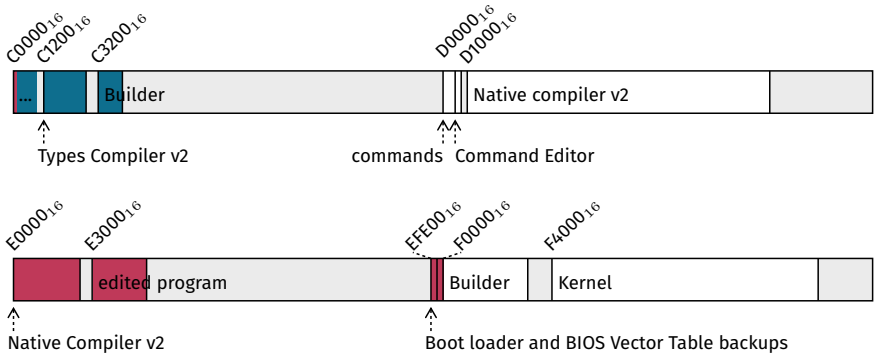


FIGURE 22.1 The Flash1 memory bank content at the end of Chapter 22. White, blue, red and gray areas represent source code, bytecode, native code, and unused memory, respectively (not to scale). See Figure 20.4 for the content before the Types Compiler.

22.2 Compilation and storage

We now need to type this program, compile it, and save it. Thanks to our file system we can store the source code as a file. However, this would make it harder to compile since our compiler can not work with files (yet). Instead, we use two steps:

1. we store the source code as a data buffer in the Flash1 memory bank, compile it, and store the compiled code in the same way.
2. we copy the source code into the file system.

For step 1 we choose to store the source code after the builder's source code, at address $F4000_{16}$ = 999424 = page 832, and the compiled code after the native compiler code, at address $E3000_{16}$ = 929792 = page 560 (see Figure 22.1).

To load and save the boot loader source code, edit the F8 and F10 commands as follows. In the memory editor, type "w000c1172"+Enter, followed by "r", to start the command editor. Then type "F8" and "e" to edit this command, and replace its code with the following (similar to the F3 command; see also Section 15.4.4):

```
fn 0
  cst 999424 cst 537330176 call 2708
  cst_0 retv
d LOAD_BOOT_LOADER_SOURCE_CODE
```

When done, type Escape and "s" to save it. In the same way, replace the F10 command with the following code (similar to the F5 command):

```
fn 0
  cst 537330176 cst 832 call 2836
  cst_0 retv
d SAVE_BOOT_LOADER_SOURCE_CODE
```

To compile the boot loader code, or in fact whatever source code has just been edited in RAM, update the F9 command as follows (see Section 20.4):

```
fn 0
  cst_0
  ptr 4 cst 537379328 cst 537330176 cst 917509
  cst 3179241480 cst 1225017232 cst 1241663748 cst 1208268159
  ptr 12 cst_1 add blx
  get 4 cst_0 ifne 64
  cst 537379328 cst 560 call 2836
  get 4 retv
d COMPILE_AND_STORE_EDITED_SOURCE_CODE
```

The second line is updated in order to compile the edited source code in RAM, at src_buffer = 20070200_{16} = 537330176 (see Figure 15.3), into dst_buffer = $2007C200_{16}$ = 537379328 (idem), with the native compiler (at interworking address $E0000_{16}$ + 4 + 1 = 917509 – see Figure 22.1). The added lines store the result in flash memory at $E3000_{16}$ = page 560, with buffer_flash, if the compilation is successful.

Compilation We can now type the boot loader source code and compile it. For this type "F2"+"r" to start a new program, and "F4"+"r" to edit it. Then type the source code listed in the previous section. For reference, we also provide this code in the `boot_loader.txt` file in https://ebruneton.github.io/toypc/sources.zip. When done, type "F10"+"r" to save it, and "F9"+"r" to compile it. The result should be 0, meaning that the compilation was successful. If not, type "F4"+"r" to fix the error, save the program and compile it again. Repeat this process until the compilation is successful.

Storage At this stage we can check if the boot loader code fits in a page by looking at its compiled code data buffer header, at address $2007C200_{16}$ (with the memory editor). The result is 246, which is indeed smaller than 256.

We can now copy its source code on disk, but it is too soon to copy its compiled code in page 0 of the Flash0 memory bank, hereafter called the *boot page*. Indeed, we still need the current content of this page to start the command editor and run its commands (to continue building our operating system in the next chapters). Instead, we copy the boot loader code in page 766 (at address $EFE00_{16}$), and we make a copy of the current boot page, hereafter called the *BIOS Vector Table*[1], in the next one (see Figure 22.1). For this type "F11"+"e" to define a new command as follows:

```
fn 0
  cst 929796 cst 766 cst 256 call 2724
  cst 524288 cst 767 cst 256 call 2724
  cst_0 retv
```

When done, type Escape and "r" to run it (we don't need to save it). The first line copies the boot loader (without its data buffer header, *i.e.*, starting at $E3000_{16}$ + 4 =

[1]Since this page contains the Vector Table of our Basic Input Output System.

929796) in page 766 with page_flash. The second lines copies the BIOS Vector Table, at address 80000_{16} = 524288, in the next page (both copy a full page).

Storing the boot loader source code on disk can't be done with a command editor command since we don't have the addresses of the functions written in the previous chapter. But we can use the builder program instead, which is easier anyway. For this, type "F3"+"r" to load it, and "F4"+"r" to edit it. Then delete the code after the `disk_delete_file` function (starting with `static NAME = ...`), and replace it with the following:

```
fn buffer_write(buffer: &u32, name: &u32, length: u32) -> u32 {
  let result = OK;
  let file = disk_find_file(name, length, null);
  if file == null {
    result = disk_create_file(name, length, &file);
  } else {
    disk_clear_file(file);
  }
  let block = file as &DiskBlock;
  let size = *buffer;
  if result == OK {
    result = disk_write_file(&block, buffer + 4, size);
  }
  return result;
}
```

```
static NAME = ['b','o','o','t','.','t','o','y'];
fn main() -> u32 {
  const BOOT_LOADER_SOURCE: &u32 = 999424;
  return buffer_write(BOOT_LOADER_SOURCE, NAME, 8);
}
```

The `buffer_write` function writes a data `buffer` in a file with the given `name`, and returns `OK` if the operation succeeds. For this it creates the file if it does not exist, or clears its content otherwise. It then appends the content of `buffer`, without its 4 bytes header, to this file. The `main` function uses it to save the boot loader source code, at address $F4000_{16}$ = 999424, in a "`boot.toy`" file.

When done, return in the command editor and type "F5"+"r" to save this code, "F6"+"r" to compile it, and "F7"+"r" to run it. The result of each step should be 0, meaning that no error occurred. Otherwise repeat the steps before the failure.

22.3 Drivers

In order to test our boot loader we first need to write a program to load. We can then try to run it with the boot loader but, to check that this process works, the loaded program must do something that we can verify. A simple option is to blink a LED. A more complex option is to display on screen each character typed on the keyboard.

This requires the clock, graphics card, and keyboard drivers. We already have them, but only in bytecode form. And since we want to eventually get rid of the bytecode interpreter, we need to re-implement them in Toy at some point. We thus choose the second option, which gives us the opportunity to write a very first version of our operating system kernel, containing the drivers re-implemented in Toy.

22.3.1 Clock driver

We start the kernel with an `entry` function having the same parameters as the boot loader's `run` function, as required. This function simply calls an `os_init` function, declared here but implemented at the very end. We also include the native `load8` and `store8` functions, copied from Section 20.3, which are always useful:

```
fn os_init(code: &u32, heap: &u32, stack: &u32);
fn entry(code: &u32, heap: &u32, stack: &u32) {
  os_init(code, heap, stack);
}

fn load8(ptr: &u32) -> u32 [ /*LDRB_R0_R0_0*/30720; /*MOV_PC_LR*/18167; ]
fn store8(ptr: &u32, value: u32) [ /*STRB_R1_R0_0*/ 28673; /*MOV_PC_LR*/ 1⟨
                                                                    ⟩8167;]
```

We continue with a re-implementation of the `clock_init` and `delay` functions from Sections 9.4 and 9.5 (see these sections for more details):

```
fn clock_init() {
  const EEFC0_MODE_REGISTER: &u32 = 1074661888;
  const EEFC1_MODE_REGISTER: &u32 = 1074662400;
  *EEFC0_MODE_REGISTER = 1536; /*wait=6*/
  *EEFC1_MODE_REGISTER = 1536; /*wait=6*/

  const PMC_MAIN_OSCILLATOR_REGISTER: &u32 = 1074660896;
  const PMC_PHASE_LOCK_LOOP_REGISTER: &u32 = 1074660904;
  const PMC_MASTER_CLOCK_REGISTER: &u32 = 1074660912;
  const PMC_STATUS_REGISTER: &u32 = 1074660968;

  *PMC_MAIN_OSCILLATOR_REGISTER = 3669769; /*Enable Crystal Oscillator*/
  while (*PMC_STATUS_REGISTER & 1) == 0 {} /*Wait until ready*/

  *PMC_MAIN_OSCILLATOR_REGISTER = 20446985; /*Select Crystal Oscillator*/
  while (*PMC_STATUS_REGISTER & 65536) == 0 {} /*Wait until selected*/

  *PMC_PHASE_LOCK_LOOP_REGISTER = 537280257; /*Configure Phase Lock Loop*/
  while (*PMC_STATUS_REGISTER & 2) == 0 {} /*Wait until ready*/

  *PMC_MASTER_CLOCK_REGISTER = 2; /*Select Phase Lock Loop output*/
  while (*PMC_STATUS_REGISTER & 8) == 0 {} /*Wait until output ready*/
```

```
  const SYSTICK_CONTROL_AND_STATUS_REGISTER: &u32 = 3758153744;
  *SYSTICK_CONTROL_AND_STATUS_REGISTER = 1; /*Enable*/

  const WATCHDOG_TIMER_MODE_REGISTER: &u32 = 1074666068;
  *WATCHDOG_TIMER_MODE_REGISTER = 32768; /*Disable*/
}

fn delay(millis: u32) {
  const SYSTICK_CONTROL_AND_STATUS_REGISTER: &u32 = 3758153744;
  const SYSTICK_RELOAD_VALUE_REGISTER: &u32 = 3758153748;
  const SYSTICK_CURRENT_VALUE_REGISTER: &u32 = 3758153752;

  *SYSTICK_RELOAD_VALUE_REGISTER = 10500 * millis;
  *SYSTICK_CURRENT_VALUE_REGISTER = 0;
  /*Wait until timer counts from 1 to 0*/
  while *SYSTICK_CONTROL_AND_STATUS_REGISTER & 65536 == 0 {}
}
```

We also re-implement the `boot_mode_select_rom` function from Section 9.6.2, without the Vector Table relocation part (we do this in `os_init` instead, see below):

```
fn boot_mode_select_rom() {
  const EEFC0_COMMAND_REGISTER: &u32 = 1074661892;
  const EEFC0_STATUS_REGISTER: &u32 = 1074661896;
  const CLEAR_BOOT_MODE_SELECTION_BIT_COMMAND: u32 = 1509949708;
  *EEFC0_COMMAND_REGISTER = CLEAR_BOOT_MODE_SELECTION_BIT_COMMAND;
  while *EEFC0_STATUS_REGISTER != 1 {}
}
```

22.3.2 Graphics card driver

Here we re-implement the graphics card driver functions from Section 10.4 (see this section for more details). We omit the last ones, namely `gpu_clear_screen`, `gpu_set_cursor`, `gpu_set_color`, and `gpu_draw_char`:

```
fn gpu_reset() {
  const PIOB_ENABLE_REGISTER: &u32 = 1074663424;
  const PIOB_OUTPUT_ENABLE_REGISTER: &u32 = 1074663440;
  const PIOB_SET_OUTPUT_DATA_REGISTER: &u32 = 1074663472;
  const PIOB_CLEAR_OUTPUT_DATA_REGISTER: &u32 = 1074663476;
  const PIOB_PULL_UP_DISABLE_REGISTER: &u32 = 1074663520;
  const PB12_PIN: u32 = 4096;

  *PIOB_ENABLE_REGISTER = PB12_PIN;
  *PIOB_OUTPUT_ENABLE_REGISTER = PB12_PIN;
  *PIOB_PULL_UP_DISABLE_REGISTER = PB12_PIN;
  *PIOB_CLEAR_OUTPUT_DATA_REGISTER = PB12_PIN;
  delay(10);
```

```
  *PIOB_SET_OUTPUT_DATA_REGISTER = PB12_PIN;
  delay(10);
}

fn spi_init() {
  const PIOA_DISABLE_REGISTER: &u32 = 1074662916;
  const PA25_26_27_28_PINS: u32 = 503316480;
  *PIOA_DISABLE_REGISTER = PA25_26_27_28_PINS;

  const PMC_PERIPHERAL_CLOCK_ENABLE_REGISTER: &u32 = 1074660880;
  const SPI_ID: u32 = 16777216;
  *PMC_PERIPHERAL_CLOCK_ENABLE_REGISTER = SPI_ID;

  const SPI_CONTROL_REGISTER: &u32 = 1073774592;
  const SPI_MODE_REGISTER: &u32 = 1073774596;
  const SPI_CHIP_SELECT_REGISTER: &u32 = 1073774640;
  *SPI_MODE_REGISTER = 1; /*Set master mode*/
  *SPI_CONTROL_REGISTER = 1; /*Enable*/
  *SPI_CHIP_SELECT_REGISTER = 5506; /*4MHz, 16bits, rising edge*/
}

fn spi_transfer(data: u32) -> u32 {
  const SPI_RECEIVE_DATA_REGISTER: &u32 = 1073774600;
  const SPI_TRANSMIT_DATA_REGISTER: &u32 = 1073774604;
  const SPI_STATUS_REGISTER: &u32 = 1073774608;

  while (*SPI_STATUS_REGISTER & 2) == 0 {} /*Wait transmitter ready*/
  *SPI_TRANSMIT_DATA_REGISTER = data;

  while (*SPI_STATUS_REGISTER & 1) == 0 {} /*Wait data received*/
  return *SPI_RECEIVE_DATA_REGISTER;
}

fn gpu_set_register(register : u32, value : u32) {
  const SELECT_REGISTER: u32 = 32768;
  spi_transfer(SELECT_REGISTER | register);
  spi_transfer(value & 255);
}

fn gpu_set_register_or_wait(register : u32, value : u32) {
  if register != 0 {
    gpu_set_register(register, value);
  } else {
    delay(value);
  }
}
```

```
static GPU_INIT_COMMANDS = [
  136,   6, /*PLL Control 1*/
  0,     1, /*delay 1ms*/
  137,   1, /*PLL Control 2*/
  0,     1, /*delay 1ms*/
  4,   129, /*Pixel Clock Setting*/
  20,   99, /*Horizontal Display Width*/
  21,    4, /*Horizontal Non-Display Period Fine Tuning*/
  22,    3, /*Horizontal Non-Display Period*/
  23,   25, /*HSYNC Start Position*/
  25,  223, /*Vertical Display Height 0*/
  26,    1, /*Vertical Display Height 1*/
  27,   21, /*Vertical Non-Display Period 0*/
  29,   21, /*VSYNC Start Position*/
  142, 128, /*Memory Clear Control*/
  0,   100, /*delay 100ms*/
  1,   128, /*Power And Display Control*/
  199,   1, /*Extra General Purpose IO*/
  138,  64, /*PWM1 Control*/
  52,   31, /*Horizontal End Point of Active Window 0*/
  53,    3, /*Horizontal End Point of Active Window 1*/
  54,  223, /*Vertical End Point of Active Window 0*/
  55,    1, /*Vertical End Point of Active Window 1*/
  64,  224, /*Memory Write Control 0*/
  68,   30  /*Blink Time Control*/
];

fn gpu_init() {
  gpu_reset();
  spi_init();
  let ptr = GPU_INIT_COMMANDS;
  while ptr < GPU_INIT_COMMANDS + 48 {
    gpu_set_register_or_wait(load8(ptr), load8(ptr + 1));
    ptr = ptr + 2;
  }
}
```

22.3.3 Keyboard driver

Finally, we re-implement the keyboard driver from Section 11.4.2, starting with the two character tables (see this section for more details):

```
static CHARACTERS = [
  0, 136, 0, 132, 130, 128, 129, 139, 0, 137, 135, 133, 131, 9, 96, 0, 0,
  141, 0, 0, 140, 113, 49, 0, 0, 0, 122, 115, 97, 119, 50, 0, 0,
  99, 120, 100, 101, 52, 51, 0, 0, 32, 118, 102, 116, 114, 53, 0, 0,
  110, 98, 104, 103, 121, 54, 0, 0, 0, 109, 106, 117, 55, 56, 0, 0,
  44, 107, 105, 111, 48, 57, 0, 0, 46, 47, 108, 59, 112, 45, 0, 0,
```

```
  0, 39, 0, 91, 61, 0, 0, 143, 0, 10, 93, 0, 92, 0, 0, 0,
  0, 0, 0, 0, 0, 8, 0, 0, 49, 0, 52, 55, 0, 0, 0, 48,
  46, 50, 53, 54, 56, 27, 142, 138, 43, 51, 45, 42, 57, 144, 0, 0,
  0, 0, 134,
  0, 136, 0, 132, 130, 128, 129, 139, 0, 137, 135, 133, 131, 9, 126, 0, 0,
  141, 0, 0, 140, 81, 33, 0, 0, 0, 90, 83, 65, 87, 64, 0, 0,
  67, 88, 68, 69, 36, 35, 0, 0, 32, 86, 70, 84, 82, 37, 0, 0,
  78, 66, 72, 71, 89, 94, 0, 0, 0, 77, 74, 85, 38, 42, 0, 0,
  60, 75, 73, 79, 41, 40, 0, 0, 62, 63, 76, 58, 80, 95, 0, 0,
  0, 34, 0, 123, 43, 0, 0, 143, 0, 10, 125, 0, 124, 0, 0, 0,
  0, 0, 0, 0, 0, 8, 0, 0, 49, 0, 52, 55, 0, 0, 0, 48,
  46, 50, 53, 54, 56, 27, 142, 138, 43, 51, 45, 42, 57, 144, 0, 0,
  0, 0, 134];

const KEYBOARD_HANDLER_STATE: &u32 = 1074666128;
const KEYBOARD_HANDLER_SHIFT: &u32 = 1074666132;
const KEYBOARD_HANDLER_CHAR: &u32 = 1074666136;

const NVIC_INTERRUPT_SET_ENABLE_REGISTER: &u32 = 3758153984;
const NVIC_INTERRUPT_CLEAR_ENABLE_REGISTER: &u32 = 3758154112;
const USART_ID: u32 = 131072;

fn keyboard_init() {
  *KEYBOARD_HANDLER_STATE = 0;
  *KEYBOARD_HANDLER_SHIFT = 0;
  *KEYBOARD_HANDLER_CHAR = 0;

  const PIOA_ENABLE_REGISTER: &u32 = 1074662912;
  const PA10_17_PINS: u32 = 132096;
  *PIOA_ENABLE_REGISTER = PA10_17_PINS;

  const PMC_PERIPHERAL_CLOCK_ENABLE_REGISTER: &u32 = 1074660880;
  *PMC_PERIPHERAL_CLOCK_ENABLE_REGISTER = USART_ID;

  const USART_CONTROL_REGISTER: &u32 = 1074364416;
  const USART_MODE_REGISTER: &u32 = 1074364420;
  const USART_INTERRUPT_ENABLE_REGISTER: &u32 = 1074364424;

  *USART_MODE_REGISTER = 1008; /*PS/2 protocol configuration*/
  *USART_INTERRUPT_ENABLE_REGISTER = 1; /*Enable RX ready interrupt*/
  *NVIC_INTERRUPT_SET_ENABLE_REGISTER = USART_ID;
  *USART_CONTROL_REGISTER = 16; /*Enable receiver*/
}

fn keyboard_put_char(c: u32) {
  if *KEYBOARD_HANDLER_CHAR == 0 {
    *KEYBOARD_HANDLER_CHAR = c;
```

```
  }
}

fn keyboard_get_char() -> u32 {
  *NVIC_INTERRUPT_CLEAR_ENABLE_REGISTER = USART_ID;
  let c = *KEYBOARD_HANDLER_CHAR;
  *KEYBOARD_HANDLER_CHAR = 0;
  *NVIC_INTERRUPT_SET_ENABLE_REGISTER = USART_ID;
  return c;
}

fn keyboard_wait_char() -> u32 {
  let c = 0;
  while c == 0 { c = keyboard_get_char(); }
  return c;
}
```

Here we merge the (keyboard_)skip_code, press_shift, release_shift, put_extended_code, and put_code functions into a single one, taking an additional action parameter equal to one of the following constants:

```
const SKIP_CODE: u32 = 0;
const RELEASE_SHIFT: u32 = 1;
const PRESS_SHIFT: u32 = 2;
const PUT_EXTENDED_CODE: u32 = 3;
const PUT_CODE: u32 = 4;

fn keyboard_handle_code(action: u32, scancode: u32) {
  if action == RELEASE_SHIFT {
    *KEYBOARD_HANDLER_SHIFT = 0;
  } else if action == PRESS_SHIFT {
    *KEYBOARD_HANDLER_SHIFT = 132;
  } else if action == PUT_EXTENDED_CODE {
    keyboard_put_char(scancode + 128);
  } else if action == PUT_CODE {
    keyboard_put_char(load8(CHARACTERS + scancode + *KEYBOARD_HANDLER_SHIF
                                                                    T));
  }
}
```

We then replace the addresses of these functions with these constants in the Finite State Machine transition table:

```
static TRANSITION_TABLE = [
  4, 0, 0, 30, 0, 20, 0, 10, 2, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
  3, 0, 0, 0, 0, 0, 0, 40, 0, 0,
  0, 10, 0, 0, 0, 0, 0, 10, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
```

and we update the keyboard_handler function to use keyboard_handle_code:

```
fn keyboard_action_column(scancode : u32) -> u32 {
  if scancode == 18 { return 8; }
  if scancode < 132 { return 0; }
  if scancode == 224 { return 4; }
  if scancode == 240 { return 6; }
  return 2;
}

fn keyboard_handler() {
  const USART_RECEIVER_HOLDING_REGISTER: &u32 = 1074364440;
  let scancode = *USART_RECEIVER_HOLDING_REGISTER;
  let cell = TRANSITION_TABLE + *KEYBOARD_HANDLER_STATE + keyboard_action_⌇
                                                          ⌇column(scancode);
  keyboard_handle_code(load8(cell), scancode);
  *KEYBOARD_HANDLER_STATE = load8(cell + 1);
}
```

22.3.4 Disk driver

The new "disk driver" functions implemented in the previous chapter can be added here, starting from the block data structure definitions and ending with the disk_delete_file function. Indeed, they will be needed by our operating system to work with files.

22.3.5 Main function

To finish this initial version of the kernel we need to implement the main function, os_init. As discussed above, this function should display on screen each character typed on the keyboard. But this is not the only thing it has to do.

Indeed, to run this kernel with the boot loader, we first need to copy the boot loader in the boot page. However, after our test with the kernel, we need to return in the memory editor to continue building our operating system. For this, os_init must restore the BIOS Vector Table in the boot page, as soon as possible (in case a next step fails).

The os_init function also needs to configure an USART handler in the Vector Table, to handle USART interrupts with the keyboard driver. However, as explained above, we don't want to change the BIOS Vector Table. The solution is to define a new Vector Table in RAM. In summary, os_init must do the following:

1. define and enable a new Vector Table in RAM.
2. restore the BIOS Vector Table in the boot page.
3. initialize the drivers.
4. display on screen each character typed on the keyboard.

For step 1 we first need an USART handler function. When it returns, such a function must make sure to restore the registers to the values they had before the interrupt (so that the interrupted program can resume execution correctly). This is why the USART handler implemented in Section 11.4.2 saves and restores the registers R4 to R6, in addition to the R0 to R3 registers automatically saved by the microprocessor. Indeed, this handler calls the bytecode interpreter, which uses registers R0 to R6.

Here we want to compile the whole kernel into native code, and thus no longer need to use the bytecode interpreter to call `keyboard_handler`. Note also that this function, and the functions it calls directly or indirectly, have at most 2 parameters and use very simple expressions. In other words, they do not use registers other than R0 to R3 (at most). Since these registers are automatically saved and restored (see Section 11.3.2), our USART handler can just call `keyboard_handler` directly:

```
fn usart_handler() {
  keyboard_handler();
}
```

For step 2 we implement the following function, which uses the disk driver functions to copy the BIOS Vector Table backup (at address EFF00_{16} = 982784 – see Figure 22.1), into the boot page, at address 80000_{16} = 524288:

```
fn restore_bios_vector_table() {
  const BIOS_VECTOR_TABLE_BACKUP: &u32 = 982784;
  const BOOT_PAGE: &u32 = 524288;
  block_copy_bytes(BIOS_VECTOR_TABLE_BACKUP, BOOT_PAGE, 256);
  block_write(BOOT_PAGE as u32);
}
```

With this we can finally implement the `os_init` function, as described above. To simplify we only set the USART handler in the Vector Table (no Hard Fault handler), relocated with the Vector Table Offset Register (see Section 7.4) at the address reserved for this by the boot loader (257 bytes before `code` – see Section 22.1.2 and Figure 22.2):

```
fn os_init(code: &u32, heap: &u32, stack: &u32) {
  const VECTOR_TABLE_OFFSET_REGISTER: &&u32 = 3758157064;
  const USART_HANDLER_OFFSET: u32 = 132;
  let vector_table = code - 257;
  *(vector_table + USART_HANDLER_OFFSET) = usart_handler + 1;
  *VECTOR_TABLE_OFFSET_REGISTER = vector_table;

  restore_bios_vector_table();
  clock_init();
  keyboard_init();
  gpu_init();
  gpu_set_register(2, '>');
  loop {
    gpu_set_register(2, keyboard_wait_char());
  }
}
```

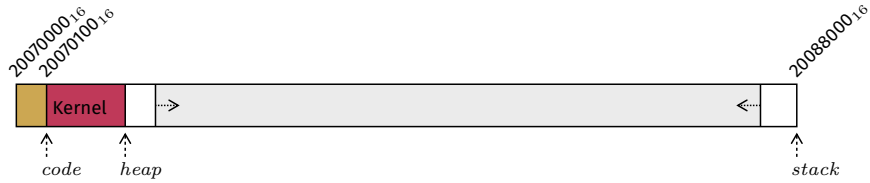


FIGURE 22.2 The layout of the kernel in RAM, and the *code*, *heap*, and *stack* arguments passed to `os_init`. The kernel code (red) is just after the relocated Vector Table (yellow). Red, white, and gray areas represent native code, used memory, and unused memory, respectively (not to scale).

22.4 Compilation and test

To compile and test this initial kernel we proceed in two steps, as for the boot loader:

1. we store the source code as a data buffer in the Flash1 memory bank, compile it, and store the compiled code in the same way.
2. we copy the compiled code into the file system, flash the boot loader in the boot page, and reset the Arduino.

For step 1 we can reuse the data buffer at address $F4000_{16}$, which currently contains the boot loader source code. Indeed, we saved this code in the "`boot.toy`" file. In particular, we can reuse the F8 and F10 commands, as is. Let's just update their description to reflect their new purpose. With the command editor, update the F8 command description to:

```
fn 0
  cst 999424 cst 537330176 call 2708
  cst_0 retv
d LOAD_KERNEL_SOURCE_CODE
```

and the F10 command description to:

```
fn 0
  cst 537330176 cst 832 call 2836
  cst_0 retv
d SAVE_KERNEL_SOURCE_CODE
```

The kernel source code includes the disk driver functions from the previous chapter. To avoid re-typing them, first make a copy of the builder source code by typing "F3"+"r" and "F10"+"r". Then type "F4"+"r" to replace its beginning part with the clock, graphics card, and keyboard drivers, and its end part with the main function from Section 22.3.5. The result should be identical to the `toys_v0.txt` file in https://ebruneton.github.io/toypc/sources.zip. When done, type "F10"+"r" to save it and "F9"+"r" to compile it. If necessary, repeat these steps until the compilation is successful.

For step 2 we modify the builder as follows. Type "F3"+"r" and "F4"+"r" to edit the builder source code, delete the code after the `buffer_write` function (starting with `static NAME = ...`), and replace it with the following:

```
fn flash_boot_loader_and_reset() {
  const BOOT_LOADER_BACKUP: &u32 = 982528;
  const BOOT_PAGE: &u32 = 524288;
  block_copy_bytes(BOOT_LOADER_BACKUP, BOOT_PAGE, 256);
  block_write(BOOT_PAGE as u32);

  const RESET_CONTROL_REGISTER: &u32 = 1074665984;
  const RESET_COMMAND: u32 = 2768240653;
  *RESET_CONTROL_REGISTER = RESET_COMMAND;
}

static NAME = ['t','o','y','s'];
fn main() -> u32 {
  const COMPILED_CODE: &u32 = 929792;
  let result = buffer_write(COMPILED_CODE, NAME, 4);
  if result == OK { flash_boot_loader_and_reset(); }
  return result;
}
```

As its name implies, `flash_boot_loader_and_reset` copies the boot loader, at address EFE00_{16}, into the boot page, and then resets the Arduino (see Section 9.7). The main function calls it after saving the compiled code of the kernel, at address $\text{E3000}_{16} = 929792$, in a "`toys`" file on disk.

When done, type "F5"+"r" to save the builder, and"F6"+"r" to compile it. If necessary, repeat these steps until the compilation is successful. Finally, type "F7"+"r" to run the builder. The Arduino should reset and you should see a ">" prompt on screen. Furthermore, any character typed should be displayed on screen, showing that the boot loader successfully loaded and launched the kernel. After a new reset, the Arduino should restart with the memory editor.

CHAPTER

23 Processes and System Calls

Thanks to our boot loader we can now reassemble a program stored in the file system into a contiguous sequence of bytes and run it. However, the boot loader can only launch a single initial program, namely the operating system kernel. To run other programs a solution is to implement a kernel which, in turn, can load and run other programs. Our kernel currently contains the clock, graphics card, keyboard and disk drivers. In particular, it contains a `disk_read_file` function which can load a program stored on disk. Hence we "just" need to improve our kernel so that it can also run programs. In fact things are not so simple.

In order to load and run a program the kernel must find a RAM region not already used by other programs. For this it must keep track of which regions are currently used and which ones are free. Moreover, to do something useful, programs launched by the kernel should be able to use the keyboard, the screen, or the disk. The kernel already provides functions for this, and we don't want to re-implement them in each program. In other words, the kernel should provide a way for a program to use its services. This chapter provides a new version of our kernel which does this. We test it at the end with a small program launching itself recursively to compute a factorial.

23.1 Requirements

Most operating systems can run several programs concurrently. Such systems can, for instance, run a music player program "at the same time" as a compiler. In fact they run the music player for a few milliseconds, then the compiler for a few milliseconds, then the music player again, etc. They are called multitasking operating systems. The opposite is a monotasking system. Such a system cannot run several programs concurrently. Monotasking systems are much simpler to implement than multitasking ones. For this reason, we use a monotasking model for our toy operating system, presented below.

A program loaded and started by the kernel is called a *process*. A process is *spawned* by loading the compiled code of a program in RAM and by calling its main function. A *parent* process P can ask the kernel to spawn a *child* process Q but, to get a monotasking system, we require P to be suspended until Q terminates. In other words, processes must behave like functions calling each other (when a function calls another, the caller is suspended until the callee returns).

By analogy with functions, which can have parameters and can return a result, we also the require the ability to spawn a process with some arguments, and the ability to return a result from a child process to its parent. Moreover, a process should be able to stop and return a result at any time, and not just when its main function returns (like our compiler with its `panic` function). We also add a safety requirement: a bug in a process should not crash everything and require the user to reboot the computer. Instead, the process should be terminated and should return some error value to its parent, which should resume its execution normally.

As described above, the kernel should provide a way for processes to use its services. In this chapter we require only two services: one to spawn a child process with some arguments, and another to stop the current process and to return a value to its parent. We define more services in the next chapter.

To solve the issue that the boot loader can only launch a single program, the kernel could ask the user to type on the keyboard the name and arguments of a program to launch, run it until completion, ask which other program to launch next, etc. But a better method is to implement this *command line interpreter*, also called a *shell*, in a process. Indeed, this reduces the kernel size and makes it easier to change the shell implementation. Hence, we require our kernel to spawn a single process when it starts, from the program file named "`shell`".

23.2 Design

Since spawning a process should behave similarly to calling a function, one could be tempted to use actual function calls for this. More precisely, a process could call a "`spawn`" function in the kernel, which would itself load some program's code and call its main function. This would lead to a single stack, with the stack frames of "`spawn`" function calls interleaved between the stack frames of process function calls.

Unfortunately this simple design cannot ensure our safety requirement. A bug in a process might cause it to write random values in the code, heap, or stack of the kernel, or of another process. This could trigger a Hard Fault some time later, when the kernel or this other process uses this incorrect instruction or value. To prevent this, one method is to completely *isolate* each process from each other, and from the kernel. This means that each process should have its own memory region to store its code, heap and stack. And that it should not be able to access any other memory region, in any way. A consequence is that the kernel and the processes should each have their own stack. Another is that processes cannot call functions in the kernel or in another process (because, for this, they would need to read the function's instructions, outside their own memory region).

Most operating systems isolate processes in this way, and thus need to use one stack per process, and to avoid function calls to system services. This is why the ARM Cortex M3, like many other microprocessors, provides several features specifically designed for these tasks. We present them below, before presenting our final design.

23.2.1 Memory Protection Unit

The Memory Protection Unit (see Section 7.1) is designed to enforce isolation between processes. We show how to use it in Chapter 26.

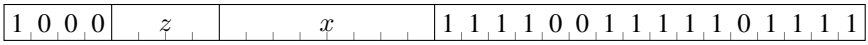
23.2.2 Process Stack Pointer

To make it easier to have one stack per process, and one stack for the kernel, the Cortex M3 has two Stack Pointer (SP) registers. One is called the Main Stack Pointer (MSP), and is intended to be used by the kernel. The other is called the Process Stack Pointer (PSP) and, as its name implies, is intended to be used by processes. Instructions which the use "the" SP, such as PUSH or POP, actually use either the MSP or the PSP, depending on the current state of the microprocessor:

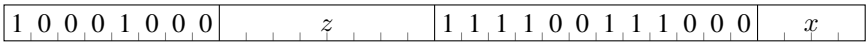
- when an exception or interrupt handler is executing (other than the Reset handler), which is called the *handler mode*, these instructions always operate on the Main Stack Pointer register.
- otherwise, in *thread mode*, the register used depends on a "stack selection" bit in a so called CONTROL register. When this bit is 0, which is the default value, the MSP is used. When it is 1, the PSP is used instead.

So far we never changed the stack selection bit, and we were thus using the MSP all the time, in handler and thread mode. It is however possible to change this bit, and we present a method to do this in the next section. This enables using the PSP with instructions such as PUSH or POP, but also with ADD, SUB, MOV, LDR, or STR instructions using the SP as operand. It is also possible to get or set the value of the MSP and of the PSP directly, with the following instructions:

MRS $\mathrm{R}z \leftarrow \mathrm{SYSR}x$



MSR $\mathrm{SYSR}z \leftarrow \mathrm{R}x$



The "Move to Register from Special Register" (MRS) instruction copies the value of the special register SYSRx into a normal register Rz. The "Move to Special Register from Register" (MSR) instruction does the opposite. The MSP, PSP, and CONTROL registers correspond to SYSR8, SYSR9, and SYSR20, respectively. Rx and Rz must not be the PC or the SP.

23.2.3 Interrupt entry and return

As described in Section 11.3.2, the microprocessor pushes some values on the stack when an exception or interrupt occurs, and pops them when it terminates. In fact things are a bit more complex since there are two stack pointers. This section gives more details about this process (but not all of them – see [17] for a complete description).

When an interrupt[1] becomes active the microprocessor pushes an *interrupt stack frame* on the *currently selected stack*. This stack frame contains (see Figure 23.4):

- the value of the R0, R1, R2, R3, R12, and R14 (LR) registers.
- the *return address*, *i.e.*, the address of the instruction to return to when the handler terminates. This address is *not* an interworking address.
- the value of a special "status" register which contains, among other things, the result of the last CMP instruction (see Section 7.3.1). For the same reason behind interworking addresses (see Section 7.2.1), bit 24 of this value must always be 1.

After this the microprocessor sets the Link Register (LR) to a special value called *EXC_RETURN*, which encodes the current execution mode[2] and stack selection bit:

Mode	Stack	EXC_RETURN
Handler	Main	$FFFFFFF1_{16}$
Thread	Main	$FFFFFFF9_{16}$
Thread	Process	$FFFFFFFD_{16}$

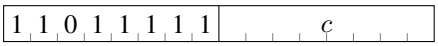
Finally, the Cortex M3 changes the current mode to handler mode, and the current stack to the Main stack. The interrupt handler then starts to execute, on this stack. When *any* one of the above EXC_RETURN value is copied into the Program Counter, the reverse process happens: the execution mode and the current stack are set according to the EXC_RETURN value, and the R0, R1, R2, R3, R12, R14, R15 (Program Counter), and "status" registers are set to values popped from this new stack.

Note that an interrupt handler can return with a different EXC_RETURN value than the one set in the LR when it started. For instance, an interrupt which occurred in thread mode while the Main stack was selected can set the PC to $FFFFFFFD_{16}$. In this case execution resumes in thread mode but on the Process stack (which should then contain a valid interrupt stack frame). An interrupt handler can also change the Process Stack Pointer during its execution. We use both possibilities in the next sections, to spawn the initial process and to switch execution between processes.

23.2.4 SVC instruction and interrupt

The Cortex M3 provides several features to make it possible to use services provided by the kernel without function calls. One is the possibility to configure the Memory Protection Unit so that memory is protected only in thread mode. Another is the following instruction:

SVC SVC interrupt



The "Supervisor Call" instruction triggers a special interrupt called an "SVC interrupt", corresponding to the 11^{th} entry of the Vector Table (counting from 0,

[1] The process is the same for exceptions.
[2] An interrupt can become active in handler mode. We explain this in the next chapter.

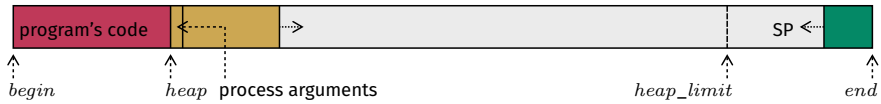


FIGURE 23.1 The layout of a process in RAM. Red, yellow, green, and gray areas represent native code, heap, stack, and unused memory, respectively.

i.e., at offset 44). Its c operand is unused. A program executing this instruction is immediately suspended, and resumes only when the stack frame pushed on interrupt entry is popped. In other words, this instruction behaves a bit like a function call to the SVC handler, hence its name. But it uses an interrupt for this, which executes in handler mode. Hence, if the Memory Protection Unit is configured as explained above, the SVC handler can access any memory, and in particular the kernel's code, heap and stack. This provides a way for processes to use kernel services, despite their memory isolation, without using actual function calls.

23.2.5 Final design

Thanks to the above Cortex M3 features we can achieve our requirements with the following design. First of all, to make it easier to isolate processes, we store each one in a contiguous region of RAM, noted $[begin, end[$. Due to limitations of the Memory Protection Unit, these addresses must be multiple of 32 bytes (see Chapter 26). We then organize this region as follows (see Figure 23.1):

- a process's memory region starts with the program's compiled code, which must itself start with its main function.
- the code is followed by a *heap* region, between $heap$ and $heap\_limit$, where the program can store its own data. The process arguments are put at the beginning of this region, as a data buffer (*i.e.*, a 32-bit header containing the arguments size s, followed by s bytes of arbitrary data).
- the heap is followed by a region reserved for the stack, between $heap\_limit$ and end. As usual, the stack grows in decreasing address order. The stack pointer is thus initialized to end. The heap can be used in arbitrary ways, but usually grows in increasing address order.

To launch this process we need a way to "tell" the program where the process arguments are, and where it can store its own data. For this we require its main function to have two parameters, corresponding to $heap$ and $heap\_limit$. This specifies both the region where data can be stored, and the location of the process arguments (since, by hypothesis, they start at $heap$).

To spawn a new process while keeping track of the used and unused regions of memory, a simple solution is to put it just after its parent (see Figure 23.2). This leads to a simple "stack" of processes, pushed when spawn and popped when they terminate. But this design wastes the parent's unused memory, which cannot use it

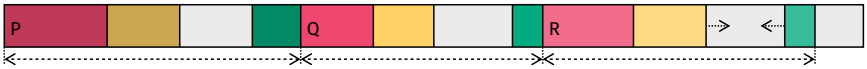


FIGURE 23.2 Storing processes one after the other would waste memory. Here spawning a process would be possible if the unused memory was not fragmented.

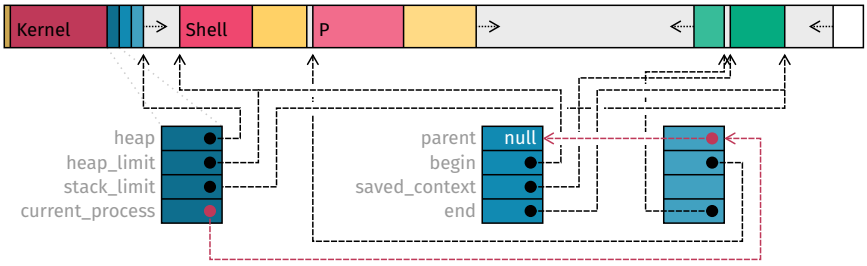


FIGURE 23.3 Processes are stored in the unused memory region of their parent, between the areas reserved for the kernel's heap (blue) and stack (white). The kernel's heap contains a `Kernel` data structure (dark blue) with a linked list (red links) of `Process` data structures, starting with the current one. Each structure describes the start and end address of a process and links to its parent.

until its child terminates since it is suspended. To solve this issue we use another design instead: we put child processes inside their parent process, like Russian dolls (see Figure 23.3). This allows each child process to use as many bytes as possible of its parent's unused memory (up to 2×31 bytes can be wasted due to the alignment constraints of $begin$ and end). Finally, we use all the available memory between the kernel's heap and stack, for which we reserve fixed amounts of RAM, to spawn the initial shell process (see Figures 22.2 and 23.3).

23.3 Data structures and algorithms

To implement our requirements we store in the kernel's heap the following data structures, and we use them in the algorithms described below (see Figure 23.3):

- a `Kernel` struct contains the limits of the regions reserved for the kernel's heap and stack. It also contains a `heap` pointer to the first free byte in the kernel's heap. Finally, it links to a struct representing the currently running process.
- a `Process` struct per process. Each struct describes the start and end address of the process, and links to the `Process` struct of its parent.

23.3.1 System calls

In order to allow processes to "call" services provided by the operating system kernel, *i.e.*, to make *system calls*, we use the SVC instruction. But this is not sufficient. This

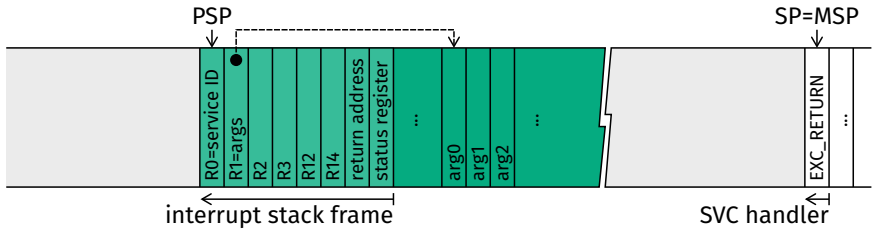


FIGURE 23.4 When the SVC handler starts it pushes the EXC_RETURN value on the Main stack (white). The stack frame pushed on interrupt entry (light green), on the Process stack (green), must start with a service ID and a pointer to its arguments.

instruction can only "call" a single interrupt handler function, and we would like our kernel to provide several services (for the keyboard, the screen, the disk, etc). To solve this we can pass to this handler an integer specifying the desired service, *i.e.*, a *service ID*. But we need to decide how. We also need a way to pass some arguments to each service, and to get a result back. This can be done via registers, with the stack, or a mix of both. Here, to simplify the implementation, we require the following:

- before using the SVC instruction R0 and R1 should contain the service ID and a pointer to the service arguments, respectively. The registers not automatically saved in the interrupt stack frame should not contain any important value (so that the interrupt handler does not need to save and restore them).
- when the interrupted process resumes R0 should contain the service's result.

As a consequence, when the SVC handler function starts executing, the Process stack and the Main stack look as depicted in Figure 23.4:

- the top of the Process stack contains the interrupt stack frame, pushed on interrupt entry, starting with the service ID and the pointer to the service arguments.
- the top of the Main stack contains the EXC_RETURN value. Indeed the handler function has no argument and, like any function, its pushes its arguments and the LR on the stack when it starts (see Figure 20.2). And the LR contains the EXC_RETURN values as described Section 23.2.3.

23.3.2 Spawn

We then define a "spawn" service, with ID 0 and with the following arguments:

- a destination pointer where to spawn the process, dst,
- a pointer to the name of the file containing its code, $name$,
- the length of this name, $name\_length$,
- a pointer to the process arguments, $args$,
- the size of these arguments, $args\_length$.

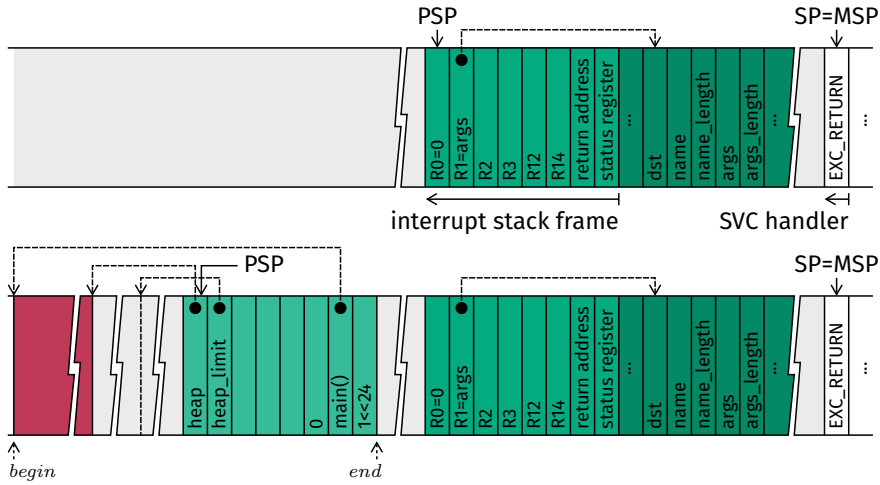


FIGURE 23.5 The memory content when the SVC handler for a "`spawn`" system call starts (top), and just before it returns (bottom).

As a consequence, when the SVC handler for a "`spawn`" system call starts executing, the Process stack of the calling process looks as depicted in Figure 23.5. We then use the following algorithm to spawn the child process (see Figures 23.1 and 23.5):

1. compute *begin* by rounding up *dst* to a multiple of 32, and *end* by rounding down the current Process Stack Pointer to a multiple of 32 (or the kernel's *stack_limit* for the initial process – there is no current process in this case).
2. compute *heap_limit* by subtracting a minimum stack size from *end* (512 bytes – if a program needs more space for its stack, it can then further decrease *heap_limit* to make sure not to store data beyond this).
3. load the program's code at *begin*, and compute *heap* as *begin* plus the code size.
4. copy the process arguments after the code, at address *heap*.
5. *synthesize an interrupt stack frame* in the new process stack area. Set the saved R0 and R1 values to *heap* and *heap_limit*, and the return address to *begin*, *i.e.*, the address of the new process's main function.
6. save the current Process Stack Pointer in a field of the current `Process` struct named `saved_context` (see Figure 23.3). Then change it, with an MSR instruction, to the top of the above synthesized stack frame.
7. create and initialize a new `Process` struct representing the spawned process, which becomes the new `Kernel`'s current process.

Thus, when the SVC handler returns, *the synthesized interrupt stack frame is popped from the stack of the spawned process*. The effect is a jump to the main function of this new process, with its *heap* and *heap_limit* parameters in R0 and R1, as required. In other words, the new process starts executing, with its own stack.

23.3.3 Exit

To terminate this new process and return a value to its parent we define an "`exit`" service, with ID 1 and a single *result* argument, and using the following algorithm:

1. delete the `Process` struct representing the child process, and sets the `Kernel`'s current process to its parent.
2. set the Process Stack Pointer to the value saved in the parent `Process` (at step 6 of the `spawn` algorithm). That is, to the top of the interrupt stack frame which was pushed when the parent called `spawn`.
3. set the saved R0 value in this interrupt stack frame to *result*.

Thus, when the SVC handler returns, execution resumes in the parent process, at the instruction just after the SVC. Moreover, the child process's result is in R0, as a normal function result.

23.3.4 Error handling

Several errors can occur when spawning a process. For instance, the program file might not exist, or there might not be enough memory to load it. In such cases we want to return an error code to the parent. There are several ways to do this. A convenient method for users is to return it in R0, like the exit value of the spawned process. To distinguish the two cases (spawning error *vs* spawned process's result) we spit the 32-bit result in two parts:

- we store the error, if any, in the most significant byte, called the result's *status*.
- we store the result of the spawned process in the 3 least significant bytes (this leaves only 2^{24} possible result values, but this is acceptable for a toy system).

The advantage of this encoding is that if the process is spawned successfully, *i.e.*, if the result's status is 0, then the 32-bit result is directly equal to the exit value. For consistency, we use this encoding for all system calls.

23.3.5 Initial process

After it has initialized itself the kernel can spawn the initial shell process with a `spawn` system call, like for any other process. Note however that this system call is made by the kernel, and not by a process. At this stage the kernel is running in Thread mode on the Main stack. This has two consequences:

- the SVC interrupt stack frame is pushed on the Main stack, instead of the Process Stack as for all other system calls. The SVC handler is pushed on the same stack just after that. Hence, the interrupt stack frame is just below the EXC_RETURN value in this case (see Figure 23.6).
- the SVC interrupt handler must change the EXC_RETURN value to FFFFFFFD_{16} to use the Process Stack on interrupt exit. Otherwise the interrupt would return in the kernel.

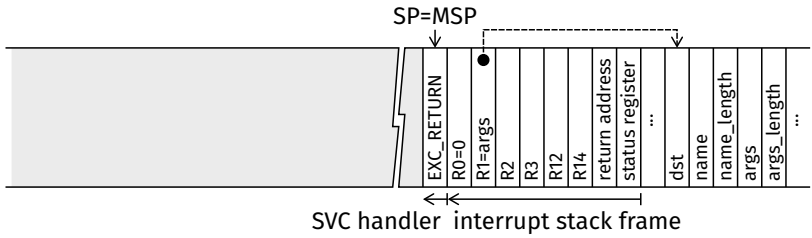


FIGURE 23.6 The Main Stack when the SVC handler for the initial "`spawn`" starts.

23.4 Implementation

We can now extend the first version of our kernel, written in the previous chapter, with the algorithms described above. We start by declaring a new function to spawn the initial shell process, after the initialization of the kernel:

```
fn os_init(code: &u32, heap: &u32, stack: &u32);
fn os_spawn_shell();

fn entry(code: &u32, heap: &u32, stack: &u32) {
  os_init(code, heap, stack);
  os_spawn_shell();
}
```

We continue by adding some error code definitions, and a utility function to compute a result with an error status (see Section 23.3.4):

- `NOT_FOUND` if spawning a process fails because its program file is not found.
- `INTERNAL_ERROR` if a process triggers a Hard Fault and thus terminates abnormally.
- `INVALID_STATE` if the initial process calls `exit` (it has no parent to return to).

```
const OK: u32 = 0;
const INVALID_ARGUMENT: u32 = 1;
const INVALID_STATE: u32 = 2;
const NOT_FOUND: u32 = 3;
const ALREADY_EXISTS: u32 = 4;
const OUT_OF_MEMORY: u32 = 5;
const INTERNAL_ERROR: u32 = 6;

fn error_result(error: u32) -> u32 { return error << 24; }
```

We then define, after the disk driver functions and before the `usart_handler`, some types corresponding to the data structures defined above (see Figure 23.3). `Context` corresponds to an interrupt stack frame (see Figure 23.4). The last 3 fields of `Kernel` contain the addresses of functions implementing our system calls (we add a temporary `draw_char` system call for testing purposes):

```
struct Context {
  r0: u32,
  r1: u32,
  r2: u32,
  r3: u32,
  r12: u32,
  r14: u32,
  return_address: u32,
  status_register: u32
}
struct Process {
  parent: &Process,
  begin: &u32,
  end: &u32,
  saved_context: &Context
}
struct Kernel {
  heap: &u32,
  heap_limit: &u32,
  stack_limit: &u32,
  current_process: &Process,
  spawn: u32,
  exit: u32,
  draw_char: u32
}
```

To find the `Kernel` allocated in the kernel's heap (see Figure 23.3) we store a pointer to it in a General Purpose Backup Register, after the ones used by the keyboard driver (at address $400E1A9C_{16} = 1074666140$ – see Section 11.4.2).

```
const KERNEL_POINTER_REGISTER: &&Kernel = 1074666140;
fn os_kernel() -> &Kernel { return *KERNEL_POINTER_REGISTER; }
```

23.4.1 Spawn

To implement the `spawn` system call we need the MRS and MSR instructions to get and set the Process Stack Pointer. This can only be done with native functions:

```
fn get_process_stack_pointer() -> &u32 [
  /*MRS_R0_PSP*/ 2148135919;
  /*MOV_PC_LR*/ 18167;
]
fn set_process_stack_pointer(value: &u32) [
  /*MSR_PSP_R0*/ 2282353536;
  /*MOV_PC_LR*/ 18167;
]
```

We use them in the following utility function, which changes the current process to the given one (the change becomes effective when the SVC handler calling it returns):

```
fn os_set_current_process(kernel: &Kernel, process: &Process) {
  set_process_stack_pointer(process.saved_context as &u32);
  kernel.current_process = process;
}
```

The spawn system call takes some pointers as arguments. A bug in the calling process might cause them to point in a reserved memory region, in the kernel's code, heap, or stack, or in another process. Since spawn reads and writes data at these addresses, such a bug could cause a crash of the kernel, even if processes are isolated. To avoid this, all system calls must check that their arguments are correct. This is the goal of the following function, which checks if the `size` bytes starting at `ptr` are inside the memory region of a process (while being careful to avoid overflows):

```
const FALSE: u32 = 0;
const TRUE: u32 = 1;
fn process_contains_buffer(self: &Process, ptr: &u32, size: u32) -> u32 {
  if ptr < self.begin || ptr >= self.end { return FALSE; }
  if size > self.end - ptr { return FALSE; }
  return TRUE;
}
```

We can now implement a function to spawn a process, as described in Section 23.3.2. We start by checking if there is enough free memory in the kernel's heap to store a new `Process` struct, and by computing $begin$ and end:

```
fn os_spawn(dst: &u32, name: &u32, name_length: u32, args: &u32, args_leng↩
                                                              ↩th: u32) -> u32 {
  let kernel = os_kernel();
  if sizeof(Process) > kernel.heap_limit - kernel.heap {
    return error_result(OUT_OF_MEMORY);
  }
  let begin = (((dst as u32 + 31) >> 5) << 5) as &u32;
  let end = (((kernel.stack_limit as u32) >> 5) << 5) as &u32;
  let parent = kernel.current_process;
  if parent != null {
    end = ((get_process_stack_pointer() as u32 >> 5) << 5) as &u32;
```

We then check the arguments, unless `parent` is `null` (this case corresponds to the kernel spawning the initial shell process, and we trust it). More precisely, we check that the $[begin, end[$ region is not empty, and that the parent's memory region contains it (as well as the name and arguments of the process to spawn). We also check that the arguments do not overlap the $[begin, end[$ region, so that we can copy them later on with `mem_copy_non_overlapping`.

```
    if end < begin || begin < parent.begin ||
        process_contains_buffer(parent, name, name_length) == FALSE ||
        process_contains_buffer(parent, args, args_length) == FALSE {
      return error_result(INVALID_ARGUMENT);
    }
```

```
  if args + args_length > begin && args < end {
    return error_result(INVALID_ARGUMENT);
  }
}
```

We continue by checking if the program file exists, and if there is enough memory to load it in the new process. If so we compute *heap* and *heap_limit*. Finally, we check if this heap region can contain the process arguments, as a data buffer (*i.e.*, with an additional 4 bytes header).

```
let file_block = disk_find_file(name, name_length, null);
if file_block == null { return error_result(NOT_FOUND); }
let code_size = disk_get_file_size(file_block);
if code_size > end - begin {
  return error_result(OUT_OF_MEMORY);
}
const MIN_STACK_SIZE: u32 = 512;
let heap = begin + code_size;
let heap_limit = end - MIN_STACK_SIZE;
if heap_limit < heap + 4 || args_length > heap_limit - heap - 4 {
  return error_result(OUT_OF_MEMORY);
}
```

At this point all the possible error cases have been checked (while being careful to avoid overflows). We can thus actually spawn the process. For this we start by loading the program's code at the *begin* address. This code starts after the header of the first file block, which contains 3 words and the file name (see Figure 21.4). We also copy the process arguments, as a data buffer, in the new process heap:

```
let offset = 12 + name_length;
disk_read_file(&file_block as &&DiskBlock, &offset, begin, code_size);
*heap = args_length;
mem_copy_non_overlapping(args, heap + 4, args_length);
```

Finally, we initialize an interrupt stack frame as described in Section 23.3.2 and Figure 23.5, save the current Process Stack Pointer, create a `Process` struct in the kernel's heap describing the spawned process, and make it the current process.

```
let context = (end - sizeof(Context)) as &Context;
context.r0 = heap as u32;
context.r1 = heap_limit as u32;
context.r14 = 0;
context.return_address = begin as u32;
context.status_register = 1 << 24;
if parent != null {
  parent.saved_context = get_process_stack_pointer() as &Context;
}
let process = kernel.heap as &Process;
kernel.heap = kernel.heap + sizeof(Process);
process.parent = parent;
```

```
  process.begin = begin;
  process.end = end;
  process.saved_context = context;
  os_set_current_process(kernel, process);
  return OK;
}
```

23.4.2 Exit

Implementing the `exit` function is much easier. We start by checking if the current `process` has a parent (the initial shell process must not exit). We also check if the result value is less than 2^{24} (see Section 23.3.4). If so we delete `process` from the kernel's heap, copy the exit value in the parent's saved R0 register, and make this parent the new current process.

```
fn os_exit(result: u32) -> u32 {
  let kernel = os_kernel();
  let process = kernel.current_process;
  if process.parent == null { return error_result(INVALID_STATE); }
  if result >= (1 << 24) { return error_result(INVALID_ARGUMENT); }
  kernel.heap = process as &u32;
  process = process.parent;
  process.saved_context.r0 = result;
  os_set_current_process(kernel, process);
  return OK;
}
```

In order to test the above system calls we need to spawn a process doing something that we can easily verify, such as drawing characters on the screen. For this we implement the following temporary function (see Section 10.2.2):

```
fn os_draw_char(c: u32) -> u32 {
  gpu_set_register(2, c);
  return OK;
}
```

23.4.3 Hard Fault and SVC handlers

As described in the requirements, an error occurring in a process should terminate it and return an error code to its parent. We thus implement the Hard Fault handler as follows (we assume that the kernel is bug free, *i.e.*, that no Hard Fault can occur while the kernel is running):

```
fn hard_fault_handler() {
  os_exit(INTERNAL_ERROR);
}
```

The SVC handler must call either `os_spawn`, `os_exit`, or `os_draw_char`, depending on the service ID in the interrupt stack frame. This could be done with a chain `if id == 0 ... else if id == 1 ... else if id == 2 ...` However, this would become less and less practical as new services are defined. Instead, we store the addresses of these functions in the `Kernel` struct, and we call the $\texttt{id}^{th}$ one with the following native function:

```
fn call(arg0: u32, arg1: u32, arg2: u32, arg3: u32, arg4: u32, arg5: u32,
    function: u32) -> u32 [
  /*BX_R6*/ 18224;
]
```

This calls a function f at interworking address `function` with up to 6 arguments. Indeed, when the BX instruction executes R0 to R5 contain the arguments, and the LR contains the return address in the caller. This is what f expects. We can thus jump to it directly with this instruction (R6 contains `function`). f will then return directly in the caller.

Before implementing the SVC handler it is useful to have some constants describing the total number of system calls and the number of parameters of each one. It is also useful to have a `struct` representing the system call arguments (*i.e.*, what the saved R1 register in the interrupt stack frame points to – see Figure 23.4):

```
const NUM_SYSTEM_CALLS: u32 = 3;
static SYSTEM_CALL_ARITY = [5, 1, 1];
```

```
struct CallArguments {
  arg0: u32,
  arg1: u32,
  arg2: u32,
  arg3: u32,
  arg4: u32,
  arg5: u32
}
```

With this we can finally implement the SVC handler. We start by computing the address of the top of the interrupt stack frame, in `context`. If there is a current process this is simply the value of the Process Stack Pointer (see Figure 23.4). Otherwise this is 4 bytes after the EXC_RETURN value (see Figure 23.6). And the EXC_RETURN value itself is 4 bytes after the `kernel` local variable, on top of the stack:

```
fn supervisor_call_handler() {
  let kernel = os_kernel();
  let exc_return_ptr = &kernel as &u32 + 4;
  let context = (exc_return_ptr + 4) as &Context;
  let process = kernel.current_process;
  if process != null {
    context = get_process_stack_pointer() as &Context;
  }
```

We then compute the address f of the function which can handle this system call, and the address of its arguments. For this we compute the address of the `kernel` field which contains f (the `spawn`, `exit`, or `draw_char` field, for ID=0, 1, or 2, respectively), and get the value at this address:

```
let id = context.r0;
let function = *(&kernel.spawn + (id << 2));
let args = context.r1 as &CallArguments;
```

We then call this function, whose interworking address is $f + 1$, after checking that the system call ID and the arguments are valid. In particular, the arguments must be contained in the process's memory region (if there is a current process):

```
let args_size = 0;
let result = error_result(INVALID_ARGUMENT);
if id < NUM_SYSTEM_CALLS {
  args_size = load8(SYSTEM_CALL_ARITY + id) << 2;
  if process == null ||
      process_contains_buffer(process, args as &u32, args_size) == TRUE {
    result = call(args.arg0, args.arg1, args.arg2, args.arg3,
        args.arg4, args.arg5, function + 1);
  }
}
```

Finally, we copy the result in the saved R0 register, so that it is popped in R0 on interrupt exit. We also set the EXC_RETURN value to $\text{FFFFFFFD}_{16} = 4294967293$, in order to force a return using the Process Stack (this is only necessary for the initial `spawn` system call):

```
  context.r0 = result;
  *exc_return_ptr = 4294967293; /*Thread Mode, Process Stack*/
}
```

23.4.4 Initialization

We update the `os_init` function to add the new Hard Fault and SVC handlers in the Vector Table, and to create and initialize the `Kernel` data structure (we also remove the temporary test code after `gpu_init`). For this we reserve 512 bytes for kernel's heap (enough for 30 processes), and 512 bytes for its stack:

```
fn os_init(code: &u32, heap: &u32, stack: &u32) {
  const VECTOR_TABLE_OFFSET_REGISTER: &&u32 = 3758157064;
  const HARD_FAULT_HANDLER_OFFSET: u32 = 12;
  const SVC_HANDLER_OFFSET: u32 = 44;
  const USART_HANDLER_OFFSET: u32 = 132;
  let vector_table = code - 257;
  *(vector_table + HARD_FAULT_HANDLER_OFFSET) = hard_fault_handler + 1;
  *(vector_table + SVC_HANDLER_OFFSET) = supervisor_call_handler + 1;
```

```
  *(vector_table + USART_HANDLER_OFFSET) = usart_handler + 1;
  *VECTOR_TABLE_OFFSET_REGISTER = vector_table;

  restore_bios_vector_table();

  const MAX_KERNEL_HEAP_SIZE: u32 = 512;
  const MAX_KERNEL_STACK_SIZE: u32 = 512;
  let kernel = heap as &Kernel;
  kernel.heap = heap + sizeof(Kernel);
  kernel.heap_limit = heap + MAX_KERNEL_HEAP_SIZE;
  kernel.stack_limit = stack - MAX_KERNEL_STACK_SIZE;
  kernel.current_process = null;
  kernel.spawn = os_spawn;
  kernel.exit = os_exit;
  kernel.draw_char = os_draw_char;
  *(KERNEL_POINTER_REGISTER as &&Kernel) = kernel;

  clock_init();
  keyboard_init();
  gpu_init();
}
```

We finish the implementation with the `os_spawn_shell` function, which spawns the program file named "`shell`", without any argument, after the kernel's heap. For this we use an SVC instruction in a native function, called from an intermediate `spawn` function:

```
fn system_call(id: u32, args: &u32) [ /*SVC*/ 57088; ]

fn spawn(dst: &u32, name: &u32, name_length: u32, args: &u32, args_length: u32) {
  system_call(0, &dst as &u32);
}

static SHELL = ['s','h','e','l','l'];
fn os_spawn_shell() {
  let dst = os_kernel().heap_limit;
  spawn(dst, SHELL, 5, null, 0);
}
```

When `spawn` starts it pushes its arguments on the stack, in the same order as required by the SVC handler (compare Figure 20.2 and Figure 23.4). Moreover, the `system_call` function stores the system call ID 0 and the address of the first argument in R0 and R1, respectively, before jumping to the SVC instruction (as required in Section 23.3.1). Finally, no other register are used at this point. In other words, everything is ready to make a system call. Note also that, since the initial process cannot exit, no instruction is needed to return from `system_call`.

23.5 Compilation and tests

Type "F8"+"r" in the command editor to load the current kernel source code, and "F4"+"r" to edit it. Then update it as described above. For reference, we also provide this new version in the `toys_v1.txt` file in https://ebruneton.github.io/toypc/sources.zip. When done, type "F10"+"r" to save it and "F9"+"r" to compile it. If necessary, repeat these steps until the compilation is successful. To copy the compiled code in the "`toys`" file use F3 to load the builder source code, and F4 to edit it. Then change its main function to the following:

```
static NAME = ['t','o','y','s'];
fn main() -> u32 {
  const COMPILED_CODE: &u32 = 929792;
  return buffer_write(COMPILED_CODE, NAME, 4);
}
```

Finally, save the builder with F5, compile it with F6, and run it with F7.

To test this kernel we need a "shell" program. We use here a small test application spawning itself recursively to compute a factorial. As in the previous chapter, we store it in the Flash1 memory bank before copying it in the file system. For this we use a new data buffer, at address $DD000_{16}$ = 905216 = page 464 (see Figure 23.7). To load and save its content, edit the F11 command as follows (see F3 and F8):

```
fn 0
  cst 905216 cst 537330176 call 2708
  cst_0 retv
d LOAD_APPLICATION_SOURCE_CODE
```

and define the F12 command with (see F5 and F10):

```
fn 0
  cst 537330176 cst 464 call 2836
  cst_0 retv
d SAVE_APPLICATION_SOURCE_CODE
```

Then use F2 to start a new program, F4 to edit it, and type the following code:

```
fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32;
fn exit(result: u32) -> u32;

fn entry(heap: &u32, heap_limit: &u32) {
  let args = heap + 4;
  let args_end = args + *heap;
  heap = (((args_end as u32 + 3) >> 2) << 2) as &u32;
  exit(main(args, args_end, heap, heap_limit));
}
fn system_call(id: u32, args: &u32) -> u32 [
  /*SVC*/ 57088;
  /*MOV_PC_LR*/ 18167;
]
```

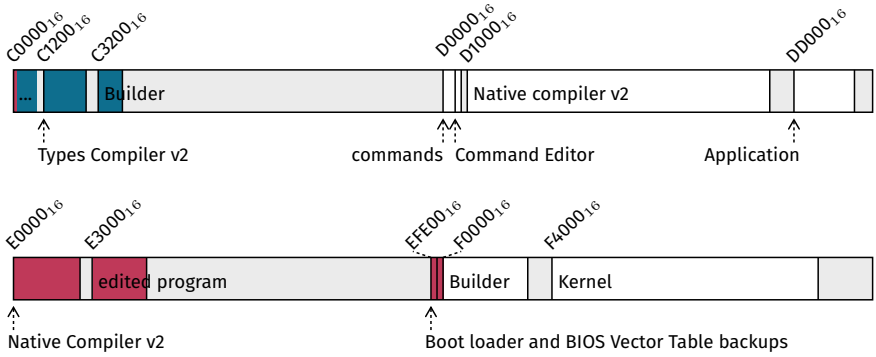


FIGURE 23.7 The Flash1 memory bank content at the end of Chapter 23. White, blue, red and gray areas represent source code, bytecode, native code, and unused memory, respectively (not to scale).

```
fn spawn(dst: &u32, name: &u32, name_length: u32, args: &u32, args_length:⌇
                                                              ⌇ u32) -> u32 {
  return system_call(0, &dst as &u32);
}
fn exit(result: u32) -> u32 {
  return system_call(1, &result);
}
fn draw_char(c: u32) -> u32 {
  return system_call(2, &c);
}

static SHELL = ['s','h','e','l','l'];
fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  if args_end == args {
    *heap = 3;
    draw_char('=');
    draw_char('0' + spawn(heap + 4, SHELL, 5, heap, 4));
    loop {}
  } else {
    if *args == 0 { return 1; }
    *args = *args - 1;
    return spawn(heap, SHELL, 5, args, 4) * (*args + 1);
  }
}
```

The `entry` function computes the start and end pointers of the arguments, and adjusts `heap` to the first multiple of 4 after `args_end` (to avoid unaligned memory accesses, which might decrease performance – this is not necessary here but we reuse this code later on). It then calls a `main` function with these new arguments and `exit` its result.

The spawn function is similar to the one in the kernel, but now returns a result (the exit value of the spawned process). The exit and draw_char functions are similar, for the corresponding system calls. The system_call native function is also extended with a "return" instruction because these 3 system calls are now expected to return.

Finally, the main function implements our factorial test. If the arguments are empty, which is the case when this application is spawned by the kernel, if spawns itself with the argument 3, at the destination address heap + 4. It then draws a digit corresponding to the result of the child process, and finally loops forever. Otherwise, if the arguments are not empty, the main function computes the factorial of the first argument with a recursive spawn.

When you are done typing this program, use F12 to save it, and F9 to compile it. If necessary, repeat these steps until the compilation is successful. To copy the compiled code in the "shell" file and to test it use F3 to load the builder source code, and F4 to edit it. Then change its main function to the following:

```
static NAME = ['s','h','e','l','l'];
fn main() -> u32 {
  const COMPILED_CODE: &u32 = 929792;
  let result = buffer_write(COMPILED_CODE, NAME, 5);
  if result == OK { flash_boot_loader_and_reset(); }
  return result;
}
```

Finally, save the builder with F5, compile it with F6, and run it with F7. If all goes well this should launch the kernel, which should launch our test program, which should then spawn itself recursively and finally display "=6", the factorial of 3. After that you can reset the Arduino, which should restart with the memory editor.

CHAPTER

24 Streams

Our kernel can now spawn an initial process, which can itself spawn other processes. However, these processes can't use the keyboard, the screen, or the disk unless they re-implement the related kernel functions. Indeed, for the reasons explained in the previous chapter, they can't call these functions directly. This chapter extends our kernel with new system calls in order to solve this problem.

24.1 Requirements

The new system calls should provide access to all existing kernel services. This includes the delay and boot mode selection services, getting the last character typed or waiting for one, displaying text on screen, and listing, creating, reading, writing, and deleting files. We also require a new service: the possibility for a process to return more data to its parent than a single exit value (other than via files).

These system calls should encapsulate the kernel's implementation details, so that it can be improved without having to rewrite user programs, also called applications. For instance, users must not have to deal with `DiskBlock` or `FileBlock`, so that these data structures can be replaced with better ones later on.

The new system calls must also provide a safe way to use the computer resources, and the disk in particular. This is not the case of the current file system functions. For instance, it is possible to inadvertently pass a free block to a function expecting a file block, which might cause a crash or corrupt the file system. This could happen, for example, if a process gets the first block of a file, spawns a child process which happens to delete the file, and then uses this block to read the file after the child terminates.

24.2 Design

In order to meet the above requirements we use a design inspired from the one used in many existing operating systems for this purpose. More precisely, we model the keyboard, the graphics card, and the files as sources and/or sinks of bytes, transmitted one after the other:

- the keyboard is a source of bytes, one per character typed.

- the graphics card is a sink of commands, sent to it to write values in its registers, and a source of values, read from its registers (see Section 10.2.3).
- a file is a source of bytes when it is read, and a sink of bytes when new data is appended to it.

These sources and sinks are called *input streams* and *output streams*, respectively. We introduce two new system calls, called `read` and `write`, to read bytes from an input stream, and to write bytes in an output stream, respectively. We then define the following streams, identified with a *stream ID*:

- STANDARD_INPUT (ID 0): the stream of characters typed on the keyboard. Reading a byte from this stream returns the last character typed since the last one read. If there is none this operation *blocks*, *i.e.*, waits until a character is typed.
- STANDARD_OUTPUT (ID 1): a stream of bytes sent by a process to its parent. These bytes are appended to a buffer provided by the parent. They allow processes to return more data to their parent than a single exit value.
- KEYBOARD (ID 2): a *non-blocking* stream of the characters typed on the keyboard. Reading a byte from this stream returns the last character typed (since the last one read from STANDARD_INPUT or KEYBOARD), or 0 if there is none.
- GPU (ID 3): an input and output stream used to send commands to the graphics card, and to read the value of its registers.

File streams can not have a fixed stream ID, since files can be created and deleted at any time. Instead, we define two more system calls to create a stream for a given file, and to delete a file stream. They are called `open` and `close`, respectively:

- `open` takes as parameter a file name and a *mode* (read or write). It creates either an input stream or an output stream, depending on the mode, and returns the ID of this new stream. Creating an output stream either creates a file if there is no file with the given name, or *clears* the existing file content.
- `close` takes as parameter a file stream ID and deletes this stream.

Several input streams can exist at a given time for a single file. Each input stream has its own *cursor*, which indicates how many bytes have already been read from the file. Thus, for instance, one stream could have already read 10 bytes of a file, while another could have read 20 bytes of the same file.

To simplify, and to ensure safety at the same time, we do not allow the coexistence of several output streams for the same file (across all processes). We do not allow either the coexistence of input and output streams for a given file. Similarly, we forbid the deletion of a file (for which we add a `delete` system call) which has some associated input or output streams (in any process).

24.3 Data structures and algorithms

24.3.1 STANDARD_INPUT

Reading a byte from standard input is easy to do with the `keyboard_wait_char` function. Note however that this function must be called from the SVC handler in order to use it in the `read` system call. In other words, it must run as part of the SVC interrupt handler. But it might itself wait for the USART handler to run, to get characters from the keyboard. For this to work, an USART interrupt must be able to interrupt the SVC handler. Otherwise the latter would wait forever. However, by default, an interrupt handler cannot interrupt another.

To handle such cases the ARM Cortex M3 uses *interrupt priorities*. Interrupts with higher priority are executed first, and can interrupt a currently running handler with a lower priority. Each type of interrupt has a configurable priority, specified with a so-called *priority level* (a non-negative integer). By default, all interrupt types have priority level 0, which corresponds to the highest priority[1]. Increasing the priority level of an interrupt decreases its priority.

To solve the above problem we thus need to increase the priority level of SVC interrupts, *i.e.*, to decrease their priority (we cannot increase the USART priority since it is already the highest possible). This can be done with the SVC Handler Priority Register, at address E000ED1C_{16} in the "System" region (see Figure 6.3). The most significant byte of this register defines the priority level of SVC interrupts (the other bytes are "reserved").

24.3.2 STANDARD_OUTPUT

In order to write data to standard output, on behalf of a process, the kernel must have a pointer to an output buffer provided by the parent, as well as its capacity. To this end:

- we add a new `output_p` parameter to the `spawn` system call. This parameter must be a pointer to a word containing the buffer's start address (see Figure 24.1).
- we use the existing spawn's `dst` parameter as the buffer's limit address.

Each time the kernel appends a byte to this buffer, it also increases the address at `output_p`. This allows a parent process to compute how many bytes were written by its child, once it terminates (see Figure 24.1).

24.3.3 GPU

The GPU stream is used to read and write the value of the graphics card registers. Recall that this is done with 3 types of commands, each using 2 bytes: Select Register, Read Data, and Write Data (Section 10.2.3). We thus use the following algorithms:

- reading a byte from GPU sends a Read Data command to the graphics card and returns the result. This gives the value of the last register selected with a Select Register command.

[1] Reset and Hard Fault have even higher, non-configurable priorities.

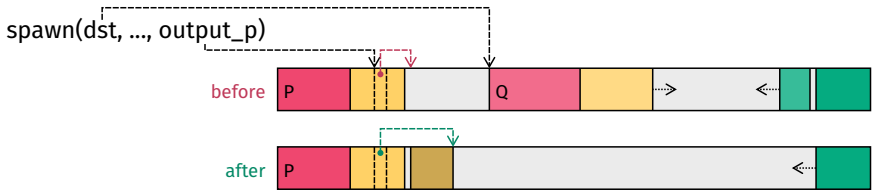


FIGURE 24.1 A child process writes STANDARD_OUTPUT data at the address stored in the word at `output_p`, up to `dst`, and updates this address (top). When it terminates (bottom), the word at `output_p` points to the end of the data it wrote (brown). Red, yellow, green, and gray areas represent native code, heap, stack, and unused memory, respectively.

- writing two bytes in GPU sends the corresponding command to the graphics card. This must be either a Select Register or a Write Data command. Some registers are sensitive and might damage the card if not used correctly, such as those setting the clock frequency. For safety, we filter out the Select Register commands for these registers.

24.3.4 File streams

A process can create several file streams with open. Each has its own ID, mode (read or write), and cursor. To keep track of this information we add a `file_streams` field in the `Process` struct, pointing to a linked list of `FileStream` data structures. Each `FileStream` contains the following data (see Figure 24.2):

- a pointer to the next `FileStream` in the list, or `null` for the last one.
- the stream's mode. We use 'r' for the read mode, and 'w' for the write mode.
- a pointer to the `FileBlock` of the file from which this stream was created.
- the stream's cursor, pointing to the next byte to read from or to write to this file. It is represented with a pointer to a `DiskBlock`, and an offset within this block:
  - for an input stream, the offset is the number of bytes that have already been read in this block, plus the block's header size.
  - for an output stream the offset is unused. Instead, bytes are appended after the last used one in the block, which is specified in the block's header.

Note that a `FileStream` does not store its ID. Instead, we use its position in the `file_streams` list to compute its ID: the i^{th} element, counting from 0, has ID $i + 4$[2]. A consequence is that, when a file stream is closed, it cannot be deleted and removed from the `file_streams` list (this would change the IDs of the next streams). Instead, we set its mode to 0, meaning "unused", and its `FileBlock` pointer to `null`.

We store the newly created `FileStream` data structures after the current `Process`, in the kernel's heap. They are automatically deleted when the process terminates.

[2]Stream IDs are local: two streams with the same ID, in different processes, can refer to different files.

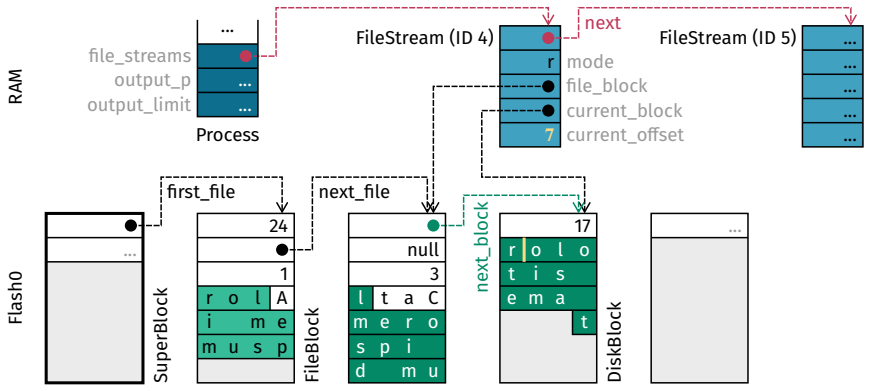


FIGURE 24.2 A process has a linked list of structs representing its file streams (red links). Each stream points to the corresponding file block (bottom) and to the next byte to read or write. Here the first stream, with implicit ID 4, has read the first block and $7 - 4 = 3$ bytes of the second block of the "Cat" file.

This is because a `Process` is deleted, in `os_exit`, by restoring the kernel's heap pointer to the value it had just before the process was created.

Files list

In order to get a list of all the files we use a virtual *directory* containing this list of names. By convention, users can open it in read mode by using an empty file name. This gives a stream from which file names can be read one by one (in their entirety; to simplify, we do not allow reading a file name piece by piece).

To implement this we represent a directory stream with a `FileStream` whose mode is 'l' (for "list"). The `FileBlock` pointer of such a stream points to the file whose name must be returned by the next `read` call. Its cursor is unused (`DiskBlock` pointer and offset).

Although we forbid deleting a file with at least one open stream, we do not forbid deleting one which is the next to be returned by a directory stream. Instead, if a directory stream points to a file which is about to be deleted, we advance it to the next file.

24.4 Implementation

We can now extend the second version of our kernel, written in the previous chapter, with the algorithms described above. The first change is the addition of the `load16` function, copied from our native compiler. We use it later on to read commands from the GPU stream:

```
fn load8(ptr: &u32) -> u32 [ /*LDRB_R0_R0_0*/30720; /*MOV_PC_LR*/18167; ]
fn load16(ptr: &u32) -> u32 [ /*LDRH_R0_R0_0*/34816; /*MOV_PC_LR*/18167; ]
```

The following existing code is unchanged, up to and including the `Context` struct. We implement the `FileStream` struct just after it, as described above. We also add new fields in `Process` to store its list of file streams, as well as its `output_p` parameter. The `output_limit` field indicates the limit of the standard output buffer.

```
const MODE_UNUSED: u32 = 0;
const MODE_LIST: u32 = 'l';
const MODE_READ: u32 = 'r';
const MODE_WRITE: u32 = 'w';

struct FileStream {
  next: &FileStream,
  mode: u32,
  file_block: &FileBlock,
  current_block: &DiskBlock,
  current_offset: u32
}
struct Process {
  parent: &Process,
  begin: &u32,
  end: &u32,
  saved_context: &Context,
  file_streams: &FileStream,
  output_p: &&u32,
  output_limit: &u32
}
```

We continue by removing the temporary `draw_char` field in `Kernel`, and by replacing it with fields for the new system calls, implemented below. We also add a small utility function computing the minimum of two numbers, needed later on.

```
  spawn: u32,
  exit: u32,
  sleep: u32,
  stat: u32,
  open: u32,
  read: u32,
  write: u32,
  close: u32,
  delete: u32,
  reboot: u32
}

const KERNEL_POINTER_REGISTER: &&Kernel = 1074666140;
fn os_kernel() -> &Kernel { return *KERNEL_POINTER_REGISTER; }

fn min(x: u32, y: u32) -> u32 {
  if x < y { return x; } else { return y; }
}
```

The os_spawn function must be updated to take the new output_p parameter into account. This requires additional validations, to make sure that the 4 bytes starting at output_p are contained in the parent's memory region. We must also check that the address stored at output_p is in this region, and before dst. After that we can simply store output_p and the output buffer limit address (dst) in the process struct.

```
fn os_spawn(dst: &u32, name: &u32, name_length: u32,
    args: &u32, args_length: u32, output_p: &&u32) -> u32 {
...
  if parent != null {
...
    if args + args_length > begin && args < end {
      return error_result(INVALID_ARGUMENT);
    }
    if process_contains_buffer(parent, output_p as &u32, 4) == FALSE ||
        *output_p < parent.begin || *output_p > dst {
      return error_result(INVALID_ARGUMENT);
    }
  }
...
  process.saved_context = context;
  process.file_streams = null;
  process.output_p = output_p;
  process.output_limit = dst;
  os_set_current_process(kernel, process);
  return OK;
}
```

We keep the os_exit function unchanged since it already automatically deletes the process's file streams. On the other hand we delete the temporary os_draw_char function, and start implementing functions for the new system calls. The first one provides access to the delay function. Since this function is restricted to delays of about $1.5s$ (see Section 9.5) we call it several times if needed:

```
fn os_sleep(millis: u32) -> u32 {
  while millis > 1000 { delay(1000); millis = millis - 1000; }
  delay(millis);
  return OK;
}
```

With the stream system calls presented above it is not possible to get the size of a file other than by reading it completely. To improve this we provide the following system call, which returns the size of the specified file. If the given name is not contained in the process's memory region, or if the file is not found, it returns an error instead (in the most significant byte, see Section 23.3.4):

```
fn os_stat(name: &u32, length: u32) -> u32 {
  let process = os_kernel().current_process;
  if process_contains_buffer(process, name, length) == FALSE {
```

```
    return error_result(INVALID_ARGUMENT);
  }
  let file_block = disk_find_file(name, length, null);
  if file_block == null { return error_result(NOT_FOUND); }
  return disk_get_file_size(file_block);
}
```

Before implementing the stream system calls it is useful to have some functions to manage the file streams of a process. The following one returns the `FileStream` with the given ID, or `null` if there is none:

```
const STANDARD_INPUT: u32 = 0;
const STANDARD_OUTPUT: u32 = 1;
const KEYBOARD: u32 = 2;
const GPU: u32 = 3;
const FIRST_FILE_STREAM_ID: u32 = 4;
```

```
fn process_get_file_stream(self: &Process, stream_id: u32) -> &FileStream ≀
                                                                        ≀{
  let stream = self.file_streams;
  let id = FIRST_FILE_STREAM_ID;
  while stream != null && id != stream_id {
    stream = stream.next;
    id = id + 1;
  }
  return stream;
}
```

The next function returns the first unused `FileStream` of a process, or creates one if all streams are used. In the latter case, it adds the `FileStream` at the end of the `file_streams` list, in order to avoid changing the implicit ID of the existing ones. This requires updating the `next` field of the last existing `FileStream`, or the process's `file_streams` field if there is no existing stream. The `stream_ptr` variable points to the former or to the later, depending on which case applies.

This function stores the ID of the returned stream at address `stream_id`. It returns `null` if there is not enough memory in the kernel's heap to create a `FileStream`.

```
fn process_get_unused_file_stream(self: &Process, stream_id: &u32) -> &Fil≀
                                                                    ≀eStream {
  let stream_ptr = &self.file_streams;
  let stream = *stream_ptr;
  *stream_id = FIRST_FILE_STREAM_ID;
  while stream != null && stream.mode != MODE_UNUSED {
    stream_ptr = &stream.next;
    stream = *stream_ptr;
    *stream_id = *stream_id + 1;
  }
  let kernel = os_kernel();
  if stream == null && kernel.heap + sizeof(FileStream) <= kernel.heap_lim≀
```

```
                                                                    ⌇it {
    stream = kernel.heap as &FileStream;
    kernel.heap = kernel.heap + sizeof(FileStream);
    stream.next = null;
    stream.mode = MODE_UNUSED;
    stream.file_block = null;
    *stream_ptr = stream;
  }
  return stream;
}
```

Another convenient function is the following, which checks if a given file can be opened in read or write mode. For this it checks all the file streams of all the processes. If one of them corresponds to `file_block`, in read or write mode, then the file can only be opened again in read mode, provided it is not already open in write mode.

```
fn process_can_open(self: &Process, file_block: &FileBlock, mode: u32) -> ⌇
                                                                    ⌇u32 {
  let process = self;
  let stream: &FileStream = null;
  while process != null {
    stream = process.file_streams;
    while stream != null {
      if stream.file_block == file_block && stream.mode != MODE_LIST {
        if stream.mode == MODE_WRITE || mode == MODE_WRITE {
          return FALSE;
        }
      }
      stream = stream.next;
    }
    process = process.parent;
  }
  return TRUE;
}
```

We can now implement the stream system call functions themselves, starting with open. This function takes a file name and a mode as parameters, and returns a stream ID or an error. It first validates its arguments, and gets or creates an unused `FileStream` and its ID. An empty file name designates the virtual directory, which can only be read (with the internal "list" mode). Otherwise, in read mode, the file must exists. In write mode it is created if it does not exist yet. It is cleared otherwise.

```
fn os_open(name: &u32, length: u32, mode: u32) -> u32 {
  let process = os_kernel().current_process;
  if process_contains_buffer(process, name, length) == FALSE ||
      mode != MODE_READ && mode != MODE_WRITE {
    return error_result(INVALID_ARGUMENT);
  }
  let stream_id = 0;
```

```
  let stream = process_get_unused_file_stream(process, &stream_id);
  if stream == null { return error_result(OUT_OF_MEMORY); }
  let file_block = SUPER_BLOCK.first_file;
  let status = OK;
  if length == 0 {
    if mode != MODE_READ { status = INVALID_ARGUMENT; }
    mode = MODE_LIST;
  } else {
    file_block = disk_find_file(name, length, null);
    if file_block == null {
      if mode == MODE_READ {
        status = NOT_FOUND;
      } else {
        status = disk_create_file(name, length, &file_block);
      }
    } else if process_can_open(process, file_block, mode) == TRUE {
      if mode == MODE_READ {
        stream.current_offset = 12 + file_block.name_length;
      } else {
        disk_clear_file(file_block);
      }
    } else {
      status = INVALID_STATE;
    }
  }
  if status != OK { return error_result(status); }
  stream.mode = mode;
  stream.file_block = file_block;
  stream.current_block = file_block as &DiskBlock;
  return stream_id;
}
```

The next function reads up to `size` bytes from the given stream, into `buffer`. It returns the number of bytes actually read, or an error. For this it first checks its arguments. Then, if the stream is STANDARD_INPUT or KEYBOARD, it reads only one character, in a blocking or non-blocking way. If the stream is GPU it reads exactly `size` bytes by sending as many Read Data commands. If it is STANDARD_OUTPUT this is an error. Any other case corresponds to a file stream or a directory stream. In the latter case, this function copies up to `size` bytes of the current file name, and advances the stream to the next file. To avoid getting truncated file names, users must pass a buffer which can contain the longest possible file name, *i.e.*, 244 characters.

```
fn os_read(stream_id: u32, buffer: &u32, size: u32) -> u32 {
  let process = os_kernel().current_process;
  if size == 0 || process_contains_buffer(process, buffer, size) == FALSE ≀
                                                                      ≀{
    return error_result(INVALID_ARGUMENT);
  }
```

```
  let i = 0;
  if stream_id < FIRST_FILE_STREAM_ID {
    if stream_id == STANDARD_INPUT {
      *buffer = keyboard_wait_char();
      return 1;
    } else if stream_id == KEYBOARD {
      *buffer = keyboard_get_char();
      return 1;
    } else if stream_id == GPU {
      while i < size {
        store8(buffer + i, spi_transfer(16384 /*Read Data Command*/));
        i = i + 1;
      }
      return size;
    } else {
      return error_result(INVALID_STATE);
    }
  }
  let stream = process_get_file_stream(process, stream_id);
  if stream == null || stream.mode == MODE_UNUSED {
    return error_result(NOT_FOUND);
  }
  if stream.mode == MODE_LIST {
    if stream.file_block == null { return 0; }
    size = min(size, stream.file_block.name_length);
    mem_copy_non_overlapping(&stream.file_block.name, buffer, size);
    stream.file_block = stream.file_block.next_file;
    return size;
  }
  if stream.mode != MODE_READ { return error_result(INVALID_STATE); }
  return size - disk_read_file(&stream.current_block, &stream.current_offs⌇
                                                              ⌇et, buffer, size);
}
```

Before implementing a function for the `write` system call we need one to validate a command before sending it to the GPU. This helper function must filter out invalid commands (only Select Register and Write Data can be used) as well as Select Register commands for "sensitive" registers. For this we define a list of "safe" registers:

```
static GPU_SAFE_REGISTERS = [
  4, 0, 1, 0, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 0, 0,
  255, 224, 255, 255, 255, 31, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
```

These 32 bytes contain 256 bits, one per possible GPU register. The i^{th} bit, counting from 0, is 1 if register Ri is judged "safe", or 0 otherwise (such as registers setting clock frequencies – see the full register list in [13]). The following function uses it to filter out invalid GPU commands as described above (the i^{th} bit is the j^{th} bit of the k^{th} byte, where $j = i \bmod 8 = i \wedge 7$ and $k = i/8 = i \gg 3$):

```
fn os_gpu_write(value: u32) {
  let command = value >> 8;
  let register = value & 255;
  let mask = 0;
  if command != 0 /*Write Data Command*/ {
    if command != 128 /*Select Register Command*/ { return; }
    mask = 1 << (register & 7);
    if *(GPU_SAFE_REGISTERS + (register >> 3)) & mask == 0 { return; }
  }
  spi_transfer(value);
}
```

The next function writes up to `size` bytes from the given `buffer`, into the given stream. It returns the number of bytes actually written, or an error. If the arguments are valid, and if the stream is STANDARD_OUTPUT, it copies as many bytes as possible into the process's output buffer (given its remaining capacity). If the stream is GPU, it sends these bytes to the graphics card two by two, with the above function (if `size` is odd the last byte is not sent). Otherwise the stream must be a file stream in write mode. In this case this function appends the whole buffer to the corresponding file (this can fail if the disk is full).

```
fn os_write(stream_id: u32, buffer: &u32, size: u32) -> u32 {
  let process = os_kernel().current_process;
  if process_contains_buffer(process, buffer, size) == FALSE {
    return error_result(INVALID_ARGUMENT);
  }
  let i = 0;
  if stream_id < FIRST_FILE_STREAM_ID {
    if stream_id == STANDARD_OUTPUT {
      size = min(size, process.output_limit - *process.output_p);
      mem_copy_non_overlapping(buffer, *process.output_p, size);
      *process.output_p = *process.output_p + size;
      return size;
    } else if stream_id == GPU {
      while i + 1 < size {
        os_gpu_write(load16(buffer + i));
        i = i + 2;
      }
      return i;
    } else {
      return error_result(INVALID_STATE);
    }
  }
  let stream = process_get_file_stream(process, stream_id);
  if stream == null || stream.mode == MODE_UNUSED {
    return error_result(NOT_FOUND);
  }
  if stream.mode != MODE_WRITE { return error_result(INVALID_STATE); }
```

```
  let status = disk_write_file(&stream.current_block, buffer, size);
  if status != OK { return error_result(status); }
  return size;
}
```

Closing a file stream is very easy (see Section 24.3.4):

```
fn os_close(stream_id: u32) -> u32 {
  let process = os_kernel().current_process;
  let stream = process_get_file_stream(process, stream_id);
  if stream == null || stream.mode == MODE_UNUSED {
    return error_result(NOT_FOUND);
  }
  stream.mode = MODE_UNUSED;
  stream.file_block = null;
  return OK;
}
```

The following function deletes the file with the given name. Before this it checks that its arguments are valid, that the file exists and that no process is currently reading it or writing to it (by checking that it could be open in write mode). It also advances all the directory streams which reference it to the next file (by iterating over all file streams of all processes):

```
fn os_delete(name: &u32, length: u32) -> u32 {
  let process = os_kernel().current_process;
  if process_contains_buffer(process, name, length) == FALSE {
    return error_result(INVALID_ARGUMENT);
  }
  let previous_file_block: &FileBlock = null;
  let file_block = disk_find_file(name, length, &previous_file_block);
  if file_block == null { return error_result(NOT_FOUND); }
  if process_can_open(process, file_block, MODE_WRITE) == FALSE {
    return error_result(INVALID_STATE);
  }
  let stream: &FileStream = null;
  while process != null {
    stream = process.file_streams;
    while stream != null {
      if stream.mode == MODE_LIST && stream.file_block == file_block {
        stream.file_block = file_block.next_file;
      }
      stream = stream.next;
    }
    process = process.parent;
  }
  disk_delete_file(file_block, previous_file_block);
  return OK;
}
```

The last new system call function reboots the computer and launches the operating system (if `mode` is not 1) or the Boot Assistant (if `mode` is 1). For this it changes the boot mode to boot from ROM if necessary, and then triggers a reset with the Reset Control Register, at address $400E1A00_{16}$ = 1074665984 (see Section 9.7):

```
fn os_reboot(mode: u32) -> u32 {
  const RESET_CONTROL_REGISTER: &u32 = 1074665984;
  const RESET_COMMAND: u32 = 2768240653;
  if mode == 1 { boot_mode_select_rom(); }
  *RESET_CONTROL_REGISTER = RESET_COMMAND;
  return OK;
}
```

After that the existing kernel code is mostly unchanged. The SVC handler, in particular, does not need to be updated to take the new system call functions into account. Instead, we just need to update the constants describing the number of system calls and the arity of each one (in the order they are declared in `Kernel`):

```
const NUM_SYSTEM_CALLS: u32 = 10;
static SYSTEM_CALL_ARITY = [6, 1, 1, 2, 3, 3, 3, 1, 2, 1];
```

The `os_init` function just needs to be updated in order to store the new system call function addresses in the `kernel` struct, and to configure the SVC handler priority level. Here we use the maximum priority level (255), *i.e.*, the lowest possible priority.

```
fn os_init(code: &u32, heap: &u32, stack: &u32) {
...
  kernel.exit = os_exit;
  kernel.sleep = os_sleep;
  kernel.stat = os_stat;
  kernel.open = os_open;
  kernel.read = os_read;
  kernel.write = os_write;
  kernel.close = os_close;
  kernel.delete = os_delete;
  kernel.reboot = os_reboot;
  *(KERNEL_POINTER_REGISTER as &&Kernel) = kernel;

  clock_init();
  keyboard_init();
  gpu_init();
  const SVC_HANDLER_PRIORITY_REGISTER: &u32 = 3758157084;
  *SVC_HANDLER_PRIORITY_REGISTER = 255 << 24;
}
```

The last required change is to take the new `output_p` parameter into account to spawn the shell. Here we set its output buffer capacity to 0.

```
fn spawn(dst: &u32, name: &u32, name_length: u32,
    args: &u32, args_length: u32, output_p: &&u32) {
```

```
  system_call(0, &dst as &u32);
}

static SHELL = ['s','h','e','l','l'];
fn os_spawn_shell() {
  let dst = os_kernel().heap_limit;
  spawn(dst, SHELL, 5, null, 0, &dst);
}
```

24.5 Compilation and tests

Type "F8"+"r" in the command editor to load the current kernel source code, and "F4"+"r" to edit it. Then update it as described above. For reference, we also provide this new version in the `toys_v2.txt` file in https://ebruneton.github.io/toypc/sources.zip. When done, type "F10"+"r" to save it and "F9"+"r" to compile it. If necessary, repeat these steps until the compilation is successful. To copy the compiled code in the "`toys`" file use F3 to load the builder source code, and F4 to edit it. Then change its main function to the following:

```
static NAME = ['t','o','y','s'];
fn main() -> u32 {
  const COMPILED_CODE: &u32 = 929792;
  return buffer_write(COMPILED_CODE, NAME, 4);
}
```

Finally, save the builder with F5, compile it with F6, and run it with F7. To test the new system calls we can use two processes doing the following:

- a parent creates a file and writes some text to it. It then spawns a child, displays on screen the output written to standard output by its child, and then any character received from standard input.
- the child reads the above file and copies its content to standard output.

Each step should work and the screen should eventually display the text written to disk by the parent. We can implement this test in a single program spawning itself, playing the parent role if spawned without argument, or the child role otherwise. For this use F11 to load the current "shell" source code, and F4 to edit it. Keep the `entry` and `system_call` functions unchanged. Add the following `OK` constant and `status` helper before the `spawn` function, and add an `output_p` parameter to the latter:

```
const OK: u32 = 0;
fn status(result: u32) -> u32 { return result >> 24; }

fn spawn(dst: &u32, name: &u32, name_length: u32,
    args: &u32, args_length: u32, output_p: &&u32) -> u32 {
  return system_call(0, &dst as &u32);
}
```

Then, after `exit`, unchanged, add similar functions for the new system calls:

```
fn exit(result: u32) -> u32 {
  return system_call(1, &result);
}
fn sleep(millis: u32) -> u32 {
  return system_call(2, &millis);
}
fn stat(name: &u32, length: u32) -> u32 {
  return system_call(3, &name as &u32);
}
fn open(name: &u32, length: u32, mode: u32) -> u32 {
  return system_call(4, &name as &u32);
}
const STANDARD_INPUT: u32 = 0;
const STANDARD_OUTPUT: u32 = 1;
const KEYBOARD: u32 = 2;
const GPU: u32 = 3;
fn read(stream_id: u32, buffer: &u32, size: u32) -> u32 {
  return system_call(5, &stream_id);
}
fn write(stream_id: u32, buffer: &u32, size: u32) -> u32 {
  return system_call(6, &stream_id);
}
fn close(stream_id: u32) -> u32 {
  return system_call(7, &stream_id);
}
fn delete(name: &u32, length: u32) -> u32 {
  return system_call(8, &name as &u32);
}
fn reboot(mode: u32) -> u32 {
  return system_call(9, &mode);
}
```

Using these, add the following function to read a character from STANDARD_INPUT, and update `draw_char` to use the new `write` system call. This new implementation writes the following commands to the GPU stream, in this order (see Sections 10.2.2 and 10.2.3):

- a Select Register command (80_{16}), in the 2 least significant bytes of `buffer`.
- a Write Data command (00_{16}), in the most significant bytes.

```
fn wait_char() -> u32 {
  let buffer = 0;
  read(STANDARD_INPUT, &buffer, 1);
  return buffer;
}
```

```
fn draw_char(c: u32) -> u32 {
  let buffer = (c & 255) << 16 | (32768 /*Select Register*/ | 2);
  return write(GPU, &buffer, 4);
}
```

Then implement a function playing the parent process role:

```
static SHELL = ['s','h','e','l','l'];
static TEST = ['t','e','s','t'];
static TEXT = ['0','1','2','3','4','5','6','7','8','9'];

fn parent(heap: &u32) -> u32 {
  let stream = open(TEST, 4, 'w');
  if status(stream) != OK { return 'o'; }
  let i = 0;
  while i < 23 {
    if write(stream, TEXT, 10) != 10 { return 'w'; }
    i = i + 1;
  }
  if close(stream) != OK { return 'c'; }
  let argument = 0;
  let output = heap;
  if spawn(heap + 512, SHELL, 5, &argument, 4, &output) != OK {
    return 's';
  }
  if delete(TEST, 4) != OK { return 'd'; }
  let ptr = heap;
  while ptr < output {
    draw_char((*ptr) & 255);
    ptr = ptr + 1;
  }
  return '.';
}
```

This function writes `0123456789` 23 times in a new file named "`test`", and closes it. It then spawns the "`shell`" program with a 4 bytes argument equal to 0, and an output buffer for this child starting at `heap`, with a maximum capacity of 512 bytes. Finally, it deletes the file and displays the content of this output buffer on screen. It returns '.' if everything goes well, or a lowercase letter indicating which step failed. Continue with a function playing the child process role:

```
fn child(heap: &u32) -> u32 {
  let stream = open(TEST, 4, 'r');
  if status(stream) != OK { return 'O'; }
  let n = 0;
  loop {
    n = read(stream, heap, 64);
    if status(n) != OK { return 'R'; }
    write(STANDARD_OUTPUT, heap, n);
```

```
    if n < 64 { break; }
  }
  exit(OK);
  return 'E';
}
```

This function reads the "test" file, in chunks of at most 64 bytes, and writes each chunk to standard output just after it has been read. If less than 64 bytes are read this means that the end of the file is reached. In this case the function exits with OK (without closing the stream; this is not necessary since terminating a process deletes all its streams). Otherwise, or if exit fails, it returns an uppercase letter indicating which step failed.

Finally, update the main function to call the parent or the child function, and to finally display any character read from standard input:

```
fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  if args_end == args {
    draw_char(parent(heap));
  } else {
    draw_char(child(heap));
  }
  loop { draw_char(wait_char()); }
}
```

When you are done typing this program, use F12 to save it, and F9 to compile it (for reference we also provide it in the streams_test.txt file in https://ebruneton.github.io/toypc/sources.zip). If necessary, repeat these steps until the compilation is successful. To copy the compiled code in the "shell" file and to test it use F3 to load the builder source code, and F4 to change its main function as follows:

```
static NAME = ['s','h','e','l','l'];
fn main() -> u32 {
  const COMPILED_CODE: &u32 = 929792;
  let result = buffer_write(COMPILED_CODE, NAME, 5);
  if result == OK { flash_boot_loader_and_reset(); }
  return result;
}
```

Finally, save the builder with F5, compile it with F6, and run it with F7. If all goes well this should launch the kernel, which should launch our test program. The screen should display 0123456789 23 times, followed by a dot. Typing any key should display it on screen. After that you can reset the Arduino, which should restart with the memory editor.

CHAPTER

25 Shell, Text Editor, and Compiler

Our kernel can spawn an initial process which can now use all the computer resources, thanks to the system calls added in the previous chapter. However, we only have a useless test initial program for now. Moreover, in order to store new programs on disk we need the command editor to edit, compile, and save them. And we need to switch back and forth between the basic input output system and the operating system to launch them. The goal of this chapter is to solve these issues by making it possible to edit, compile, save, and launch programs from the operating system alone.

For this we need a text editor and a compiler which can run as processes, and which can read and write files (instead of data buffers). We already have a text editor, but only in binary bytecode form. We thus need to rewrite it in Toy, and to update it to work with files. Our compiler is already in Toy, but also needs to be updated to work with files. Finally, to launch arbitrary programs, we need something like the command editor, but more practical, called a *shell*. This chapter provides these 3 programs. We test them at the end with a "Hello, Word!" application.

25.1 Shell

25.1.1 Requirements

The shell must allow the user to interactively launch arbitrary programs. For this the user must be able to specify which program to launch, and its arguments. On disk programs and data are stored as files, and can thus be specified with file names. Hence, for instance, "`toyc hello hello.toy`" could be used to specify that the program stored in the file named "`toyc`" must be launched with the file names "`hello`" and "`hello.toy`" as arguments. We thus require the following:

- the shell should allow the user to type a line of text called a *command line*, or *command*, after a *prompt* ">". To simplify, we do not require the possibility to insert or delete characters in the middle of a command, but only at the end.
- commands must be of the form "<*program*> <*arguments*>". Typing Enter should spawn the program stored in the file named <*program*>, with <*arguments*> as arguments (which can be empty). The launched program is then responsible to parse <*arguments*> in order to extract each individual argument.

- the shell must display on screen the output of the last executed command. Either an error message if the program could not be launched, or the text written to standard output by this program. It should then repeat the above steps.

To save memory we only require our shell to display the last executed command, its output, and the currently edited command. Older commands are not displayed. For instance, after a "`hello`" command writing "Hello, World!" to standard output, and while typing a new "`edit hello.toy`" command, the screen should display:

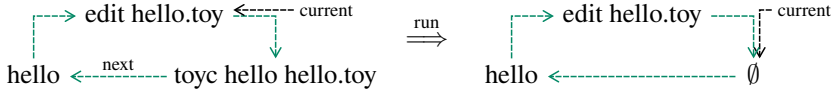
```
>hello
Hello, World!
>edit hello.to_
```

To simplify, for now, we do not require the possibility to save commands, as in the command editor. Still, to avoid having to re-type recent commands, we require the shell to keep in memory a *history* of the last N executed commands. It should then be possible, with the arrow up and down keys, to replace the currently edited command with one from the above history.

25.1.2 Data structures

In order to meet the above requirements we use a buffer to store the currently edited command, N buffers to store the previous N commands, and one buffer to store the output of the last executed one (see Figure 25.1).

After a command is run we must 1) add it to the history, and 2) remove the oldest command from this history. To make this is easy to implement we link the $N + 1$ command buffers in a circular *doubly linked list*. This means that each buffer has a link to a next one and to a previous one, and that these links form two rings going in opposite directions (see the green and red links in Figure 25.1). We also use a pointer to the currently edited command. Then, after this command is run, it suffice to update this pointer to the next command, and to clear this command, to automatically achieve 1) and 2) above:



In fact this only requires links from one command to the next. We use the opposite links to easily go back in the history, with the arrow up key.

In summary we use a `Command` data structure, with two pointers to previous and next commands, and a length indicating the number of characters of this command (followed by the characters themselves – see Figure 25.1). We also use a `Shell` data structure, containing a pointer to one of the commands, as well as pointers to the output of the last command: its beginning (inclusive), its end (exclusive), and the limit of the underlying buffer (exclusive – see Figure 25.1).

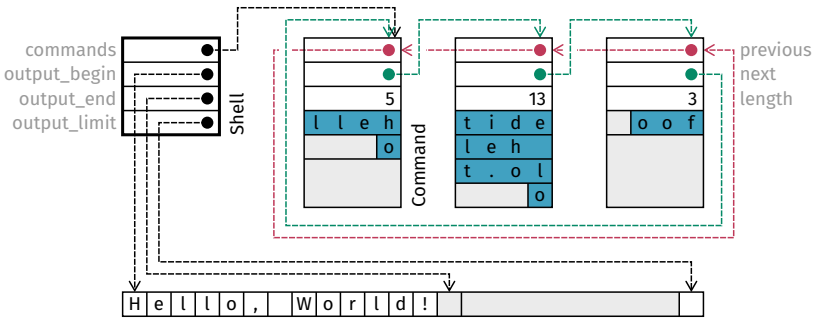


FIGURE 25.1 A shell with a history of two commands ($N = 2$). The user is currently editing an "edit hello.to" command, of length 13. The previous command was "hello" and its output was "Hello, World!". The command before it was "foo".

25.1.3 Implementation

We implement the shell by editing our test application, which already contains some useful code, in order to avoid re-typing it. After the `entry` function, unchanged, we insert the `load8`, `store8`, `load16`, and `store16` functions, copied from our native compiler (the last two are not needed right now but will be useful later).

```
fn load8(ptr: &u32) -> u32 [ /*LDRB_R0_R0_0*/30720; /*MOV_PC_LR*/18167; ]
fn load16(ptr: &u32) -> u32 [ /*LDRH_R0_R0_0*/34816; /*MOV_PC_LR*/18167; ]
fn store8(ptr: &u32, value: u32) [ /*STRB_R1_R0_0*/28673; /*MOV_PC_LR*/181↩
                                                                    ↩67; ]
fn store16(ptr: &u32, value: u32) [ /*STRH_R1_R0_0*/32769; /*MOV_PC_LR*/18↩
                                                                    ↩167; ]
```

We continue with a function to read a token from a command, *i.e.*, the program name or an argument. This function uses a simplified version of Algorithm 15.1, with a pointer to *src* instead of *src* itself as parameter. It returns a pointer to the token's first character, stores its length at address *length* and updates *src* to the address of the first character after it. This allows reading a command with repeated calls to this function with the same first two arguments.

```
fn sh_read_token(src_p: &&u32, src_end: &u32, length: &u32) -> &u32 {
  let src = *src_p;
  while src < src_end && load8(src) == ' ' { src = src + 1; }
  if src >= src_end { return null; }
  let token = src;
  while src < src_end && load8(src) != ' ' { src = src + 1; }
  *length = src - token;
  *src_p = src;
  return token;
}
```

We also copy, after the existing `OK` constant, the error codes from the kernel (which are useful to return errors and analyze the result of commands):

```
const OK: u32 = 0;
const INVALID_ARGUMENT: u32 = 1;
const INVALID_STATE: u32 = 2;
const NOT_FOUND: u32 = 3;
const ALREADY_EXISTS: u32 = 4;
const OUT_OF_MEMORY: u32 = 5;
const INTERNAL_ERROR: u32 = 6;
```

The shell needs to draw text on the screen, with double buffering to avoid flickering (see Section 14.2.3). For this we implement the following functions, after the existing system call functions (from `status` to `reboot`). They are re-implementations of functions of the same names in Sections 10.4 and 14.3, in Toy and using system calls (see these sections and the `draw_char` function from Section 24.5 for more details).

```
fn gpu_set_register(id: u32, value: u32) {
  let buffer = (value & 255) << 16 | (32768 /*Select Register*/ | id);
  write(GPU, &buffer, 4);
}
```

```
fn gpu_set_double_buffer() {
  gpu_set_register(32 /*Display Configuration*/, 128);
  gpu_set_register(65 /*Memory Write Control 1*/, 1);
}
```

```
fn gpu_set_single_buffer() {
  gpu_set_register(65 /*Memory Write Control 1*/, 0);
  gpu_set_register(82 /*Layer Transparency 0*/, 0);
  gpu_set_register(32 /*Display Configuration*/, 0);
}
```

```
fn gpu_switch_buffer() {
  let buffer = 32768 /*Select Register*/ | 65 /*Memory Write Control 1*/;
  write(GPU, &buffer, 2);
  let layer = 0;
  read(GPU, &layer, 1);
  buffer = /*0 (Write Data) | */ 1 - layer;
  write(GPU, &buffer, 2);
  gpu_set_register(82 /*Layer Transparency 0*/, layer);
}
```

```
fn gpu_clear_screen() {
  gpu_set_register(142 /*Memory Clear Control*/, 192);
  let buffer = 0;
  loop {
    read(GPU, &buffer, 1);
    if buffer & 128 == 0 { return; }
  }
}
```

```
fn gpu_set_cursor(col: u32, row: u32) {
  gpu_set_register(42 /*Font Write Cursor H Position 0*/, col << 3);
  gpu_set_register(43 /*Font Write Cursor H Position 1*/, col >> 5);
  gpu_set_register(44 /*Font Write Cursor V Position 0*/, row << 4);
  gpu_set_register(45 /*Font Write Cursor V Position 1*/, row >> 4);
}
```

```
fn gpu_set_color(r: u32, g: u32, b: u32) {
  gpu_set_register(99 /*Foreground Color 0*/, r);
  gpu_set_register(100 /*Foreground Color 1*/, g);
  gpu_set_register(101 /*Foreground Color 2*/, b);
}
```

```
fn gpu_draw_char(c: u32) { gpu_set_register(2, c); }
```

The shell also needs to allocate memory for its data structures, and to copy commands from the history. For this we copy the following functions from the Toy compiler and from the kernel (with a `return null` instead of a `panic`):

```
fn mem_allocate(size: u32, heap_p: &&u32, heap_limit: &u32) -> &u32 {
  let ptr = *heap_p;
  if size > heap_limit as u32 || ptr > heap_limit - size { return null; }
  *heap_p = ptr + size;
  return ptr;
}
fn mem_copy_non_overlapping(src: &u32, dst: &u32, size: u32) {
  let i = 0;
  while i < size {
    store8(dst + i, load8(src + i));
    i = i + 1;
  }
}
```

We can now start the "real" shell implementation, beginning with a definition of its data structures, and of their maximum sizes (`NUM_COMMANDS` is equal to $N + 1$; `data` contains the first 4 characters of a command; hence, `&c.data` points the first character of command `c`):

```
const NUM_COMMANDS: u32 = 4;
const MAX_COMMAND_LENGTH: u32 = 196;
const MAX_OUTPUT_SIZE: u32 = 512;
```

```
struct Command {
  previous: &Command,
  next: &Command,
  length: u32,
  data: u32
}
```

```
struct Shell {
  commands: &Command,
  output_begin: &u32,
  output_end: &u32,
  output_limit: &u32
}
```

We continue with a function to copy the text of a command into another, and a function to create a `Shell` struct. The latter allocates all the required memory, returns `null` if this fails, and initializes the shell and its commands otherwise. In particular, it initializes the `previous` and `next` command links as illustrated in Figure 25.1. These links stay unchanged after that.

```
fn command_copy(src: &Command, dst: &Command) {
  dst.length = src.length;
  mem_copy_non_overlapping(&src.data, &dst.data, src.length);
}
```

```
fn sh_new(heap_p: &&u32, heap_limit: &u32) -> &Shell {
  let shell = mem_allocate(sizeof(Shell), heap_p, heap_limit) as &Shell;
  let command_size = sizeof(Command) - 4 + MAX_COMMAND_LENGTH;
  let command =
      mem_allocate(NUM_COMMANDS * command_size, heap_p, heap_limit) as &Co⌇
                                                                    ⌇mmand;
  let output = mem_allocate(MAX_OUTPUT_SIZE, heap_p, heap_limit);
  if shell == null || command == null || output == null { return null; }
  shell.commands = command;
  shell.output_begin = output;
  shell.output_end = output;
  shell.output_limit = output + MAX_OUTPUT_SIZE;
  let i = 0;
  let previous_command = command + (NUM_COMMANDS - 1) * command_size;
  while i < NUM_COMMANDS {
    previous_command.next = command;
    command.previous = previous_command;
    command.length = 0;
    previous_command = command;
    command = command + command_size;
    i = i + 1;
  }
  return shell;
}
```

The next function appends up to `length` characters from `src` to the shell's output buffer (depending on its remaining capacity):

```
fn sh_print(self: &Shell, src: &u32, length: u32) {
  if length > self.output_limit - self.output_end {
    length = self.output_limit - self.output_end;
```

```
  }
  mem_copy_non_overlapping(src, self.output_end, length);
  self.output_end = self.output_end + length;
}
```

We use it in the following function, which runs the command in $[src, src\_end[$. This function starts by extracting the program name, *i.e.*, the command's first token. If it is not empty it spawns this program, with the rest of the command as arguments, and the shell's output buffer as standard output buffer. It then appends to this buffer an error message if the program could not be launched, if it ran out of memory, or if it crashed. Finally, it appends a new line character if the output of the previous steps is not empty and does not already ends with a new line:

```
static CANT_FIND = ['C','a','n',''','t',' ','f','i','n','d',' '];
static CANT_LAUNCH = ['C','a','n',''','t',' ','l','a','u','n','c','h',' '];
static NOT_ENOUGH_MEMORY = ['O','u','t',' ','o','f',' ','m','e','m','o','r↩
                                                                  ↩','y'];
static CRASHED = [' ','c','r','a','s','h','e','d'];

const NEW_LINE: u32 = 10;

fn sh_run(self: &Shell, src: &u32, src_end: &u32) {
  let length = 0;
  let name = sh_read_token(&src, src_end, &length);
  if name == null { return; }
  let old_end = self.output_end;
  let result = spawn(self.output_limit, name, length, src, src_end - src, ↩
                                                        ↩&self.output_end);
  if status(result) == NOT_FOUND {
    sh_print(self, CANT_FIND, 11);
    sh_print(self, name, length);
  } else if status(result) != OK {
    sh_print(self, CANT_LAUNCH, 13);
    sh_print(self, name, length);
  } else if result == OUT_OF_MEMORY {
    sh_print(self, NOT_ENOUGH_MEMORY, 13);
  } else if result == INTERNAL_ERROR {
    sh_print(self, name, length);
    sh_print(self, CRASHED, 8);
  }
  let new_line = NEW_LINE;
  if self.output_end > old_end && load8(self.output_end - 1) != NEW_LINE {
    sh_print(self, &new_line, 1);
  }
}
```

The next 3 functions are used to draw commands and their output on screen. The first one draws the characters in $[src, src\_end[$, starting at column and row $(col, *row)$ on screen. It automatically starts a new line when a "new line" character is found or

when the maximum line width is reached (100 characters). To simplify, tabulations are not supported.

```
fn sh_draw_string(src: &u32, src_end: &u32, col: u32, row: &u32) {
  let c = 0;
  while src < src_end {
    c = load8(src);
    src = src + 1;
    if c != NEW_LINE {
      gpu_draw_char(c);
      col = col + 1;
    }
    if c == NEW_LINE || col == 100 {
      col = 0;
      *row = *row + 1;
      gpu_set_cursor(col, *row);
    }
  }
}
```

The second one draws the given command, after a ">" prompt, and updates $*row$ to the next screen row:

```
fn sh_draw_command(command: &Command, row: &u32) {
  gpu_draw_char('>');
  sh_draw_string(&command.data, &command.data + command.length, 1, row);
  *row = *row + 1;
}
```

The third one draws the last executed command in green (if it is not empty), its output in yellow (if it is not empty), and the currently edited command in green:

```
fn sh_draw(self: &Shell, current_command: &Command) {
  gpu_clear_screen();
  gpu_set_cursor(0, 0);
  gpu_set_color(0, 7, 0);
  let row = 0;
  if current_command.previous.length > 0 {
    sh_draw_command(current_command.previous, &row);
    gpu_set_cursor(0, row);
  }
  if self.output_end > self.output_begin {
    gpu_set_color(7, 7, 0);
    sh_draw_string(self.output_begin, self.output_end, 0, &row);
    gpu_set_cursor(0, row);
    gpu_set_color(0, 7, 0);
  }
  sh_draw_command(current_command, &row);
  gpu_switch_buffer();
}
```

The next function handles characters typed on the keyboard in a loop, and redraws the screen after each key typed (with the above function). It maintains two pointers:

- `current` points to the currently edited command. It is moved to the next command after the current one is run.
- `history` can point to any command. It is moved to the previous or next command with the arrow up and down keys, respectively. Selecting a new history command copies it into the current one.

The supported keys and their associated actions are the following:

- the Escape key exits the shell. This can fail if the current process is the initial one (this is not always the case since the shell can also be spawned by a process). In this case we restore double buffering, which is disabled before calling `exit`.
- the Enter key executes the current command, sets the next command as the current one, clears it, and resets the `history` pointer to `current`.
- the ArrowUp key moves the history command to the previous one, unless there is none (*i.e.*, unless this would circle back to the current one).
- the ArrowDown key moves the history command to the next one, if there is one. If this goes back to the current command we clear it instead of copying it to itself.
- the Backspace key deletes the last character of the current command. Printable ASCII characters are appended to the current command, if it is not full.

```
const BACKSPACE_KEY: u32 = 8;
const ENTER_KEY: u32 = 10;
const ESCAPE_KEY: u32 = 27;
const DELETE_KEY: u32 = 127;
const ARROW_UP_KEY: u32 = 245;
const ARROW_DOWN_KEY: u32 = 242;
```

```
fn sh_run_editor(self: &Shell) -> u32 {
  gpu_set_double_buffer();
  let current = self.commands;
  let history = current;
  let c = 0;
  loop {
    sh_draw(self, current);
    read(STANDARD_INPUT, &c, 1);
    if c == ESCAPE_KEY {
      gpu_set_single_buffer();
      exit(OK);
      gpu_set_double_buffer();
    } else if c == ENTER_KEY {
      self.output_end = self.output_begin;
      sh_run(self, &current.data, &current.data + current.length);
      gpu_set_double_buffer();
      current = current.next;
```

```
      current.length = 0;
      history = current;
    } else if c == ARROW_UP_KEY && history.previous != current {
      history = history.previous;
      command_copy(history, current);
    } else if c == ARROW_DOWN_KEY && history != current {
      history = history.next;
      if history == current {
        current.length = 0;
      } else {
        command_copy(history, current);
      }
    } else if c == BACKSPACE_KEY && current.length > 0 {
      current.length = current.length - 1;
    } else if c >= 32 && c < DELETE_KEY && current.length < MAX_COMMAND_LE⟩
                                                                  ⟩NGTH {
      store8(&current.data + current.length, c);
      current.length = current.length + 1;
    }
  }
}
```

Finally, we delete the existing functions (from `wait_char` to `child`), and replace the main function with the following one. This function simply creates and runs the shell if there are no arguments. Otherwise, for testing purposes, it writes them to standard output.

```
fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  let shell = sh_new(&heap, heap_limit);
  if shell == null { return OUT_OF_MEMORY; }
  if args_end > args {
    write(STANDARD_OUTPUT, args, args_end - args);
    return OK;
  }
  return sh_run_editor(shell);
}
```

25.1.4 Compilation and tests

Type "F11"+"r" in the command editor to load the current application source code, and "F4"+"r" to edit it. Then update it as described above. For reference, we also provide this new version in the `shell_v0.txt` file in https://ebruneton.github.io/toypc/sources.zip. When done, type "F12"+"r" to save it and "F9"+"r" to compile it. If necessary, repeat these steps until the compilation is successful. To copy the source code (at address $DD000_{16}$ = 905216) and the compiled code in files use F3 to load the builder source code, and F4 to edit it. Then change its main function to the following:

```
static NAME = ['s','h','e','l','l'];
static SOURCE = ['s','h','e','l','l','.','t','o','y'];
```

```
fn main() -> u32 {
  const COMPILED_CODE: &u32 = 929792;
  const SOURCE_CODE: &u32 = 905216;
  let result = buffer_write(COMPILED_CODE, NAME, 5);
  if result == OK { result = buffer_write(SOURCE_CODE, SOURCE, 9); }
  if result == OK { flash_boot_loader_and_reset(); }
  return result;
}
```

Finally, save the builder with F5, compile it with F6, and run it with F7. If all goes well this should launch the kernel and the shell. To test it, try the commands "`hello world`" and "`shell hello world`". The former should fail because there is no program named "`hello`". The latter should print its arguments, *i.e.*, "`hello world`". You can also try the "`shell`" command, which should launch another shell. You can check this with the arrow keys, which should not show the previous commands. Then type Escape to return in the initial shell, and try the Escape, arrow up and down keys again. Finally, reset the Arduino, which should restart with the memory editor.

25.2 Text editor

25.2.1 Requirements

The text editor should take as argument the name of the file to edit, and should create it if it does not exist. It should also take an initial text cursor position as an optional argument. The goal is to be able to edit a source file directly at the location of an error indicated by the compiler. Typing Escape should show a message asking the user whether to save the file or not before exiting.

25.2.2 Implementation

We implement the text editor by editing the shell program, which already contains some useful code, in order to avoid re-typing it. More precisely we keep the beginning of its code, up to the `mem_allocate` function (excluded), and delete everything else. We then re-implement the functions from Section 14.3, and the `mem_copy` function from Section 13.2 that they need (see these sections for more details):

```
fn mem_copy(src: &u32, dst: &u32, n: u32) {
  let i = 0;
  if dst < src {
    while i + 4 <= n {
      *(dst + i) = *(src + i);
      i = i + 4;
    }
    while i < n {
      store8(dst + i, load8(src + i));
      i = i + 1;
```

```
    }
  } else {
    i = n;
    while i >= 4 {
      i = i - 4;
      *(dst + i) = *(src + i);
    }
    while i > 0 {
      i = i - 1;
      store8(dst + i, load8(src + i));
    }
  }
}
```

```
const ARROW_DOWN_KEY: u32 = 242;
const ARROW_LEFT_KEY: u32 = 235;
const ARROW_RIGHT_KEY: u32 = 244;
const ARROW_UP_KEY: u32 = 245;
const BACKSPACE_KEY: u32 = 8;
const DELETE_KEY: u32 = 127;
const ENTER_KEY: u32 = 10;
const ESCAPE_KEY: u32 = 27;
const PAGE_DOWN_KEY: u32 = 250;
const PAGE_UP_KEY: u32 = 253;
const TAB_KEY: u32 = 9;
```

```
fn ted_set_cursor(begin: &u32, cursor: &u32, gap: u32, new_cursor: &u32) -≀
                                                                    ≀> &u32 {
  if new_cursor > cursor {
    mem_copy(cursor + gap, cursor, new_cursor - cursor);
  } else {
    mem_copy(new_cursor, new_cursor + gap, cursor - new_cursor);
  }
  return new_cursor;
}
```

```
fn ted_move_backward(begin: &u32, cursor: &u32, lines: u32, col: &u32, row≀
                                                          ≀: &u32) -> &u32 {
  let c = 0;
  let ptr = cursor;
  while ptr > begin {
    c = load8(ptr - 1);
    if c == ENTER_KEY {
      if *row == lines { break; }
      *row = *row + 1;
    } else if *row == 0 {
      if c == TAB_KEY {
```

```
        *col = *col + 2;
      } else {
        *col = *col + 1;
      }
    }
    ptr = ptr - 1;
  }
  return ptr;
}
```

```
fn ted_move_forward(cursor: &u32, gap: u32, end: &u32, lines: u32) -> &u32{
                                                                        } {
  loop {
    if cursor == end - gap { return cursor; }
    if load8(cursor + gap) == ENTER_KEY {
      if lines == 1 { return cursor + 1; }
      lines = lines - 1;
    }
    cursor = cursor + 1;
  }
}
```

```
fn ted_handle_key(begin: &u32, cursor: &u32, gap: u32, end: &u32, c: u32) {
                                                                  }-> &u32 {
  let col = 0;
  let row = 0;
  if c == ARROW_LEFT_KEY && cursor > begin {
    return cursor - 1;
  } else if c == ARROW_RIGHT_KEY && cursor < end - gap {
    return cursor + 1;
  } else if c == ARROW_UP_KEY {
    return ted_move_backward(begin, cursor, 1, &col, &row);
  } else if c == ARROW_DOWN_KEY {
    return ted_move_forward(cursor, gap, end, 1);
  } else if c == PAGE_UP_KEY {
    return ted_move_backward(begin, cursor, 30, &col, &row);
  } else if c == PAGE_DOWN_KEY {
    return ted_move_forward(cursor, gap, end, 30);
  }
  return cursor;
}
```

```
fn ted_draw(begin: &u32, cursor: &u32, gap: u32, end: &u32) {
  gpu_clear_screen();
  gpu_set_cursor(0, 0);
  let r = 0;
  let c = 0;
```

```
  let col = 0;
  let row = 0;
  let ptr = ted_move_backward(begin, cursor, 15, &col, &row);
  if ptr == cursor { ptr = ptr + gap; }
  while ptr < end && r < 30 {
    c = load8(ptr);
    if c == ENTER_KEY {
      r = r + 1;
      gpu_set_cursor(0, r);
    } else if c == TAB_KEY {
      gpu_draw_char(' ');
      gpu_draw_char(' ');
    } else {
      gpu_draw_char(c);
    }
    ptr = ptr + 1;
    if ptr == cursor { ptr = ptr + gap; }
  }
  gpu_switch_buffer();
  gpu_set_cursor(col, row);
}
```

```
fn text_editor(buffer: &u32, offset: u32, max_length: u32) {
  let length = *buffer;
  if length > max_length { return; }
  let begin = buffer + 4;
  let cursor = begin + length;
  let end = begin + max_length;
  let gap = end - cursor;
  let c = 0;
  if offset > length { offset = length; }
  cursor = ted_set_cursor(begin, cursor, gap, begin + offset);
  gpu_set_color(0, 7, 0);
  gpu_set_double_buffer();
  ted_draw(begin, cursor, gap, end);
  loop {
    read(STANDARD_INPUT, &c, 1);
    if c == ESCAPE_KEY {
      *buffer = ted_set_cursor(begin, cursor, gap, end - gap) - begin;
      gpu_set_single_buffer();
      return;
    }
    if c == BACKSPACE_KEY {
      if cursor > begin {
        cursor = cursor - 1;
        gap = gap + 1;
      }
```

```
    } else if c < DELETE_KEY {
      if gap > 0 {
        store8(cursor, c);
        cursor = cursor + 1;
        gap = gap - 1;
      }
    } else {
      cursor = ted_set_cursor(begin, cursor, gap,
          ted_handle_key(begin, cursor, gap, end, c));
    }
    ted_draw(begin, cursor, gap, end);
  }
}
```

The rest of the code is new and implements the above requirements. The following function computes the value of the optional command line argument, the initial cursor position. It stores the value of this argument, if present, at address `offset`. It returns an error if this argument is not a number, and `OK` otherwise. $[args, args\_end[$ must be the rest of the command line arguments, after the name of the file to edit.

```
fn read_offset(args: &&u32, args_end: &u32, offset: &u32) -> u32 {
  let length = 0;
  let argument = sh_read_token(args, args_end, &length);
  if argument == null { return OK; }
  let i = 0;
  let c = 0;
  while i < length {
    c = load8(argument + i);
    if c < '0' || c > '9' { return INVALID_ARGUMENT; }
    *offset = (*offset) * 10 + (c - '0');
    i = i + 1;
  }
  return OK;
}
```

The next function simply returns an error after writing a corresponding error message to standard output. It is followed by some messages needed in `main`:

```
fn write_error(src: &u32, length: u32, error: u32) -> u32 {
  write(STANDARD_OUTPUT, src, length);
  return error;
}
```

```
static USAGE = ['U','s','a','g','e',':',' ',
  'e','d','i','t',' ','f','i','l','e',' ',
  '[','o','f','f','s','e','t',']'];
```

```
static READ_ERROR = ['R','e','a','d',' ','e','r','r','o','r'];
static WRITE_ERROR = ['W','r','i','t','e',' ','e','r','r','o','r'];
static SAVE_PROMPT = ['S','a','v','e',' ','(','y','/','n',')','?'];
```

The `main` function starts by reading the command line arguments. If they are invalid, it prints a message explaining their expected format and returns an error. It also checks that the heap is large enough to edit a small text.

```
fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  let length = 0;
  let name = sh_read_token(&args, args_end, &length);
  let offset = 0;
  if name == null || read_offset(&args, args_end, &offset) != OK {
    return write_error(USAGE, 25, INVALID_ARGUMENT);
  }
  if heap_limit < heap + 256 { return OUT_OF_MEMORY; }
```

It continues by reading the file to edit, if it exists, into the data buffer named `buffer`, using all the available heap memory (minus 4 bytes for the data buffer header). Otherwise it initializes an empty buffer.

```
  let buffer = heap;
  let max_length = heap_limit - heap - 4;
  let stream = open(name, length, 'r');
  if status(stream) == OK {
    *buffer = read(stream, buffer + 4, max_length);
    close(stream);
    if status(*buffer) != OK {
      return write_error(READ_ERROR, 10, status(*buffer));
    }
    if *buffer == max_length { return OUT_OF_MEMORY; }
  } else {
    *buffer = 0;
  }
```

The end of the function calls `text_editor` to edit this buffer. When it returns, it displays a message asking the user whether the changes must be saved or not, with `ted_draw`. If the user types "y" it saves the `buffer` content (without its 4 bytes header) into the edited file and returns `OK`, or an error if the file could not be saved.

```
  text_editor(buffer, offset, max_length);
  ted_draw(SAVE_PROMPT, SAVE_PROMPT + 11, 0, SAVE_PROMPT + 11);
  let c = 0;
  read(STANDARD_INPUT, &c, 1);
  if c != 'y' { return OK; }
  stream = open(name, length, 'w');
  if status(stream) != OK {
    return write_error(WRITE_ERROR, 11, status(stream));
  }
  let n = write(stream, buffer + 4, *buffer);
  if status(n) != OK {
    return write_error(WRITE_ERROR, 11, status(n));
  }
  return OK;
}
```

25.2.3 Compilation and tests

Type "F11"+"r" in the command editor to load the current shell source code, and "F4"+"r" to edit it. Then update it as described above. For reference, we also provide this code in the `edit_v0.txt` file in https://ebruneton.github.io/toypc/sources.zip. When done, type "F12"+"r" to save it and "F9"+"r" to compile it. If necessary, repeat these steps until the compilation is successful. To copy the text editor source code and compiled code in files use F3 to load the builder source code, and F4 to edit it. Then change its main function to the following:

```
static NAME = ['e','d','i','t'];
static SOURCE = ['e','d','i','t','.','t','o','y'];
fn main() -> u32 {
  const COMPILED_CODE: &u32 = 929792;
  const SOURCE_CODE: &u32 = 905216;
  let result = buffer_write(COMPILED_CODE, NAME, 4);
  if result == OK { result = buffer_write(SOURCE_CODE, SOURCE, 8); }
  if result == OK { flash_boot_loader_and_reset(); }
  return result;
}
```

Finally, save the builder with F5, compile it with F6, and run it with F7. If all goes well this should launch the kernel and the shell. To test the text editor, type "`edit test.txt`" and Enter. Type some text and then Escape and "y" to save it. To check that this worked, type "`edit test.txt 2`": the text editor should display the text you just saved, and the cursor should be under the third character. Finally, reset the Arduino, which should restart with the memory editor.

25.3 Compiler

25.3.1 Requirements

The shell and the text editor start with the same source code which is in fact useful for most programs (namely the `entry` function, and the system call functions). To avoid duplicating it in each program we require the possibility to compile a program from several source files. We can then put the shared code in a separate file, implemented only once, and compile it together with a program specific file. The compiler should thus take as arguments the name of the compiled program to generate, followed by one or more source file name(s). In case of a compilation error it should write the error code, the error location, and the name of the source file to standard output. The goal is to be able to edit this file directly at the location of the error with the text editor.

25.3.2 Design

We meet the above requirements with a simple method requiring very few changes to our compiler, but which is not memory efficient (we improve it in Chapter 27). More

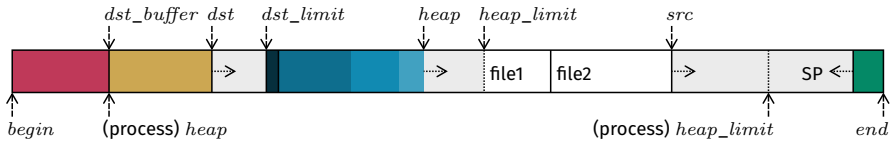


FIGURE 25.2 The compiler process's heap (bottom), between its compiled code (red) and its stack (green), contains the generated code (yellow), the `Compiler` struct (dark blue), the compiler's heap (shades of blue), and the input source code (white).

precisely, we load all the input files in RAM, after the compiler's heap, which is itself only a part of the compiler process's heap (compare Figure 25.2 with Figures 19.4 and 23.1). And we compile each file with `tc_parse_program`, one by one, into the same *dst_buffer* that we finally save into a file.

25.3.3 Implementation

We implement the above design by editing the current compiler code as follows. We first replace the `tc_main` function declaration and the `main` function with the same `entry` function as in the shell and the text editor (we don't have a compiler able to compile several files yet, and thus need to duplicate it for now):

```
fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32;
fn exit(result: u32) -> u32;
```

```
fn entry(heap: &u32, heap_limit: &u32) {
  let args = heap + 4;
  let args_end = args + *heap;
  heap = (((args_end as u32 + 3) >> 2) << 2) as &u32;
  exit(main(args, args_end, heap, heap_limit));
}
```

Then, after `load8`, `load16`, `store8`, and `store16` (unchanged), and for the same reason, we copy the following functions from the shell and the text editor:

```
fn sh_read_token(src_p: &&u32, src_end: &u32, length: &u32) -> &u32 {
  let src = *src_p;
  while src < src_end && load8(src) == ' ' { src = src + 1; }
  if src >= src_end { return null; }
  let token = src;
  while src < src_end && load8(src) != ' ' { src = src + 1; }
  *length = src - token;
  *src_p = src;
  return token;
}
```

```
const OK: u32 = 0;
const INVALID_ARGUMENT: u32 = 1;
```

```
const INVALID_STATE: u32 = 2;
const NOT_FOUND: u32 = 3;
const ALREADY_EXISTS: u32 = 4;
const OUT_OF_MEMORY: u32 = 5;
const INTERNAL_ERROR: u32 = 6;

fn status(result: u32) -> u32 { return result >> 24; }

fn system_call(id: u32, args: &u32) -> u32 [
  /*SVC*/ 57088;
  /*MOV_PC_LR*/ 18167;
]
fn exit(result: u32) -> u32 {
  return system_call(1, &result);
}
fn open(name: &u32, length: u32, mode: u32) -> u32 {
  return system_call(4, &name as &u32);
}
const STANDARD_OUTPUT: u32 = 1;
fn read(file_descriptor: u32, buffer: &u32, size: u32) -> u32 {
  return system_call(5, &file_descriptor);
}
fn write(file_descriptor: u32, buffer: &u32, size: u32) -> u32 {
  return system_call(6, &file_descriptor);
}
fn close(file_descriptor: u32) -> u32 {
  return system_call(7, &file_descriptor);
}
```

We then keep all the existing code between `panic_result` and `tc_parse_fn` unchanged, but we remove the call to `tc_check_symbols` in `tc_parse_program`. Otherwise a function declared in one file would need to be implemented in this same file, which is too restrictive (recall that `tc_check_symbols` checks that all declared functions are effectively implemented).

```
fn tc_parse_program(self: &Compiler) {
  loop {
...
      if self.next_token != 0 { panic(23); }
      return;
    }
  }
}
```

The following code is new. It replaces the `tc_main` function and implements the above requirements. We start with a function to write a decimal number x to standard output, which is needed to write an error code or its location. This function writes x divided by 10 by calling itself recursively, followed by the remainder of this division. The next function is similar to the one with the same name in the text editor:

```
fn write_integer(x: u32) {
  let quotient = x / 10;
  x = x - 10 * quotient + '0';
  if quotient > 0 { write_integer(quotient); }
  write(STANDARD_OUTPUT, &x, 1);
}
```

```
fn write_error(src1: &u32, length1: u32, src2: &u32, length2: u32, error: ⌇
                                                              ⌇u32) -> u32 {
  write(STANDARD_OUTPUT, src1, length1);
  write(STANDARD_OUTPUT, src2, length2);
  return error;
}
```

The `main` function starts by reading the command line arguments. If there are less than two it prints a message explaining their expected format and returns an error. It then increases the stack area, by decreasing the process's heap limit, and checks that this heap is large enough to compile a small program (see Figure 25.2).

```
static USAGE = ['U','s','a','g','e',':',' ',
  't','o','y','c',' ','o','u','t','p','u','t',' ',
  'i','n','p','u','t','1',' ','i','n','p','u','t','2',' ',
  '.','.','.'];
```

```
static CANT_OPEN = ['C','a','n',''','t',' ','o','p','e','n',' '];
static CANT_READ = ['C','a','n',''','t',' ','r','e','a','d',' '];
static CANT_WRITE = ['C','a','n',''','t',' ','w','r','i','t','e',' '];
static ERROR = ['E','r','r','o','r',' '];
static AT = [' ','a','t',  ' '];
static IN = [' ','i','n',  ' '];
```

```
fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  let out_length = 0;
  let in_length = 0;
  let out = sh_read_token(&args, args_end, &out_length);
  let in = sh_read_token(&args, args_end, &in_length);
  if out == null || in == null {
    write(STANDARD_OUTPUT, USAGE, 36);
    return INVALID_ARGUMENT;
  }
  const MAX_CODE_SIZE: u32 = 12288;
  const MAX_HEAP_SIZE: u32 = 18432;
  const MIN_SRC_SIZE: u32 = 256;
  heap_limit = heap_limit - 512;
  if heap_limit < heap + MAX_CODE_SIZE + sizeof(Compiler) + MAX_HEAP_SIZE ⌇
                                                          ⌇+ MIN_SRC_SIZE {
    return OUT_OF_MEMORY;
  }
```

The main function continues by creating and initializing the `compiler` struct. Note that *dst* is a multiple of 4, as required (see Section 20.3), thanks to the rounding done in the `entry` function. It also implements a "panic handler" writing the error code, offset and source file name to standard output, as required (*src* points to the beginning of the source code loaded from the file whose name starts at *in*).

```
let error = 0;
let dst = heap;
let compiler = (dst + MAX_CODE_SIZE) as &Compiler;
compiler.dst = dst;
compiler.dst_limit = compiler as &u32;
compiler.heap = compiler.dst_limit + sizeof(Compiler);
compiler.heap_limit = compiler.heap + MAX_HEAP_SIZE;
compiler.symbols = null;
let src = compiler.heap_limit;
error = panic_result();
if error != 0 {
  write(STANDARD_OUTPUT, ERROR, 6);
  write_integer(error);
  if in == null { return error; }
  write(STANDARD_OUTPUT, AT, 4);
  write_integer(compiler.src - src);
  return write_error(IN, 4, in, in_length, error);
}
```

This handler is followed by a loop which reads each input source file and compiles it. Each file is loaded at *src*, which is then incremented by the file size. If the file can be read successfully and if there is enough memory to load its content, it is compiled with `tc_parse_program`. This requires initializing the scanner first, with `tc_read_char` and `tc_read_token`, which in turn requires initializing the compiler's *src* and *src_end* fields.

```
let stream = 0;
let src_size = 0;
while in != null {
  stream = open(in, in_length, 'r');
  if status(stream) != OK {
    return write_error(CANT_OPEN, 11, in, in_length, status(stream));
  }
  src_size = read(stream, src, heap_limit - src);
  close(stream);
  if status(src_size) != OK || src_size == heap_limit - src {
    return write_error(CANT_READ, 11, in, in_length, status(src_size));
  }
  compiler.src = src - 1;
  compiler.src_end = src + src_size;
  tc_read_char(compiler);
  tc_read_token(compiler);
```

```
    tc_parse_program(compiler);
    in = sh_read_token(&args, args_end, &in_length);
    src = src + src_size;
  }
```

The `main` function ends by checking that all declared functions, in all files, are effectively implemented, and by writing the compiled code to disk:

```
  tc_check_symbols(compiler.symbols, null);
  stream = open(out, out_length, 'w');
  if status(stream) != OK {
    return write_error(CANT_OPEN, 11, out, out_length, status(stream));
  }
  let n = write(stream, dst, compiler.dst - dst);
  if status(n) != OK {
    return write_error(CANT_WRITE, 12, out, out_length, status(n));
  }
  return OK;
}
```

25.3.4 Compilation and tests

To update the compiler as described above we first need some commands to load and save its source code, at address $D1000_{16}$ = 856064 = page 272 (see Figure 23.7). For this type "F11"+"e" in the command editor, update this command to:

```
fn 0
  cst 856064 cst 537330176 call 2708
  cst_0 retv
d LOAD_COMPILER_SOURCE_CODE
```

and type "s" to save it. With the same method, update the F12 command to:

```
fn 0
  cst 537330176 cst 272 call 2836
  cst_0 retv
d SAVE_COMPILER_SOURCE_CODE
```

Then use F11 and F4 to load and edit the current compiler source code, and update it as described above. For reference, we also provide this code in the `toyc_v0.txt` file in https://ebruneton.github.io/toypc/sources.zip. When done, use F12 and F9 to save it and to compile it. If necessary, repeat these steps until the compilation is successful. To copy the new compiler source code and compiled code in files use F3 to load the builder source code, and F4 to edit it. Then change its main function to the following:

```
static NAME = ['t','o','y','c'];
static SOURCE = ['t','o','y','c','.','t','o','y'];
fn main() -> u32 {
```

```
  const COMPILED_CODE: &u32 = 929792;
  const SOURCE_CODE: &u32 = 856064;
  let result = buffer_write(COMPILED_CODE, NAME, 4);
  if result == OK { result = buffer_write(SOURCE_CODE, SOURCE, 8); }
  if result == OK { flash_boot_loader_and_reset(); }
  return result;
}
```

Finally, save the builder with F5, compile it with F6, and run it with F7. If all goes well this should launch the kernel and the shell. To test our new compiler, first create a "`hello.toy`" file containing the following program (*i.e.*, type "`edit hello.toy`" in the shell, enter this code, and finally type Escape and "y" to save it):

```
fn main();
fn entry(heap: &u32, heap_limit: &u32) { main(); }

fn system_call(id: u32, args: &u32) -> u32 [
  /*SVC*/ 57088;
  /*MOV_PC_LR*/ 18167;
]
fn exit(result: u32) -> u32 {
  return system_call(1, &result);
}
fn write(stream_id: u32, buffer: &u32, size: u32) -> u32 {
  return system_call(6, &stream_id);
}

static HELLO = ['H','e','l','l','o',',',' ','W','o','r','l','d','!'];
fn main() {
  write(1 /*standard output*/, HELLO, 13);
  exit(0);
}
```

Then compile it with "`toyc hello hello.toy`". If there is a compilation error, edit the source code at the location indicated by the compiler. Once the compilation is successful, run the compiled program with the "`hello`" command line, which should print the "Hello, World!" message. You can then reset the Arduino, which should restart with the memory editor.

25.4 Self hosting

We now have everything we need to edit, compile and run new programs from our operating system alone. Moreover, the file system contains the source code of the shell, the text editor, and the compiler (and of the boot loader). We can thus update them and recompile them from the operating alone too. In other words, we no longer need our bytecode interpreter, the basic input output system, the command editor, or anything else we built in the Flash1 memory bank. In particular, we no longer

need to restore the BIOS Vector Table in our kernel to restart the basic input output system after a reset of the operating system. To remove it, type "F8"+"r" in the command editor to load the kernel source code, and "F4"+"r" to edit it. Then delete the `restore_bios_vector_table` function, and remove the call to this function in `os_init`. Use F10 to save these changes and F9 to compile them.

To copy this new kernel, and its source code (so that we can update it from the operating system alone), use F3 to load the builder source code, and F4 to edit it. Then change its main function, one last time, to the following (the kernel source code is at address $F4000_{16}$ = 999424 – see Figure 23.7):

```
static NAME = ['t','o','y','s'];
static SOURCE = ['t','o','y','s','.','t','o','y'];
fn main() -> u32 {
  const COMPILED_CODE: &u32 = 929792;
  const SOURCE_CODE: &u32 = 999424;
  let result = buffer_write(COMPILED_CODE, NAME, 4);
  if result == OK { result = buffer_write(SOURCE_CODE, SOURCE, 8); }
  if result == OK { flash_boot_loader_and_reset(); }
  return result;
}
```

Finally, save the builder with F5, compile it with F6, and run it with F7. If all goes well this should launch the kernel and the shell. Then reset the Arduino. This should relaunch the kernel and the shell again.

CHAPTER

26 Memory Protection

Our operating system is now self-hosting, which means that we can edit and recompile its entire source code with itself. But the kernel is not finished. Indeed, a bug in a process can currently crash everything, instead of exiting this process with an error code, as required in Chapter 23. To meet this requirement the plan was to make sure that a process can only access its own memory region (see Section 23.2). This chapter implements this strategy with the help of the microprocessor's Memory Protection Unit, which is presented first.

26.1 Memory Protection Unit

The Memory Protection Unit (MPU) is a component of the Cortex M3 microprocessor inside the microcontroller (see Figure 6.2). As briefly introduced in Section 7.1, it can divide the memory into regions, and can associate different access rights with each region. For instance, one region can be made inaccessible, another read-only, etc. If an instruction tries to read memory in an inaccessible region, or to write a value in a read-only one, a Memory Management exception is triggered. If the handler for this type of exceptions is not explicitly enabled, the generic Hard Fault handler is called instead.

26.1.1 Regions

The MPU uses 8 configurable regions, numbered from 0 to 7, plus a non-configurable "background region". Each of the 8 regions has the following configurable properties:

- a *base address* and a *size*. These properties define the region boundaries, namely $[base\ address, base\ address + size[$. *size* must be a power of 2 larger than or equal to 32, and *base address* must be a multiple of *size*.
- some *attributes*, which include *access permissions* such as "no access", "read-only" or "full access".
- an *enabled* bit. A disabled region does not have any effect on any memory access.

The background region can only be enabled or disabled. If enabled, it allows full access to all memory addresses (*i.e.*, its base address is 0, its size is 4 GB, and its access permissions are "full access").

Finally, the MPU as a whole can be enabled or not. If it is disabled, which is the default after a reset, all memory accesses are permitted. If it is enabled, a memory access at an address a is checked as follows:

- if a is inside at least one enabled region, access is checked against the permissions of the enabled region containing a with the largest number. For instance, if a is inside region 1, enabled with "no access", and also inside region 3, enabled as "read-only", then reading the value at a is permitted. As another example, if the background region is enabled and if a is inside region 2, enabled with "no access", then any access to a is rejected (configurable regions take precedence over the background region).
- if a is outside the boundaries of all enabled regions, access is rejected.

26.1.2 Subregions

Regions whose size is larger than or equal to 256 bytes are divided in 8 subregions of equal size. For instance, a 32 KB region starting at address 512 KB is divided in 8 subregions of 4 KB each: subregion 0 in $[512\text{ KB}, 516\text{ KB}[$, subregion 1 in $[516\text{ KB}, 520\text{ KB}[$, etc. These subregions have the same attributes as the enclosing region, but can be independently enabled or disabled. For the above access checking rules, an address a is inside an enabled region R if R is enabled, and if a is inside an enabled subregion of R.

26.1.3 Privilege level

The MPU regions are configured with some registers, presented in the next section, inside the "System" memory region (see Figure 6.3). If the MPU is configured to forbid all memory accesses outside the RAM region of a process, then these registers become inaccessible. In other words, it becomes impossible to reconfigure the MPU to switch to another process. A solution to this problem could be to allow only memory accesses to the memory region of a process *or* to the "System" region. But then, due to a bug or an intentional "attack", the process could access the MPU registers and reconfigure them to allow all memory accesses!

To solve these issues the Cortex M3 can run at one of two *privilege levels*: privileged or unprivileged. At the privileged level the MPU registers are always accessible, even if the MPU would otherwise forbid this. The above problems can thus be solved if 1) the kernel executes at the privileged level and 2) processes execute at the unprivileged level and have no way to change this level. Indeed, in this case:

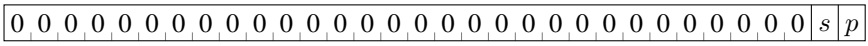
- the MPU can be configured to forbid any memory access outside a process's memory region, while this process is running. The process is then unable to reconfigure the MPU because it cannot access its registers, and because it cannot switch to privileged level.
- when the kernel is running it can access the MPU registers thanks to its privileges, and can thus reconfigure the MPU to switch to another process.

Name	Type	Address
Control Register	Read-Write	$E000ED94_{16}$
Region Number Register	Read-Write	$E000ED98_{16}$
Region Base Address Register	Read-Write	$E000ED9C_{16}$
Region Attribute and Size Register	Read-Write	$E000EDA0_{16}$

TABLE 26.1 The Memory Protection Unit registers used in this book.

Exception and interrupt handlers always execute at the privileged level, which ensures 1) above (after its initialization, the kernel always runs inside some exception or interrupt handler, such as the USART or SVC handler). Outside exception and interrupt handlers, *i.e.*, in Thread mode, the microprocessor can run at any privilege level. This is configured with a bit of the CONTROL register (see Section 23.2.2), which can only be modified at the privileged level (thus ensuring 2) above).

In fact the CONTROL register, which can be accessed with the MRS and MSR instructions (see Section 23.2.2), has the following binary format:

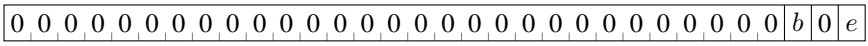


where p specifies the privilege level in Thread mode, and s selects the current Stack Pointer. More precisely, the default value $p = 0$ configures Thread mode to run at privileged level, and $p = 1$ to run at unprivileged level. $s = 1$ selects the Process Stack Pointer, and $s = 0$ selects the Main Stack Pointer (see Section 23.2.2). Setting s in Handler mode has no effect since this mode always uses the Main Stack Pointer.

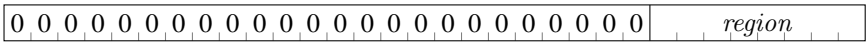
26.1.4 MPU registers

The Memory Protection Unit is configured with the following registers (see Table 26.1):

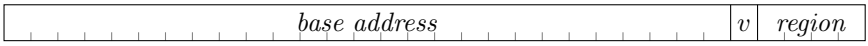
- The Control Register enables or disables the MPU and the background region. Its binary format is the following (we show only the bits that we use):

  where $e = 1$ enables the MPU, $e = 0$ disables it, $b = 1$ enables the background region at the privileged level, and $b = 0$ disables it. *The background region is always disabled at the unprivileged level.*
- The Region Number Register defines a "current region number", used by the next registers (*region* must be between 0 and 7 included):

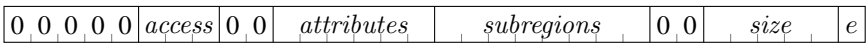
- The Region Base Address Register defines the base address of a region:

  If $v = 1$ then writing into this register stores *region* in the Region Number Register, and sets the base address of this region. If $v = 0$ writing into this register sets the base address of the region whose number is stored in the Region Number Register.

In both cases, the region's base address is set to $32 * base\ address$. For instance, setting this register to $B3_{16}$, *i.e.*, setting $base\ address$ to 5, v to 1 and $region$ to 3 sets the base address of region 3 to $32 * 5 = 160$.

- The Region Attribute and Size Register defines the size, access permissions and subregions of the region whose number is stored in the Region Number Register. Its binary format is the following (we show only the bits that we use):



  - $e = 1$ enables this region, and $e = 0$ disables it.
  - $size$ sets the region size to 2^{size+1} ($size$ must be larger than or equal to 4).
  - $subregions$ specifies which subregions are *disabled*: the i^{th} subregion, counting from 0, is disabled if the i^{th} bit of $subregions$ is 1. For instance, $subregions = 5 = 101_2$ disables subregions 0 and 2 and enables the others.
  - $attributes$ depends on the region type (Flash, RAM, etc). For a RAM region, the recommended value is 6 (see Sections 10.23.5 and 10.23.9.1 in [8]).
  - $access$ defines the region's access permissions. $access = 0$ means "no access", 3 means "full access", and 7 means "read-only" (see Table 10.39 in [8]).

Note that, with the above registers, it is possible to set a base address which is not a multiple of the region size, as required. In this case the MPU uses this base address, rounded down to a multiple of the region size, to perform the access checks. For instance, if a region is set to $[96, 96 + 64[$, the MPU internally uses $[64, 64 + 64[$ to check memory accesses.

26.2 Algorithm

Thanks to the Memory Protection Unit and the privilege levels we can make sure that a process can only access its own memory region as follows:

- enable the MPU and the background region with the MPU Control Register. The background region, only enabled in privileged mode, allows the kernel to access any memory address, and in particular its own data structures.
- set the privilege level of Thread mode to unprivileged with the CONTROL register, so that processes run at this privilege level. This must not be done during the kernel initialization, which runs in Thread mode. Otherwise the kernel would loose its privileges. In particular, this would disable the background region, and the kernel might no longer be able to access its own data structures. Instead, this step can be done just before spawning the initial process, in the spawn system call handler.
- just before switching to a child process, or back to a parent process, configure the MPU registers to forbid any memory access outside this process's memory region, hereafter noted $[begin, end[$.

The first two steps are trivial to implement, but the last one is not. Indeed, we cannot simply configure an MPU region with base address $begin$ and size $end - begin$. This is because the base address must be a multiple of the region size, which must itself

be a power of 2 (larger than or equal to 32). For instance, $[begin, end[= [32, 128[$ is not a valid MPU region because 32 is not a multiple of $96 = 128 - 32$, and because 96 is not a power of 2. However, this interval can be *covered* with two MPU regions, namely $[32, 64[$ and $[64, 128[$. It can also be covered with subregions 1, 2, and 3 of the MPU region $[0, 256[$. This example shows that, in general, several MPU regions or subregions are needed to cover a given interval.

The problem is now to find a way to compute the base address, size, and enabled subregions of each MPU region so that they cover a given $[begin, end[$ interval. A prerequisite is that $begin$ and end are multiple of 32, since the smallest possible MPU region or subregion is 32 bytes. This is why we rounded these values to multiples of 32 in Chapter 23. If the MPU had a large enough number of regions we could then cover the $[begin, end[$ interval with many 32 bytes regions. But it has only 8. To solve this problem we need to use regions with the largest possible sizes.

Consider for instance an interval $I_0 = [32, \ldots[$, ignoring the end part for now. The largest region which can cover its beginning part is $[0, 256[$, with subregion $[0, 32[$ disabled. This leaves an interval $I_1 = [256, \ldots[$ to cover. The largest region which can cover its beginning part is $[0, 2048[$, with subregion $[0, 256[$ disabled. By repeating this process we then get the MPU regions $[0, 16\ \mathrm{KB}[$ and $[0, 128\ \mathrm{KB}[$, with their subregion 0 disabled. Note that the 4^{th} region is larger than the RAM (96 KB). Hence, with this method, we never need more than 4 regions to cover the beginning part of a process's memory interval (contained in RAM). The same method, applied to the end part, shows that the 4 remaining MPU regions are sufficient to cover the rest.

26.2.1 Definition

The above ideas can be formalized as follows. To cover the $[begin, end[$ interval, where $begin$ and end are multiples of $2^{level-3}$ (with $level \geq 8$ and $begin < end$):

- configure a region of size $s = 2^{level}$, with base address b_0 equal to $begin$ rounded down to a multiple of s, and with subregions covering $[begin, gap\_begin[$, where $gap\_begin = \min(b_0 + s, end) > begin$.
- configure a region of size s, with base address b_1 equal to end rounded up to a multiple of s, minus s, and with subregions covering $[gap\_end, end[$, where $gap\_end = \max(begin, b_1) < end$.
- if $gap\_begin < gap\_end$, repeat the above steps with $begin$, end, and $level$ replaced with gap_begin, gap_end, and $level + 3$, respectively.

The first two steps configure regions which satisfy the MPU constraints since, by construction, s is a power of 2, and b_0 and b_1 are multiples of s. Moreover, by construction too, gap_begin and gap_end are inside their respective region, and are multiples of $2^{level-3} = s/8$ (since $begin$, end, b_0, b_1, and s are). This ensures that $[begin, gap\_begin[$ and $[gap\_end, end[$ can be covered with subregions.

Finally, if $gap\_begin = end$ then $gap\_begin > gap\_end$ since $end > gap\_end$. Hence, if gap_begin is less than gap_end it is necessarily equal to $b_0 + s$, and is thus a multiple of s. A similar argument shows that, in this case, gap_end is necessarily a

multiple of s too. The third step above thus satisfies the required hypotheses to repeat the process with $level + 3$.

For an interval inside the 96 KB of RAM, and starting with $level = 8$, this process can be repeated at most 4 times. Indeed, a fifth repetition would imply a non-empty $[gap\_begin, gap\_end[$ region whose size is a multiple of $s = 2^{17} = 128$ KB. And this is not possible since this is larger than the RAM.

26.2.2 Example

To illustrate this, consider the case $[begin, end[= [672, 4448]$ (see Figure 26.1).

At the first iteration, with $level = 8$, the first step gives $s = 256$, $b_0 = \lfloor 672/256 \rfloor * 256 = 512$, and $gap\_begin = b_0 + s = 768$. We thus configure a region with base address 512, size 256, and with its last 3 subregions enabled so that they cover $[begin, gap\_begin[= [672, 768[$. The second step gives $b_1 = \lfloor (4448 + 255)/256 \rfloor * 256 - 256 = \lfloor (4448 - 1)/256 \rfloor * 256 = 4352$ and $gap\_end = b_1 = 4352$. We thus configure a second region with base address 4352, size 256, and with its first 3 subregions enabled so that they cover $[gap\_end, end[= [4352, 4448[$.

At the second iteration, with $begin = 768$, $end = 4352$, and $level = 11$, the first step gives $s = 2048$, $b_0 = \lfloor 768/2048 \rfloor * 2048 = 0$, and $gap\_begin = b_0 + s = 2048$. We thus configure a third region with base address 0, size 2048, and with its last 5 subregions enabled so that they cover $[begin, gap\_begin[= [768, 2048[$. The second step gives $b_1 = \lfloor (4352 - 1)/2048 \rfloor * 2048 = 4096$ and $gap\_end = b_1 = 4096$. We thus configure a fourth region with base address 4096, size 2048, and with its first subregion enabled so that it covers $[gap\_end, end[= [4096, 4352[$.

At the third and last iteration, with $begin = 2048$, $end = 4096$, and $level = 14$, the first step gives $s = 16384$, $b_0 = \lfloor 2048/16384 \rfloor * 16384 = 0$, and $gap\_begin = \min(b_0 + s, end) = 4096$. We thus configure a fifth region with base address 0, size 16384, and with its subregion 1 enabled so that it covers $[begin, gap\_begin[= [2048, 4096[$. The second step gives exactly the same region and subregion.

26.3 Implementation

The above algorithm uses the $\min$ and $\max$ mathematical functions. The former is already implemented in our kernel, but the latter is not. We also need a function to set the CONTROL register with a Move to Special Register from Register (MSR) instruction (see Section 23.2.2). We add them just after the `min` function:

```
fn max(x: u32, y: u32) -> u32 {
  if x > y { return x; } else { return y; }
}
```

```
fn set_control_register(value: u32) [
  /*MSR_CONTROL_R0*/ 2283074432;
  /*MOV_PC_LR*/ 18167;
]
```

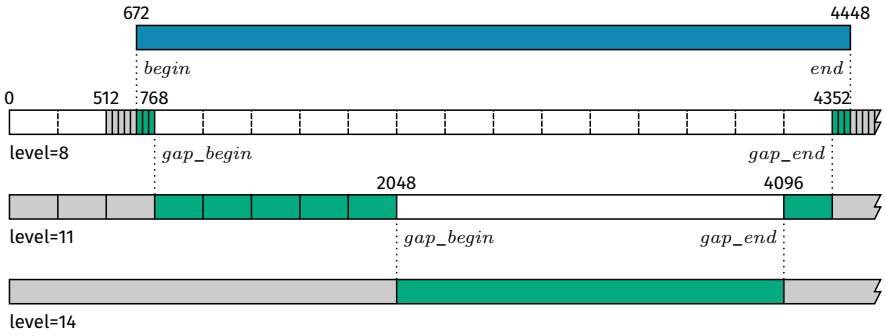


FIGURE 26.1 The MPU regions used to forbid any memory access outside the $[672, 4448[$ memory interval (blue). Enabled regions are shown with their enabled subregions in green, and their disabled subregions in gray.

The MPU regions must be configured when the current process is changed to a new one (a child or a parent). This is done in `os_set_current_process`. We thus implement the above algorithm just before this function.

The following function configures the MPU region whose number is id. It enables it, sets its size to $s = 2^{level}$, and sets its base address and subregions in order to cover the $[begin, end[$ interval. It assumes that $begin$ and end are multiple of $s/8$, that $begin > end$, and that $end - begin \leq s$. It sets the base address to $begin$ rounded down to a multiple of s, *i.e.*, to $\lfloor begin/s \rfloor * s = (begin \gg level) \ll level$. It then sets the attributes, size and disabled sub regions as follows:

- $access$ is set to 3 ("full access"), $attributes$ to 6 ("RAM"), and e to 1 ("enabled"), yielding 3060001_{16} = 50724865,
- $size$ is set to $level - 1$ (recall that the region size is 2^{size+1}),
- the disabled subregions are set to $255 - ((2^e - 1) - (2^b - 1)) = 255 - 2^e + 2^b = 255 - (1 \ll e) + (1 \ll b)$, where $[b, e[$ are the indices of the enabled subregions. For instance, if $b = 2$ and $e = 5$, this gives $2^e - 1 = 11111_2$, $2^b - 1 = 11_2$, $(2^e - 1) - (2^b - 1) = 11100_2$, and $255 - ((2^e - 1) - (2^b - 1)) = 11111111_2 - 11100_2 = 11100011_2$, which disables the subregions other than 2, 3, or 4.

```
fn mpu_set_region(id: u32, begin: u32, end: u32, level: u32) {
  const MPU_REGION_BASE_ADDRESS_REGISTER: &u32 = 3758157212;
  const MPU_REGION_ATTRIBUTE_AND_SIZE_REGISTER: &u32 = 3758157216;
  const RAM_FULL_ACCESS_ENABLED: u32 = 50724865;
  let base_address = (begin >> level) << level;
  begin = (begin - base_address) >> (level - 3);
  end = (end - base_address) >> (level - 3);
  *MPU_REGION_BASE_ADDRESS_REGISTER = base_address | 16 | id;
  *MPU_REGION_ATTRIBUTE_AND_SIZE_REGISTER = RAM_FULL_ACCESS_ENABLED |
      (255 - (1 << end) + (1 << begin)) << 8 | (level - 1) << 1;
}
```

We use this function in the following one, which performs one iteration of the algorithm in Section 26.2.1. It assumes that $*begin$ and $*end$ are multiple of $2^{level-3}$, and that $*begin > *end$. It configures the MPU regions with number id and $id+1$ to cover as much as possible of the $[*begin, *end[$ interval, and updates these variables to the (possibly empty) uncovered gap.

```
fn mpu_set_end_regions(id: u32, begin: &u32, end: &u32, level: u32) {
  let gap_begin = min(((*begin >> level) + 1) << level, *end);
  let gap_end = max(*begin, ((*end - 1) >> level) << level);
  mpu_set_region(id, *begin, gap_begin, level);
  mpu_set_region(id + 1, gap_end, *end, level);
  *begin = min(gap_begin, gap_end);
  *end = gap_end;
}
```

The next function calls it 4 times, with increasing $level$ values and distinct region numbers, to cover the whole $[begin, end[$ interval (whose bounds must be multiple of 32 and whose size must be less than 128 KB). Note that, for some of these calls, $begin$ and end might be equal. The above functions actually support this, even if this common value is not a multiple of $2^{level-3}$. In such cases, they give MPU regions whose all subregions are disabled. Thus, by always calling `mpu_set_end_regions` exactly 4 times, we make sure that unneeded regions are disabled (not doing so could keep unwanted regions configured for the previous process).

```
fn mpu_set_regions(begin: u32, end: u32) {
  mpu_set_end_regions(0, &begin, &end, 8);
  mpu_set_end_regions(2, &begin, &end, 11);
  mpu_set_end_regions(4, &begin, &end, 14);
  mpu_set_end_regions(6, &begin, &end, 17);
}
```

We finally call the above function in `os_set_current_process` to forbid all memory accesses outside the memory region of the new current process:

```
fn os_set_current_process(kernel: &Kernel, process: &Process) {
  set_process_stack_pointer(process.saved_context as &u32);
  mpu_set_regions(process.begin as u32, process.end as u32);
  kernel.current_process = process;
}
```

To make sure that this process cannot change the MPU registers to grant itself more access rights, we also need to set the privilege level of Thread mode to unprivileged. For that we set the p bit of the CONTROL register to 1 in `os_spawn`, just before spawning the initial process (which has no parent by definition):

```
  context.status_register = 1 << 24;
  if parent != null {
    parent.saved_context = get_process_stack_pointer() as &Context;
  } else {
    set_control_register(/*unprivileged*/1);
  }
```

Last but not least, we need to enable the MPU and the background region with the MPU Control Register. We do this at the end of the `os_init` function, but only if the user presses 'y':

```
  const SVC_HANDLER_PRIORITY_REGISTER: &u32 = 3758157084;
  const MPU_CONTROL_REGISTER: &u32 = 3758157204;
  *SVC_HANDLER_PRIORITY_REGISTER = 255 << 24;
  if keyboard_wait_char() == 'y' { *MPU_CONTROL_REGISTER = 5; }
}
```

Without this precaution, a bug in the above implementation could cause a crash in the kernel, the shell, the text editor or the compiler. We would then have no way to fix the error. Instead, with this precaution, and if a problem occurs, we can reboot with the MPU disabled and fix the issue.

26.4 Compilation and tests

Type "`edit toys.toy`" and Enter to edit the current kernel source code, and update it as described above. For reference, we also provide this code in the `toys_v3.txt` file in https://ebruneton.github.io/toypc/sources.zip. Then save it and compile it with "`toyc toys toys.toy`". Repeat these steps until the compilation is successful.

To test these changes we can introduce a voluntary bug in our "`hello`" program. Type "`edit hello.toy`" to edit it and update its main function as follows:

```
fn main() {
  write(1 /*standard output*/, HELLO, 13);
  const KERNEL_POINTER_REGISTER: &u32 = 1074666140;
  *KERNEL_POINTER_REGISTER = 0;
  exit(0);
}
```

Then type "`toyc hello hello.toy`" to compile it. This bug sets to 0 the General Purpose Backup Register containing the address of the `Kernel` data structure (see its definition in "`toys.toy`"). It could thus make the kernel unable to access its own data structures. To confirm this, type "`hello`" to run this program. The shell should become unresponsive, because the kernel crashed. Indeed, we haven't restarted it yet, and the MPU is thus not yet enabled.

Now reset the Arduino, and type "y" to enable the MPU. This should launch the shell, and you should be able to launch the text editor and compile programs. If not reset the Arduino again and type any key other than "y" to launch the kernel without MPU. Then repeat the steps from the beginning of this section.

Finally, type "`hello`" to run this program again. This time the MPU should trigger a Hard Fault when the process attempts to set the `KERNEL_POINTER_REGISTER` to 0. This should cause the kernel to terminate this process with an `INTERNAL_ERROR` status. In turn, the shell should print "`Hello, World!`" followed by the message "`hello crashed`". But the shell and the kernel should now continue to work.

At this stage we no longer need the possibility to disable the MPU when launching the kernel. Type "`edit toys.toy`" and replace the line

```
if keyboard_wait_char() == 'y' { *MPU_CONTROL_REGISTER = 5; }
```

with

```
*MPU_CONTROL_REGISTER = 5; /*Enable MPU and background region*/
```

Then recompile the kernel with "`toyc toys toys.toy`" and reset the Arduino. The shell should start without having to press any key.

CHAPTER

27 Utilities

As illustrated in the previous chapter, our operating system provides everything we need to edit, compile, and run programs, including its kernel. However, it is not very practical to use. For instance, to delete a file, one has to write a program doing this, compile it, and run it. This is a lot of work for a simple task. Fortunately, once this is done, this program can be reused to delete any file (provided it takes the name of the file to delete as command line argument). For instance, with such a program named "`delete`" we can simply type "`delete foo.txt`" in the shell to delete "`foo.txt`". This chapter provides this program, as well as a few others performing other simple tasks such as copying a file or listing all the files. It also provides a few small compiler and shell improvements.

27.1 Split

The above utility programs need to parse their command line arguments and to make system calls. We improved the compiler in Chapter 25 so that the functions doing this could be stored in a separate source file, shared by several programs. It now remains to create this shared file, before implementing the programs using it.

The shell already contains the code we want to put in this file, namely the `entry` function, the `sh_read_token` function, and the system call functions. It also contains other functions which could be shared, such as those drawing text on the screen, or allocating and copying memory. Unfortunately we have no way to copy paste text, to avoid rewriting this code from scratch in new files. We could implement such a functionality, but a much simpler method is to write a program to split a file in several parts. This is what we do in this section.

27.1.1 Requirements

The "`split`" program must take as argument the name of the file to split. This file must contain the name and content of each part, separated by "~". For instance

```
file1.txt~lorem ipsum
dolor sit amet
~file2.txt~consectetur
adipiscing elit
```

should be split in a "`file1.txt`" file containing

```
lorem ipsum
dolor sit amet
```

and a "`file2.txt`" file containing

```
consectetur
adipiscing elit
```

Furthermore, the original file should remain unchanged.

27.1.2 Implementation

In order to implement this program we need to rewrite one last time some of the functions we want to share (but not all of them, making this program still worth the effort – see their description in the previous chapters):

```
fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32;
fn exit(result: u32) -> u32;

fn entry(heap: &u32, heap_limit: &u32) {
  let args = heap + 4;
  let args_end = args + *heap;
  heap = (((args_end as u32 + 3) >> 2) << 2) as &u32;
  exit(main(args, args_end, heap, heap_limit));
}

fn load8(ptr: &u32) -> u32 [ /*LDRB_R0_R0_0*/30720; /*MOV_PC_LR*/18167; ]

fn sh_read_token(src_p: &&u32, src_end: &u32, length: &u32) -> &u32 {
  let src = *src_p;
  while src < src_end && load8(src) == ' ' { src = src + 1; }
  if src >= src_end { return null; }
  let token = src;
  while src < src_end && load8(src) != ' ' { src = src + 1; }
  *length = src - token;
  *src_p = src;
  return token;
}

const OK: u32 = 0;
const INVALID_ARGUMENT: u32 = 1;
const OUT_OF_MEMORY: u32 = 5;
fn status(result: u32) -> u32 { return result >> 24; }

fn system_call(id: u32, args: &u32) -> u32 [
  /*SVC*/ 57088;
  /*MOV_PC_LR*/ 18167;
]
```

```
fn exit(result: u32) -> u32 {
  return system_call(1, &result);
}
fn open(name: &u32, length: u32, mode: u32) -> u32 {
  return system_call(4, &name as &u32);
}
fn read(stream_id: u32, buffer: &u32, size: u32) -> u32 {
  return system_call(5, &stream_id);
}
fn write(stream_id: u32, buffer: &u32, size: u32) -> u32 {
  return system_call(6, &stream_id);
}
fn close(stream_id: u32) -> u32 {
  return system_call(7, &stream_id);
}
```

We can then implement the actual functionality of this program. It is simple enough to be entirely included in the `main` function. We start by getting the name of the file to split from the command line arguments, and by reading as many bytes as possible from this file. We return an error if the file name argument is missing, if the file cannot be opened or read, or cannot be read entirely (supporting the case of files which do not fit in RAM is more complex and not needed):

```
fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  let length = 0;
  let name = sh_read_token(&args, args_end, &length);
  if name == null { return INVALID_ARGUMENT; }

  let stream = open(name, length, 'r');
  if status(stream) != OK { return status(stream); }
  length = read(stream, heap, heap_limit - heap);
  close(stream);
  if status(length) != OK { return status(length); }
  if length == heap_limit - heap { return OUT_OF_MEMORY; }
```

We then use a loop to split each part. At the beginning of each iteration, `src` and `ptr` point to the first character of the name of the next file to create. Then `ptr` is advanced to the next "~". The file name is thus between `src` and `ptr` (excluded), and a stream is opened to write into this file. Finally, `src` and `ptr` are advanced to the character after the "~", `ptr` is advanced to the next "~" again, and the content, between `src` and `ptr` (excluded), is written to the stream:

```
  let src = heap;
  let src_end = src + length;
  let ptr = src;
  while src < src_end {
    while ptr < src_end && load8(ptr) != '~' { ptr = ptr + 1; }
    if ptr == src_end { return INVALID_ARGUMENT; }
    stream = open(src, ptr - src, 'w');
```

```
    if status(stream) != OK { return status(stream); }
    src = ptr + 1;
    ptr = src;
    while ptr < src_end && load8(ptr) != '~' { ptr = ptr + 1; }
    length = write(stream, src, ptr - src);
    close(stream);
    if status(length) != OK { return status(length); }
    src = ptr + 1;
    ptr = src;
  }
  return OK;
}
```

27.1.3 Compilation and tests

Type "`edit split.toy`" and enter the above source code. Then save it and compile it with "`toyc split split.toy`". If necessary, repeat these steps until the compilation is successful. To split the shell source code, first type "`edit shell.toy`" and edit it as follows:

```
src/base.toy~fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u
                                                              u32) -> u32;
...
}
~src/gpu.toy~fn gpu_set_register(id: u32, value: u32) {
...
fn gpu_draw_char(c: u32) { gpu_set_register(2, c); }
~src/memory.toy~fn mem_allocate(size: u32, heap_p: &&u32, heap_limit: &u32
                                                            ) -> &u32 {
...
}
~src/shell/shell.toy~const NUM_COMMANDS: u32 = 4;
...
```

Then type "`split shell.toy`" to actually split it. This should create three files "`src/base.toy`", "`src/gpu.toy`", and "`src/memory.toy`", intended to be shared, plus a "`src/shell/shell.toy`" file for the shell itself.

To check that this worked, type "`toyc shell2 src/base.toy src/gpu.toy src/memory.toy src/shell/shell.toy`" to compile a new shell program from these parts. There should be no error, and typing "`shell2 test`" should launch a new shell, printing "`test`".

The beginning of the text editor contains the same code as in the "`src/base.toy`" and "`src/gpu.toy`" files above. We can thus delete it and use these files instead, to reduce the total amount of code. For this the fastest way is to split the text editor source code in two parts and simply discard the first part. To this end, type "`edit edit.toy`" and edit this code as follows:

```
trash.txt~fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32⸮
                                                              ⸮) -> u32;
...
fn gpu_draw_char(c: u32) { gpu_set_register(2, c); }
~src/edit/edit.toy~fn mem_copy(src: &u32, dst: &u32, n: u32) {
...
```

Then type "`split edit.toy`" to actually split it. To check that this worked, type "`toyc edit2 src/base.toy src/gpu.toy src/edit/edit.toy`" to compile a new text editor program from these parts. There should be no error, and typing "`edit2 src/edit/edit.toy`" should open the new text editor code.

27.2 Reboot

We can now implement some utility programs to make our operating system easier to use. We start here with a program to reboot the computer. We don't really need one to reboot the operating system – we can simply use the RESET button instead – but rebooting with the Boot Assistant is useful. For instance, this can be used to make a copy of the Flash memory on an external computer, to save work done on the Arduino. The following program uses the `reboot` system call to reboot with the operating system or with the Boot Assistant (if run with the "`-rom`" command line argument). Thanks to the shared files created above we only need to implement the `main` function, which should be self-explanatory:

```
static USAGE = ['U','s','a','g','e',':',' ',
  'r','e','b','o','o','t',' ','[','-','r','o','m',']'];

static ROM = ['-','r','o','m'];

fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  let length = 0;
  let argument = sh_read_token(&args, args_end, &length);
  let mode = 0; /*Flash*/
  if argument != null {
    if length != 4 || *argument != *ROM {
      write(STANDARD_OUTPUT, USAGE, 20);
      return INVALID_ARGUMENT;
    }
    mode = 1; /*ROM*/
  }
  return status(reboot(mode));
}
```

Type "`edit src/reboot/reboot.toy`", enter this code, save it, and compile it with "`toyc reboot src/base.toy src/reboot/reboot.toy`". To test it, type "`reboot`", and then "`reboot -rom`". The system should become unresponsive, because the Boot Assistant should now be running instead of the operating system. To reboot

again with the operating system, run the following command on an external computer (see Section 9.7):

```
user@host:~$ python3 flash_helper.py
>reset#
```

27.3 Delete

The "delete" utility deletes the file whose name is given as command line argument. It is as simple as the previous one:

```
static USAGE = ['U','s','a','g','e',':',' ',
  'd','e','l','e','t','e',' ','f','i','l','e'];

static CANT_DELETE = ['C','a','n',''','t',' ','d','e','l','e','t','e',' '];

fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  let length = 0;
  let name = sh_read_token(&args, args_end, &length);
  if name == null {
    write(STANDARD_OUTPUT, USAGE, 18);
    return INVALID_ARGUMENT;
  }
  let result = status(delete(name, length));
  if result != OK {
    write(STANDARD_OUTPUT, CANT_DELETE, 13);
    write(STANDARD_OUTPUT, name, length);
  }
  return result;
}
```

Type "edit src/delete/delete.toy", enter this code, save it, and compile it with "toyc delete src/base.toy src/delete/delete.toy". To test it, type "delete trash.txt" to delete the "trash.txt" file produced above while splitting the text editor. Typing this command again should give an error because the file no longer exists.

27.4 Copy

The "copy" utility copies a file into another. It takes as command line arguments the names of these files, starting with the source one. It reads as many bytes as possible from the source file, writes them to the destination file, and repeats this until all bytes of the source file are read. It thus supports files which do not fit in RAM. Its source code is longer than that of the previous utility programs, but should still be easy to understand:

```
fn write_error(src1: &u32, length1: u32, src2: &u32, length2: u32, result:↩
                                                                ↩ u32) -> u32 {
```

```
  write(STANDARD_OUTPUT, src1, length1);
  write(STANDARD_OUTPUT, src2, length2);
  return status(result);
}

static USAGE = ['U','s','a','g','e',':',' ',
  'c','o','p','y',' ','s','r','c',' ','d','s','t'];

static CANT_OPEN = ['C','a','n','''','t',' ','o','p','e','n',' '];
static CANT_READ = ['C','a','n','''','t',' ','r','e','a','d',' '];
static CANT_WRITE = ['C','a','n','''','t',' ','w','r','i','t','e',' '];

fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  let src_length = 0;
  let dst_length = 0;
  let src_name = sh_read_token(&args, args_end, &src_length);
  let dst_name = sh_read_token(&args, args_end, &dst_length);
  if src_name == null || dst_name == null {
    write(STANDARD_OUTPUT, USAGE, 19);
    return INVALID_ARGUMENT;
  }
  let buffer = heap;
  let buffer_size = heap_limit - heap;
  if buffer_size < 64 { return OUT_OF_MEMORY; }

  let src_stream = open(src_name, src_length, 'r');
  if status(src_stream) != OK {
    return write_error(CANT_OPEN, 11, src_name, src_length, src_stream);
  }
  let dst_stream = open(dst_name, dst_length, 'w');
  if status(dst_stream) != OK {
    return write_error(CANT_OPEN, 11, dst_name, dst_length, dst_stream);
  }
  let n = 0;
  let result = 0;
  loop {
    n = read(src_stream, buffer, buffer_size);
    if status(n) != OK {
      return write_error(CANT_READ, 11, src_name, src_length, n);
    }
    result = write(dst_stream, buffer, n);
    if status(result) != OK {
      return write_error(CANT_WRITE, 12, dst_name, dst_length, result);
    }
    if n < buffer_size { return OK; }
  }
}
```

Type "edit src/copy/copy.toy", enter this code, save it, and compile it with "toyc copy src/base.toy src/copy/copy.toy". To test it, type "copy src/copy/copy.toy test.txt" to copy its own source code to a new file. Check that this worked with "edit test.txt".

27.5 List

The "list" utility prints the name of each file and directory in a given "directory". Here we call "directory" a component of a file name before a slash. For instance, the file "src/copy/copy.toy" is considered as a "copy.toy" file inside the "copy" directory, itself inside the "src" directory, itself inside the *root* directory (whose name is empty). Thus, for instance, listing the files and directories in the root directory should give "src", but not "src/copy" or "src/copy/copy.toy" (since "copy" and "copy.toy" are not inside the root directory). Similarly, listing the files and directories in "src" should give "copy", but not "src" or "copy/copy.toy".

We start this utility with the "mem_compare" function, already used in the compiler, and needed later on (an alternative is to add it in "src/memory.toy"):

```
fn mem_compare(ptr1: &u32, ptr2: &u32, size: u32) -> u32 {
  let i = 0;
  while i < size && load8(ptr1 + i) == load8(ptr2 + i) {
    i = i + 1;
  }
  return size - i;
}
```

The next function computes the length of the first component of a file name, *i.e.*, until the first slash (included), if there is one. For instance, it returns 4 for "src/copy/copy.toy", whose first component is "src/" (we include the slash to distinguish directory names from file names). It returns 8 for "copy.toy", which has only one component.

```
fn get_first_component_length(name: &u32, length: u32) -> u32 {
  let i = 0;
  while i < length && load8(name + i) != '/' { i = i + 1; }
  if i < length { i = i + 1; }
  return i;
}
```

The main function starts by checking that the command line argument, if any, is a directory *path*. That is, a list of directory names separated by slashes and ending with a slash (such as "src/copy/"):

```
static USAGE = ['U','s','a','g','e',':',' ',
  'l','i','s','t',' ','[','p','a','t','h','/',']'];

static SEPARATOR = [' ',' '];
```

```
fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  let path_length = 0;
  let path = sh_read_token(&args, args_end, &path_length);
  if path_length > 0 && load8(path + path_length - 1) != '/' {
    write(STANDARD_OUTPUT, USAGE, 19);
    return INVALID_ARGUMENT;
  }
```

It then uses a loop to get the (full) name of each file in the file system, one by one (by reading the root directory – see Section 24.3.4). If this full name starts with the path given as argument, it then gets the length of the first component of the rest of this name. Finally, if this component has not already been written during the last iteration, it writes it to standard output.

As an example, let's assume that the input path is "`src/`" and that there are 4 files "`copy`", "`src/base.toy`", "`src/copy/copy.toy`", and "`src/copy/help.txt`". The above loop skips the first because it does not start with the input path. It writes "`base.toy`" for the second one (the first component after the input path), and "`copy/`" for the third. And it does nothing for the fourth one because "`copy/`" has already been written at the previous iteration.

```
  const MAX_NAME_LENGTH: u32 = 256;
  let name = mem_allocate(MAX_NAME_LENGTH, &heap, heap_limit);
  let last_item = mem_allocate(MAX_NAME_LENGTH, &heap, heap_limit);
  if name == null || last_item == null { return OUT_OF_MEMORY; }

  let stream = open(name, 0, 'r');
  let length = read(stream, name, MAX_NAME_LENGTH);
  let last_length = 0;
  let item: &u32 = null;
  while length != 0 {
    if length > path_length && mem_compare(name, path, path_length) == 0 {
      item = name + path_length;
      length = get_first_component_length(item, length - path_length);
      if length != last_length || mem_compare(item, last_item, length) != ≀
                                                                        ≀0 {
        write(STANDARD_OUTPUT, item, length);
        write(STANDARD_OUTPUT, SEPARATOR, 2);
        mem_copy_non_overlapping(item, last_item, length);
        last_length = length;
      }
    }
    length = read(stream, name, MAX_NAME_LENGTH);
  }
  return OK;
}
```

Type "`edit src/list/list.toy`", enter the above code, save it, and compile it with "`toyc list src/base.toy src/memory.toy src/list/list.toy`". To test it, type "`list`", "`list src/`", or "`list dst/`", for example.

27.6 Stat

Our last utility program, "stat", gives some statistics about the file whose name is given as command line argument. More precisely, it prints the number of bytes and the number of lines of this file. It supports files which do not fit in RAM by reading them in several steps if necessary.

We start with a function counting the number of lines between src and src_end (excluded). This function counts the number of "new line" characters. More precisely, to avoid counting a "new line" at the very end of a file as one line, it counts the number of characters which are immediately preceded by a "new line". This requires knowing the character just before src, for files read in several steps. This is the purpose of *last_char which, on return, contains the last character before src_end:

```
const NEW_LINE: u32 = 10;

fn line_count(src: &u32, src_end: &u32, last_char: &u32) -> u32 {
  let count = 0;
  while src < src_end {
    if *last_char == NEW_LINE { count = count + 1; }
    *last_char = load8(src);
    src = src + 1;
  }
  return count;
}
```

The next functions, copied from our compiler (see Section 25.3.3), are needed to write the file statistics as decimal numbers, or an error message if an error occurs:

```
fn write_integer(x: u32) {
  let quotient = x / 10;
  x = x - 10 * quotient + '0';
  if quotient > 0 { write_integer(quotient); }
  write(STANDARD_OUTPUT, &x, 1);
}
fn write_error(src1: &u32, length1: u32, src2: &u32, length2: u32, result:↩
                                                                ↩ u32) -> u32 {
  write(STANDARD_OUTPUT, src1, length1);
  write(STANDARD_OUTPUT, src2, length2);
  return status(result);
}
```

The main function reads as many bytes as possible from the source file, adds the number of bytes (resp. lines) in this chunk to the total number of bytes (resp. lines), and repeat these steps until the whole file is read:

```
static USAGE = ['U','s','a','g','e',':',' ',
  's','t','a','t',' ','f','i','l','e'];

static CANT_OPEN = ['C','a','n','''','t',' ','o','p','e','n',' '];
```

```
static CANT_READ = ['C','a','n',''','t',' ','r','e','a','d',' '];
static BYTES = [' ','b','y','t','e','s',' '];
static LINES = [' ','l','i','n','e','s'];

fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  let length = 0;
  let name = sh_read_token(&args, args_end, &length);
  if name == null {
    write(STANDARD_OUTPUT, USAGE, 16);
    return INVALID_ARGUMENT;
  }
  let buffer = heap;
  let buffer_size = heap_limit - heap;
  if buffer_size < 64 { return OUT_OF_MEMORY; }

  let stream = open(name, length, 'r');
  if status(stream) != OK {
    return write_error(CANT_OPEN, 11, name, length, stream);
  }
  let n = 0;
  let bytes = 0;
  let lines = 0;
  let last_char = NEW_LINE;
  loop {
    n = read(stream, buffer, buffer_size);
    if status(n) != OK {
      return write_error(CANT_READ, 11, name, length, n);
    }
    bytes = bytes + n;
    lines = lines + line_count(buffer, buffer + n, &last_char);
    if n < buffer_size { break; }
  }
  write_integer(bytes);
  write(STANDARD_OUTPUT, BYTES, 7);
  write_integer(lines);
  write(STANDARD_OUTPUT, LINES, 6);
  return OK;
}
```

Type "`edit src/stat/stat.toy`", enter the above code, save it, and compile it with "`toyc stat src/base.toy src/stat/stat.toy`". To test it, type "`stat src/stat/stat.toy`", for example.

27.7 Compiler improvements

Our compiler starts with the same functions as in the shared "`src/base.toy`" file (we added them in Section 25.3.3). To avoid this duplicated code we can remove them

and use the shared file instead. For this the easiest way is to split the compiler source code and to discard the first part, as we did for the text editor. To do this, and to split the second part in smaller, more manageable files, type edit "`edit toyc.toy`" and edit this code as follows:

```
trash.txt~fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32
                                                              ) -> u32;
...
}
~src/toyc/base.toy~fn panic_result() -> u32 [
...
}
~src/toyc/scanner.toy~static TC_CHAR_TYPES = [
...
}
~src/toyc/backend.toy~fn mem_allocate(size: u32, ptr_p: &&u32, ptr_limit:
                                                          &u32) -> &u32 {
...
}
~src/toyc/parser.toy~fn sym_lookup(symbol: &Symbol, name: &u32, length: u3
                                                          2) -> &Symbol {
...
}
~src/toyc/toyc.toy~fn write_integer(x: u32) {
...
```

Finally, type "`split toyc.toy`" to actually split it. We can now take advantage of this to reduce the amount of RAM required by the compiler. Indeed, the compiler currently loads all its input files in RAM, next to each other (Figure 25.2). However, after a source file has been compiled it is no longer needed, and we could delete it from RAM before loading the next one. This would reduce the memory need to the size of the largest source file, instead of their total size. But there is a catch: the symbol names are pointers to the source code (see Figure 16.1). And they are still needed after a file has been compiled (so that its symbols can be used in the next files). To solve this issue we need to copy the symbol names somewhere else before discarding the source code.

For this, type "`edit src/toyc/parser.toy`" and add the following function at the end of this file. This function copies the names of each symbol in the list starting with `symbol`, up to `end_symbol` (excluded). It copies each name in the compiler's heap. It also copies the names of the "nested" symbols, such as the field names in a `struct` symbol (see Figure 19.1). Note that the parameter names in a `fn` symbol (see Figure 19.2) are no longer needed after the corresponding function has been compiled.

```
fn tc_copy_symbol_names(self: &Compiler, symbol: &Symbol, end_symbol: &Sym
                                                              bol) {
  let name_copy: &u32 = null;
  let i = 0;
  while symbol != end_symbol {
```

```
    name_copy = mem_allocate(symbol.length, &self.heap, self.heap_limit);
    i = 0;
    while i < symbol.length {
      store8(name_copy + i, load8(symbol.name + i));
      i = i + 1;
    }
    symbol.name = name_copy;
    if symbol.kind == SYM_STRUCT {
      tc_copy_symbol_names(self, symbol.type, null);
    }
    symbol = symbol.next;
  }
}
```

Then type "`edit src/toyc/toyc.toy`" and implement the above idea by updating the `main` function as follows:

```
fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
...
  const MAX_CODE_SIZE: u32 = 12288;
  const MAX_HEAP_SIZE: u32 = 24576;
  const MIN_SRC_SIZE: u32 = 256;
...
  let stream = 0;
  let src_size = 0;
  let last_copied_symbol = compiler.symbols;
  while in != null {
...
    in = sh_read_token(&args, args_end, &in_length);
    tc_copy_symbol_names(compiler, compiler.symbols, last_copied_symbol);
    last_copied_symbol = compiler.symbols;
  }
...
}
```

That is, remove the "`src = src + src_size;`" statement so that each new source file is loaded "on top" of the previous one. And replace it with a call to the above function to copy the symbol names not already copied in previous iterations. Also, to make space for the copied names in the `Compiler`'s heap, increase its size from 18 to 24 KB.

To test these changes, type "`toyc toyc2 src/base.toy src/toyc/base.toy src/toyc/scanner.toy src/toyc/backend.toy src/toyc/parser.toy src/toyc /toyc.toy`" to recompile the compiler from these new source files, into a new program. There should be no error. Finally, to check that "`toyc2`" works correctly, use it to recompile itself with "`toyc2 toyc src/base.toy src/toyc/base.toy src/toyc/scanner.toy src/toyc/backend.toy src/toyc/parser.toy src/toyc/toyc.toy`". You can then use "`delete`" to delete the "`toyc.toy`", "`trash.txt`", and "`toyc2`" files, no longer needed.

27.8 Shell improvements

The command line used at the end of the previous section is quite long to type and can be hard to remember. Saving it in a file can solve the latter issue, but not the former (since we cannot copy paste text). To solve this we improve the shell so that it can run commands stored in a file, called a *script*.

More precisely, we add an optional command line argument to the shell. If present, this argument should be the name of a file containing some commands, one per line. To support commands longer than 100 characters, which is the maximum line length in the text editor, these lines can be wrapped with backslash characters. For instance,

```
copy edit edit.old
toyc edit src/base.toy\
 src/edit/edit.toy
```

corresponds to two commands, namely "`copy edit edit.old`" and "`toyc edit src/base.toy src/edit/edit.toy`".

To implement this new feature, type "`edit src/shell/shell.toy`" and edit this code as follows. First add new messages for errors related to the script file:

```
static CANT_FIND = ['C','a','n',''','t',' ','f','i','n','d',' '];
static CANT_OPEN = ['C','a','n',''','t',' ','o','p','e','n',' '];
static CANT_READ = ['C','a','n',''','t',' ','r','e','a','d',' '];
```

Then add the following function, after "`sh_run`". This function reads the script file whose name is given as argument and runs its commands with `sh_run`. It reads the file one character at a time, and unwraps each command into the same `src` buffer (from one of the shell's `Command` struct – see Figure 25.1). When the end of the file is reached, or a "new line" not preceded by a backslash, it runs the command between `src` and `src_end` (excluded) and then clears the buffer. Otherwise, it appends each new character to the buffer, unless it is a "new line" preceded by a backslash (in which case the backslash is deleted). It ends by writing to standard output the text written in the shell's output buffer by all the executed commands.

```
fn sh_run_script(self: &Shell, name: &u32, length: u32) -> u32 {
  let stream = open(name, length, 'r');
  if status(stream) != OK {
    write(STANDARD_OUTPUT, CANT_OPEN, 11);
    write(STANDARD_OUTPUT, name, length);
    return status(stream);
  }
  let src = &self.commands.data;
  let src_end = src;
  let c = 0;
  let n = 0;
  loop {
    n = read(stream, &c, 1);
    if status(n) != OK {
```

```
      sh_print(self, CANT_READ, 11);
      sh_print(self, name, length);
      break;
    } else if n == 0 {
      sh_run(self, src, src_end);
      break;
    } else if c == NEW_LINE {
      if src_end > src && load8(src_end - 1) == '\' {
        src_end = src_end - 1;
      } else {
        sh_run(self, src, src_end);
        src_end = src;
      }
    } else if src_end - src < MAX_COMMAND_LENGTH {
      store8(src_end, c);
      src_end = src_end + 1;
    }
  }
  write(STANDARD_OUTPUT, self.output_begin, self.output_end - self.output_
                                                                  begin);
  return status(n);
}
```

Finally, update the `main` function to call "`sh_run_script`" if the shell is launched with a command line argument, or the interactive command editor otherwise. Also add some code to initialize the shell's output buffer with the content of a "banner" file, if it exists, in order to display a welcome message in interactive mode:

```
static BANNER = ['s','r','c','/','s','h','e','l','l','/',
  'b','a','n','n','e','r','.','t','x','t'];

fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  let shell = sh_new(&heap, heap_limit);
  if shell == null { return OUT_OF_MEMORY; }
  let length = 0;
  let name = sh_read_token(&args, args_end, &length);
  if name != null {
    return sh_run_script(shell, name, length);
  }
  let stream = open(BANNER, 20, 'r');
  let n = 0;
  if status(stream) == OK {
    n = read(stream, shell.output_begin, MAX_OUTPUT_SIZE);
    if status(n) == OK { shell.output_end = shell.output_begin + n; }
    close(stream);
  }
  return sh_run_editor(shell);
}
```

Once this is done compile this new shell version with "`toyc shell src/base.toy src/memory.toy src/gpu.toy src/shell/shell.toy`". Then create a banner file by typing "`edit src/shell/banner.txt`" and enter the following text (we name our operating system "Toys", as its kernel file):

```
Welcome to
  _____
 |_   _|__  _   _ ___
   | |/ _ \| | | / __|
   | | (_) | |_| \__ \
   |_|\___/ \__, |___/
            |___/

Type 'list' for a list of available commands.
```

To test these changes, type "`shell`" to start a new shell. This should display the above banner before the prompt. Type Escape to return in the initial shell, and then create the following test script with "`edit script.sh`":

```
list src/
list\
 src/toyc/
```

Finally, run this script with "`shell script.sh`". This should list the content of the "`src`" and "`src/toyc`" directories.

27.9 Final steps

To finalize our operating system we can split the kernel source code (to take advantage of our compiler improvements) and delete the files which are no longer needed. For this type "`edit toys.toy`", edit this file as follows:

```
src/toys/drivers.toy~fn os_init(code: &u32, heap: &u32, stack: &u32);
...
}
~src/toys/filesystem.toy~struct DiskBlock {
...
}
~src/toys/processes.toy~struct Context {
...
}
~src/toys/systemcalls.toy~fn os_sleep(millis: u32) -> u32 {
...
}
~src/toys/toys.toy~fn hard_fault_handler() {
...
```

and type "`split toys.toy`" to split it. To check that everything is correct, type "`edit src/toys/BUILD`", enter the following script:

```
toyc toys src/toys/drivers.toy src/toys/filesystem.toy\
 src/toys/processes.toy src/toys/systemcalls.toy src/toys/toys.toy
```

and type "`shell src/toys/BUILD`" to run it, *i.e.*, to recompile the kernel from the split files. There should be no error. We can also add a build script for the compiler, and in fact for each program, since we improved the shell precisely for this. To this end, create the following files with the text editor:

`src/toyc/BUILD`:

```
toyc toyc src/base.toy src/toyc/base.toy src/toyc/scanner.toy\
 src/toyc/backend.toy src/toyc/parser.toy src/toyc/toyc.toy
```

`src/copy/BUILD`:

```
toyc copy src/base.toy src/copy/copy.toy
```

`src/delete/BUILD`:

```
toyc delete src/base.toy src/delete/delete.toy
```

`src/edit/BUILD`:

```
toyc edit src/base.toy src/gpu.toy src/edit/edit.toy
```

`src/list/BUILD`:

```
toyc list src/base.toy src/memory.toy src/list/list.toy
```

`src/reboot/BUILD`:

```
toyc reboot src/base.toy src/reboot/reboot.toy
```

`src/shell/BUILD`:

```
toyc shell src/base.toy src/gpu.toy src/memory.toy src/shell/shell.toy
```

`src/stat/BUILD`:

```
toyc stat src/base.toy src/stat/stat.toy
```

For consistency with the other source files, copy "`boot.toy`" to the `src` directory with "`copy boot.toy src/boot/boot.toy`" and create a script to compile it:

`src/boot/BUILD`:

```
toyc boot src/boot/boot.toy
```

Finally, delete all the files which are no longer needed: `boot.toy`, `edit.toy`, `edit2`, `hello`, `hello.toy`, `script.sh`, `shell.toy`, `shell2`, `split`, `split.toy`, `test.txt`, and `toys.toy`. After this the file system content should be the one in Table 27.1 (the file sizes can differ if you used some spaces instead of tabulations, for instance). For reference, these files are also provided in the companion website of this book (https://ebruneton.github.io/toypc/).

Name	Size	Lines
src/base.toy	2264	87
src/gpu.toy	1488	49
src/memory.toy	348	14
src/boot/BUILD	28	
src/boot/boot.toy	1489	67
src/copy/BUILD	41	
src/copy/copy.toy	1619	48
src/delete/BUILD	47	
src/delete/delete.toy	568	19
src/edit/BUILD	53	
src/edit/edit.toy	5727	225
src/list/BUILD	56	
src/list/list.toy	1621	51
src/reboot/BUILD	47	
src/reboot/reboot.toy	496	18
src/shell/BUILD	71	
src/shell/banner.txt	176	
src/shell/shell.toy	6580	231
src/stat/BUILD	41	
src/stat/stat.toy	1827	67
src/toyc/BUILD	123	
src/toyc/backend.toy	7708	247
src/toyc/base.toy	1601	77
src/toyc/parser.toy	21873	760
src/toyc/scanner.toy	4245	148
src/toyc/toyc.toy	3067	92
src/toys/BUILD	123	
src/toys/drivers.toy	9032	269
src/toys/filesystem.toy	7529	273
src/toys/processes.toy	5564	196
src/toys/systemcalls.toy	7244	240
src/toys/toys.toy	2919	103
Total	95615	3281

Name	Size
copy	744
delete	476
edit	2404
list	916
reboot	464
shell	2414
stat	830
toyc	9926
toys	6822
Total	24996

TABLE 27.1 The final content of the file system, containing the complete source code (left) and compiled code (right) of the Toys operating system. To which the compiled boot loader code, not part of the file system, must be added (less than 256 bytes).

CHAPTER

28 Snake Game

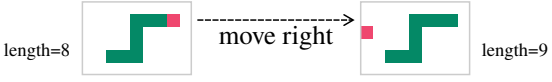
The Toys operating system is now complete. It can of course be improved in numerous ways, but this book is already quite long. Thus, instead of doing this, and since toys are made to play, we use it in this last chapter to implement a small "snake" game.

28.1 Requirements

In this game the player uses the arrow keys to control a "snake" moving at constant speed across the screen. The goal is to "eat" as many "apples" as possible. Each time the snake eats one, its get longer and a new apple appears at random in a free spot.

The snake is represented with a contiguous series of green square cells, from head to tail. The apple is represented with a red cell (there is only one apple at any given time). At each time step the head moves in the same direction as in the previous step or, if an arrow key was pressed, in the arrow direction. The body follows it, as illustrated in the following example:

If the head hits a "wall" (the screen boundary) or the snake's own body, the game is over. If the head moves in the apple cell the tail does not move, which increases the snake's length:



Then a new apple appears at random *in a free cell*. If the player is extremely good there might be no free cell left, in which case the game is over. The score is the number of apples eaten.

28.2 Data structures

In order to implement the above requirements we need to store in memory the current state of the game. For this we start by dividing the screen in a grid of cells. Each

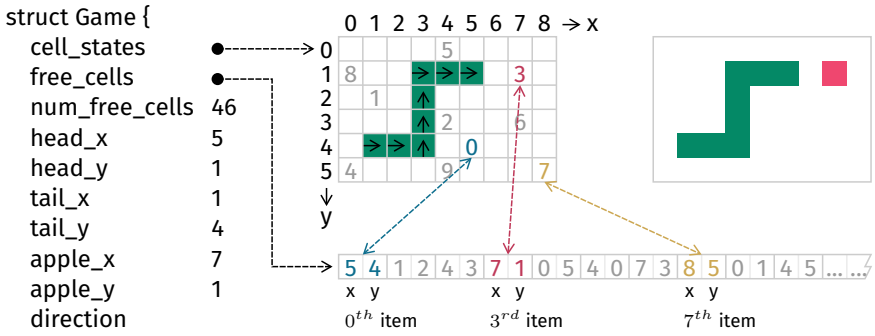


FIGURE 28.1 The game state on the right is represented with the struct on the left, containing pointers to the cell states and the free cells, and the (x, y) coordinates of the head, tail and apple. The state of a free cell is its index in the free cells list. This list contains the (x, y) coordinates of each free cell. The state of a used cell is the direction towards the next used cell.

cell is uniquely identified by its (x, y) coordinates, with x increasing from left to right and y from top to bottom (see Figure 28.1). We can then store the current apple position with its grid coordinates. Storing the current snake state is more complex. One possibility is to store a list of (x, y) coordinates, representing the cells occupied by the snake, from head to tail. We can then move the snake by adding a new element at the beginning of the list, and by removing the last element (unless the snake eats the apple). However, this method has two drawbacks:

1. to determine if the head hits the body we need to iterate over each element of the list, and check whether it is equal to the head.
2. there is no easy way to find a free cell to place a new apple. The best we can do is try a cell c at random, and then iterate over each snake cell to check if c is free or not. If not, we need to repeat this process until a free cell is found.

To mitigate this we can use an additional grid data structure, storing the current state of each cell: used or free. This state is easy to update when moving the snake, and avoids a loop over the snake's list of cells in 1) and 2). But this is not sufficient to avoid trying cells at random until a free one is found.

To solve this issue we introduce a third data structure, namely a list of all the free cells, identified by their (x, y) coordinates. Finding a free cell is then very easy: just pick any element in this list at random. But we then need to remove this cell from the list of free cells. We also need to update this list when moving the snake. For instance, we need to remove the new head cell from the list of free cells. This requires finding a cell in this list, given its coordinates. To do this without iterating over this list (we are trying to avoid any loop), we can store more than a "used or free" state for each cell. More precisely, we can store for each free cell its index in the list of free cells. This gives two data structures referencing each other (see Figure 28.1):

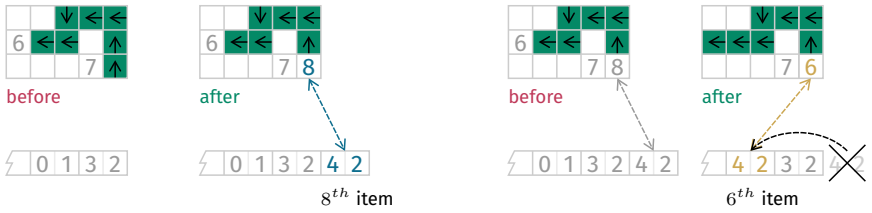


FIGURE 28.2 Left: changing the $(4, 2)$ cell to a free cell. Right: changing the free cell $(0, 1)$ to a used cell is done by replacing it with the last cell in the free cells list.

- a grid data structure storing the state of each cell, *i.e.*, whether it is used or free and, if so, its index in the list of free cells.
- a list of all the free cells, specified by their (x, y) coordinates in the grid data structure.

Finally, to avoid a third data structure for the list of snake cells, we encode this list in the grid structure. More precisely, we store for each used cell the direction of the next used cell (towards the head). Then the only other additional state which is needed are the coordinates of the head and tail cells (see Figure 28.1).

28.2.1 Operations

In order to move the snake the new head cell must be changed from a free cell to a used cell. And the old tail cell must be changed from a used cell to a free cell. With the above data structures, these operations can be done as follows:

- to change a cell (x, y) to a free cell (see Figure 28.2, left):
  1. add the (x, y) element at the end of the free cells list. Note n the new size of this list.
  2. change the state of the (x, y) cell in the grid to "free, stored at index $n - 1$" in the free cells list.
- to change a free cell (x, y) to a used cell, replace this cell in the free list with the last free cell, in order to avoid a "hole" in this list (see Figure 28.2, right):
  1. get the free cell index stored in the (x, y) cell of the grid, noted i.
  2. set the state of the (x, y) cell of the grid to "used".
  3. get the (x', y') coordinates in the last element of the free cells list.
  4. change the free cell index stored in the (x', y') cell of the grid to i.
  5. change the coordinates in the i^{th} element of the free cells list to (x', y').
  6. remove the last element of the free cells list.

To place a new apple we need to choose a random cell in the free cells list. For this it suffice to choose a number at random between 0 and n (excluded), where n is the size of this list. But the microprocessor has no instruction for this. The solution is

to use values which *look* random. For instance, 0, 3, 2, 13, 4, 7, 6, 1, 8, 11, 10, 5, 12, 15, 14, 9 seems to contain the numbers between 0 and 16 (excluded) in random order. In fact each value v_{t+1} is computed from the previous one v_t with $v_{t+1} = 5v_t + 3 \mod 16$. More generally, using $v_{t+1} = a.v_t + b \mod m$, can give values which look random *if a, b, and m are well chosen* (for instance, $a = b = 1$ is a bad choice). The Knuth & Lewis parameters, $a = 1664525$, $b = 1013904223$, and $m = 32$, are frequently used. They give all the values between 0 and 2^{32} (excluded) in "random" order, without repetitions.

28.2.2 Encoding

We store the elements of the free cells list one after the other in a buffer, using one byte for each coordinate. The x (resp. y) coordinate of the i^{th} free cell (counting from 0) is thus at offset $2i$ (resp $2i + 1$) from the beginning of this buffer.

We represent a used cell with a number from 0 to 3, encoding the direction of the next used cell (0 for left, 1 for right, 2 for up, and 3 for bottom). We represent a free cell with its index in the list of free cells, *plus 4*. Hence, a cell state less than 4 represents a used cell, while a cell state greater than or equal to 4 denotes a free cell. To support "large" grids (up to 65532 cells) we store each state on 16 bits. We store them one after the other in a buffer, from left to right and from top to bottom. Hence, if the grid width is W cells, the state of the (x, y) cell is at a offset $2(x + y.W)$ from the beginning of this buffer.

28.3 Implementation

The screen has 30 rows of 100 characters, but each character is 8×16 pixels. To get square cells we represent each cell with 2 characters, yielding 30 rows of 50 cells. We reserve the first line to display the current score, which leaves a play area of 29 rows and 50 columns:

```
const WIDTH: u32 = 50;
const HEIGHT: u32 = 29;
const LEFT: u32 = 0;
const RIGHT: u32 = 1;
const UP: u32 = 2;
const DOWN: u32 = 3;
```

The following struct implements the data structures discussed above. The first two fields are pointers to the grid of cell states and to the list of free cells. The `seed` field contains the last v_t value computed with the Knuth & Lewis random number generator:

```
struct Game {
  cell_states: &u32,
  free_cells: &u32,
  num_free_cells: u32,
```

```
  head_x: u32,
  head_y: u32,
  tail_x: u32,
  tail_y: u32,
  apple_x: u32,
  apple_y: u32,
  direction: u32,
  score: u32,
  seed: u32
}
```

The following functions are used to read and write values in the grid of cell states and in the list of free cells. They use the offsets explained in the previous section:

```
fn game_get_cell_state(self: &Game, x: u32, y: u32) -> u32 {
  return load16(self.cell_states + 2 * (x + y * WIDTH));
}
fn game_set_cell_state(self: &Game, x: u32, y: u32, state: u32) {
  store16(self.cell_states + 2 * (x + y * WIDTH), state);
}
fn game_get_free_cell_coords(self: &Game, index: u32, x: &u32, y: &u32) {
  *x = load8(self.free_cells + 2 * index);
  *y = load8(self.free_cells + 2 * index + 1);
}
fn game_set_free_cell_coords(self: &Game, index: u32, x: u32, y: u32) {
  store8(self.free_cells + 2 * index, x);
  store8(self.free_cells + 2 * index + 1, y);
}
```

The next functions use them to change a used cell into a free cell, and vice versa (with the algorithms and encodings presented in Sections 28.2.1 and 28.2.2):

```
fn game_free_cell(self: &Game, x: u32, y: u32) {
  let free_cell_index = self.num_free_cells;
  game_set_cell_state(self, x, y, free_cell_index + 4);
  game_set_free_cell_coords(self, free_cell_index, x, y);
  self.num_free_cells = free_cell_index + 1;
}

fn game_use_cell(self: &Game, x: u32, y: u32, direction: u32) {
  let free_cell_index = game_get_cell_state(self, x, y) - 4;
  game_set_cell_state(self, x, y, direction);
  let last_free_cell_index = self.num_free_cells - 1;
  let last_x = 0;
  let last_y = 0;
  game_get_free_cell_coords(self, last_free_cell_index, &last_x, &last_y);
  game_set_free_cell_coords(self, free_cell_index, last_x, last_y);
  game_set_cell_state(self, last_x, last_y, free_cell_index + 4);
  self.num_free_cells = last_free_cell_index;
}
```

We can use them to create a new Game data structure, with its cell grid and free cells list buffers, and to initialize all the cells to free cells:

```
fn game_new(heap_p: &&u32, heap_limit: &u32) -> &Game {
  let game = mem_allocate(sizeof(Game), heap_p, heap_limit) as &Game;
  let cell_states = mem_allocate(2 * WIDTH * HEIGHT, heap_p, heap_limit);
  let free_cells = mem_allocate(2 * WIDTH * HEIGHT, heap_p, heap_limit);
  if game == null || cell_states == null || free_cells == null {
    return null;
  }
  game.cell_states = cell_states;
  game.free_cells = free_cells;
  game.num_free_cells = 0;
  let y = 0;
  let x = 0;
  while y < HEIGHT {
    x = 0;
    while x < WIDTH {
      game_free_cell(game, x, y);
      x = x + 1;
    }
    y = y + 1;
  }
  return game;
}
```

The following functions implement the Knuth & Lewis random number generator, and use it to place a new apple (they assume there is at least one free cell). The generated numbers are in $[0, 2^{32}[$, but we need a value in $[0, n[$, where n is the size of the free cells list ($n \leq 50 * 29 = 1450$). For this we use the 11 most significant bits[1] of a random number, modulo n:

```
fn random(seed: &u32) -> u32 {
  *seed = *seed * 1664525 + 1013904223;
  return *seed;
}
fn modulo(x: u32, m: u32) -> u32 { return x - (x / m) * m; }

fn game_new_apple(self: &Game) {
  let index = modulo(random(&self.seed) >> 21, self.num_free_cells);
  game_get_free_cell_coords(self, index, &self.apple_x, &self.apple_y);
}
```

To draw the apple, and to move the snake, we just need to draw a red cell, a green cell (for the new head), or a black cell (to erase the old tail). We thus implement a function to draw a cell in a given color. Since we reserved the first line for the score,

[1] The least significant bits are "less random". For instance, in the 0, 3, 2, 13, 4, 7, 6, 1, 8, 11, 10, 5, 12, 15, 14, 9 sequence shown above, numbers are alternatively even and odd, *i.e.*, bit 0 is 0, 1, 0, 1, 0, 1, ...

and use 2 characters per cell, the (x, y) cell corresponds to row $y + 1$ and to columns $2x$ and $2x + 1$. To fill these characters with the given color we draw two spaces with this color as background color (the graphics card has 3 registers to set it, using the same format as for the foreground color – see Section 10.2.2 and [13]).

```
fn gpu_set_background(r: u32, g: u32, b: u32) {
  gpu_set_register(96 /*Background Color 0*/, r);
  gpu_set_register(97 /*Background Color 1*/, g);
  gpu_set_register(98 /*Background Color 2*/, b);
}
fn draw_cell(x: u32, y: u32, r: u32, g: u32, b: u32) {
  gpu_set_background(r, g, b);
  gpu_set_cursor(2 * x, y + 1);
  gpu_draw_char(' ');
  gpu_draw_char(' ');
  gpu_set_background(0, 0, 0);
}
```

The next 3 functions are used to draw the score in the top-left corner:

```
static SCORE = ['S','c','o','r','e',':',' '];
static GAME_OVER = ['G','a','m','e',' ','o','v','e','r','!'];

fn draw_integer(x: u32) {
  let quotient = x / 10;
  if quotient > 0 { draw_integer(quotient); }
  gpu_draw_char(x - 10 * quotient + '0');
}
fn draw_string(src: &u32, length: u32) {
  let i = 0;
  while i < length {
    gpu_draw_char(load8(src + i));
    i = i + 1;
  }
}
fn draw_score(score: u32) {
  gpu_set_cursor(0, 0);
  draw_string(SCORE, 7);
  draw_integer(score);
}
```

The following function places an initial snake in the middle of screen, using 6 cells and moving left, draws it, places and draws the apple, and finally initializes and draws the score. It does not initialize the `seed` on purpose, otherwise the apple would always be at the same initial position. Instead, the seed gets whatever value was here in memory (most likely the last `seed` value from a previous run of this game).

```
fn game_init(self: &Game) {
  self.head_x = WIDTH / 2;
```

```
  self.head_y = HEIGHT / 2;
  self.tail_x = self.head_x + 5;
  self.tail_y = self.head_y;
  self.direction = LEFT;
  self.score = 0;
  let x = self.head_x;
  while x <= self.tail_x {
    game_use_cell(self, x, self.head_y, LEFT);
    draw_cell(x, self.head_y, 0, 7, 0);
    x = x + 1;
  }
  game_new_apple(self);
  draw_cell(self.apple_x, self.apple_y, 7, 0, 0);
  draw_score(0);
}
```

The function below moves the snake in the current direction, and returns `OK` if this is valid, or `INVALID_STATE` otherwise. It starts by updating the head position and detecting collisions with the "walls":

```
fn game_update(self: &Game) -> u32 {
  let old_head_x = self.head_x;
  let old_head_y = self.head_y;
  if self.direction == LEFT {
    if self.head_x == 0 { return INVALID_STATE; }
    self.head_x = self.head_x - 1;
  } else if self.direction == RIGHT {
    if self.head_x == WIDTH - 1 { return INVALID_STATE; }
    self.head_x = self.head_x + 1;
  } else if self.direction == UP {
    if self.head_y == 0 { return INVALID_STATE; }
    self.head_y = self.head_y - 1;
  } else if self.direction == DOWN {
    if self.head_y == HEIGHT - 1 { return INVALID_STATE; }
    self.head_y = self.head_y + 1;
  }
```

It then checks if the head hits the body. If not, it updates the old and new head cells, and draws the new one:

```
  if game_get_cell_state(self, self.head_x, self.head_y) < 4 {
    return INVALID_STATE;
  }
  game_set_cell_state(self, old_head_x, old_head_y, self.direction);
  game_use_cell(self, self.head_x, self.head_y, 0);
  draw_cell(self.head_x, self.head_y, 0, 7, 0);
```

If the head just moved in the apple cell, it places and draws a new apple (if there is at least one free cell left), and then updates the score:

```
  if self.head_x == self.apple_x && self.head_y == self.apple_y {
    if self.num_free_cells == 0 { return INVALID_STATE; }
    game_new_apple(self);
    draw_cell(self.apple_x, self.apple_y, 7, 0, 0);
    self.score = self.score + 1;
    draw_score(self.score);
    return OK;
  }
```

Finally, if the apple was not eaten, this function ends by freeing the current tail cell, erasing it on screen, and updating the tail position to the next used cell (using the direction stored in the old tail cell):

```
  let tail_direction = game_get_cell_state(self, self.tail_x, self.tail_y);
  game_free_cell(self, self.tail_x, self.tail_y);
  draw_cell(self.tail_x, self.tail_y, 0, 0, 0);
  if tail_direction == LEFT {
    self.tail_x = self.tail_x - 1;
  } else if tail_direction == RIGHT {
    self.tail_x = self.tail_x + 1;
  } else if tail_direction == UP {
    self.tail_y = self.tail_y - 1;
  } else if tail_direction == DOWN {
    self.tail_y = self.tail_y + 1;
  }
  return OK;
}
```

We can finally implement the main function of the game. This function starts by clearing the screen, hiding the cursor (see Section 10.2.1), creating and initializing the game data structures, and drawing this initial state:

```
const ESCAPE_KEY: u32 = 27;
const ARROW_LEFT_KEY: u32 = 235;
const ARROW_RIGHT_KEY: u32 = 244;
const ARROW_UP_KEY: u32 = 245;
const ARROW_DOWN_KEY: u32 = 242;

fn main(args: &u32, args_end: &u32, heap: &u32, heap_limit: &u32) -> u32 {
  gpu_set_single_buffer();
  gpu_clear_screen();
  gpu_set_register(64 /*Memory Write Control 0*/, 128 /*Hide cursor*/);
  gpu_set_color(0, 7, 0);
  let game = game_new(&heap, heap_limit);
  if game == null { return OUT_OF_MEMORY; }
  game_init(game);
```

It continues by waiting for a short time, updating the current direction if an arrow key was pressed, updating the game state in memory and on screen, and by repeating this until the game is over:

```
  let c = 0;
  loop {
    sleep(125);
    read(KEYBOARD, &c, 1);
    if c == ESCAPE_KEY { break; }
    else if c == ARROW_LEFT_KEY { game.direction = LEFT; }
    else if c == ARROW_RIGHT_KEY { game.direction = RIGHT; }
    else if c == ARROW_UP_KEY { game.direction = UP; }
    else if c == ARROW_DOWN_KEY { game.direction = DOWN; }
    if game_update(game) != OK {
      gpu_set_cursor(44, 0);
      gpu_set_color(7, 0, 0);
      draw_string(GAME_OVER, 10);
      read(STANDARD_INPUT, &c, 1);
      break;
    }
  }
  gpu_set_register(64 /*Memory Write Control 0*/, 224 /*Show cursor*/);
  return OK;
}
```

28.4 Compilation and play

Type "`edit src/snake/snake.toy`" and enter the above code. For reference, we also provide this code in the `snake.txt` file in https://ebruneton.github.io/toypc/sources.zip. Then use the text editor again to create the following build file:

`src/snake/BUILD`:

```
toyc snake src/base.toy src/memory.toy src/gpu.toy src/snake/snake.toy
```

Finally, type "`shell src/snake/BUILD`" to compile the game and "`snake`" to play with it!

Conclusion

An operating system manages the resources and the peripherals of a computer, such as the memory, the disk, the keyboard, the screen, etc. It also provides a simple and safe way for processes to use them, via system calls. In this part we built a very basic monotasking operating system, in several steps. Starting from a basic input output system and a compiler, in the Flash1 memory bank, we progressively built and stored in the Flash0 bank an initial, self-hosting operating system. Then, using this system alone, we improved its kernel and added some utility programs.

The resulting computer is much easier to use than what it was at the beginning of this part, but can still be improved in many ways. For instance, if a process enters in an infinite loop because of a bug, we currently have no way to stop it other than rebooting the computer. To solve this we could use a combination of keys, detected in the keyboard interrupt handler, to stop the current process. Incidentally, we could also improve the keyboard driver to support combinations of keys, such as Ctrl+C or Ctrl+V. Another possible improvement is to format the Flash1 memory bank, no longer needed, into free blocks added to the Flash0 file system (to get more disk space). Alternatively, we could format the Flash1 bank with a better file system format, for instance with hierarchical directories. We could then build a new kernel version, supporting this new file system, and store it in this new disk. By successively switching between the two memory banks like this, we could then improve our kernel in arbitrary ways (in the limit of the available memory and disk space).

Further readings

Another possible improvement is to switch to a multitasking operating system, capable of running several processes concurrently. But this is a complex task, which impacts almost all parts of the system. To know how this can be done, or to know more about operating systems in general, you can read one of the following books:

- "Modern Operating Systems" [20]. This book presents the fundamental concepts used in operating systems, and discusses several strategies to implement each aspect (processes, memory management, file systems, peripherals, etc). It starts with processes, discusses all the problems related to their concurrent execution, and presents methods to solve them.
- "Operating Systems Design and Implementation" [21]. This book is from the same author and contains many sections also present in [20]. But it has a more practical point of view and presents the source code of a real operating system, small enough to be included in the book.

You can also visit the OSDev website (https://wiki.osdev.org), which provides

many more resources about operating systems. It contains for instance a list of operating systems, including educational ones such as MentOS (https://mentos-team.github.io/) and xv6 (https://pdos.csail.mit.edu/6.828/2023/xv6.html), used for teaching in universities. Finally, you can also read "Project Oberon: the design of an operating system and compiler" [23] (a second edition is available online), which describes a small self-hosting operating system and its compiler.

References

[1] Adafruit. RA8875 Touch Display Driver Board Schematic. https://learn.adafruit.com/ra8875-touch-display-driver-board/downloads, November 2023.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] Arduino. Arduino Due Full Pinout. https://content.arduino.cc/assets/A000056-full-pinout.pdf, July 2022.

[4] John Boxall. *Arduino Workshop, 2nd Edition: A Hands-on Introduction with 65 Projects*. No Starch Press, 2021.

[5] Jeroen Brinkman. MERCIA Relay Computer. https://www.relaiscomputer.nl/, 2015.

[6] K.D. Cooper and L. Torczon. *Engineering a Compiler*. Elsevier Science, 2022.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[8] Atmel Corporation. SAM3X/3A Series Datasheet. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf, March 2015.

[9] A.G. Dean. *Embedded Systems Fundamentals with Arm Cortex-M Based Microcontrollers: A Practical Approach Nucleo-F091RC Edition*. ARM Education Media, 2021.

[10] Charles Fox. *Computer Architecture*. No Starch Press, 2024.

[11] Arturo Guadalupi. Arduino Due Schematics. https://content.arduino.cc/assets/ArduinoDUE_V02g_sch.pdf, February 2021.

[12] Sarah Harris and David Harris. *Digital Design and Computer Architecture, RISC-V Edition*. Morgan Kaufmann, 1st edition edition, 2021.

[13] RAiO Technology Inc. RAiO RA8875 Character/Graphic TFT LCD Controller Specification. https://cdn-shop.adafruit.com/datasheets/RA8875_DS_V12_Eng.pdf, January 2012.

[14] American National Standards Institute. American National Standards for Information Systems – Coded Character Sets – 7-Bit American National Standard Code for Information Interchange (7-Bit ASCII). Technical Report X3.4, 1986.

[15] S. Klabnik and C. Nichols. *The Rust Programming Language, 2nd Edition*. No Starch Press, 2023.

[16] Arm Limited. Cortex-M3 Technical Reference Manual. https://documentation-service.arm.com/static/5e8e107f88295d1e18d34714, June 2008.

[17] Arm Limited. Armv7-M Architecture Reference Manual. https://documentation-service.arm.com/static/606dc36485368c4c2b1bf62f, February 2021.

[18] James Newman. The Megaprocessor. https://www.megaprocessor.com/, 2016.

[19] Shenzhen ChengHao Optoelectronic. Specification, Product Model : CH700WV03A-T. https://cdn-shop.adafruit.com/product-files/2354/Datasheet+.pdf, November 2016.

[20] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, USA, 4th edition, 2014.

[21] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., USA, 2006.

[22] C. Ünsalan, H.D. Gürhan, and M.E. Yücel. *Embedded System Design with ARM Cortex-M Microcontrollers: Applications with C, C++ and MicroPython*. Springer International Publishing, 2022.

[23] Niklaus Wirth and Jürg Gutknecht. *Project Oberon: the design of an operating system and compiler*. ACM Press/Addison-Wesley Publishing Co., USA, 1992.

[24] Joseph Yiu. *The Definitive Guide To ARM Cortex-M3 and Cortex-M4 Processors*. Newnes, 3rd edition edition, 2014.

APPENDIX

A Bill of Materials

The table below lists all the necessary components to assemble our toy computer, with their price as of July 2024. It is important to use the exact Arduino, display, driver board and keyboard models listed here. Otherwise they might not work with the programs presented in this book.

Assembling these components requires some soldering tasks. The tools necessary for this are not included here (you can avoid buying them if you have access to a makerspace).

Part	Net price
Arduino Due Cortex M3 84MHz, 512 KB Flash, 96 KB RAM, 3.3V https://store-usa.arduino.cc/products/arduino-due	$48
7″ TFT Display 800x480 pixels with Touchscreen https://www.adafruit.com/product/2354	$35
RA8875 Driver Board for 40-pin TFT Touch Displays - 800x480 max https://www.adafruit.com/product/1590	$40
Miniature Keyboard PS/2 and USB interface https://www.adafruit.com/product/857	$30
4 channel Logic Level Converter https://www.sparkfun.com/products/12009	$4
Half Sized Breadboard - 400 Tie Points https://www.adafruit.com/product/64	$5
Male/Male Jumper Wires - 20 x 3″ (75mm) https://www.adafruit.com/product/1956	$2
Female/Male Jumper Wires - 20 x 3″ (75mm) https://www.adafruit.com/product/1953	$2
Total	**$166**

TABLE A.1 The Bill of Materials of our toy computer.

APPENDIX

B ASCII codes

The table below lists the character codes defined by the "American Standard Code for Information Interchange" used in this book. The full list can be found in [14].

Code	Char
08	BackSpace
09	Tab
0A	Enter
1B	Escape
20	Space
21	!
22	"
23	#
24	$
25	%
26	&
27	’
28	(
29	)
2A	*
2B	+
2C	,
2D	-
2E	.
2F	/
30	0
31	1
32	2
33	3
34	4

Code	Char
35	5
36	6
37	7
38	8
39	9
3A	:
3B	;
3C	<
3D	=
3E	>
3F	?
40	@
41	A
42	B
43	C
44	D
45	E
46	F
47	G
48	H
49	I
4A	J
4B	K
4C	L
4D	M

Code	Char
4E	N
4F	O
50	P
51	Q
52	R
53	S
54	T
55	U
56	V
57	W
58	X
59	Y
5A	Z
5B	[
5C	\
5D	]
5E	^
5F	_
60	‘
61	a
62	b
63	c
64	d
65	e
66	f

Code	Char
67	g
68	h
69	i
6A	j
6B	k
6C	l
6D	m
6E	n
6F	o
70	p
71	q
72	r
73	s
74	t
75	u
76	v
77	w
78	x
79	y
7A	z
7B	{
7C	|
7D	}
7E	~
7F	Delete

APPENDIX

C IBM PC Set 2 scancodes

The table below lists the scancodes which are emitted by each key of the MCSaite keyboard (see Table A.1) when it is pressed or released. This is a subset of the IBM PC Set 2 scancodes (https://wiki.osdev.org/PS/2_Keyboard#Scan_Code_Set_2).

Key	Press	Release
F9	01	F0,01
F5	03	F0,03
F3	04	F0,04
F1	05	F0,05
F2	06	F0,06
F12	07	F0,07
F10	09	F0,09
F8	0A	F0,0A
F6	0B	F0,0B
F4	0C	F0,0C
Tab	0D	F0,0D
‘ ~	0E	F0,0E
Alt (left)	11	F0,11
Shift (left)	12	F0,12
Ctrl (left)	14	F0,14
q Q	15	F0,15
1 !	16	F0,16
z Z	1A	F0,1A
s S	1B	F0,1B
a A	1C	F0,1C
w W	1D	F0,1D
2 @	1E	F0,1E
c C	21	F0,21
x X	22	F0,22

Key	Press	Release
d D	23	F0,23
e E	24	F0,24
4 $	25	F0,25
3 #	26	F0,26
Space	29	F0,29
v V	2A	F0,2A
f F	2B	F0,2B
t T	2C	F0,2C
r R	2D	F0,2D
5 %	2E	F0,2E
n N	31	F0,31
b B	32	F0,32
h H	33	F0,33
g G	34	F0,34
y Y	35	F0,35
6 ^	36	F0,36
m M	3A	F0,3A
j J	3B	F0,3B
u U	3C	F0,3C
7 &	3D	F0,3D
8 *	3E	F0,3E
, <	41	F0,41
k K	42	F0,42
i I	43	F0,43

Key	Press	Release
o O	44	F0,44
0)	45	F0,45
9 (	46	F0,46
. >	49	F0,49
/ ?	4A	F0,4A
l L	4B	F0,4B
; :	4C	F0,4C
p P	4D	F0,4D
- _	4E	F0,4E
' "	52	F0,52
[{	54	F0,54
= +	55	F0,55
CapsLock	58	F0,58
Shift (right)	59	F0,59
Enter	5A	F0,5A
] }	5B	F0,5B
\ |	5D	F0,5D
BackSpace	66	F0,66
1 (keypad)	69	F0,69
4 (keypad)	6B	F0,6B
7 (keypad)	6C	F0,6C
0 (keypad)	70	F0,70
. (keypad)	71	F0,71
2 (keypad)	72	F0,72

Key	Press	Release
5 (keypad)	73	F0,73
6 (keypad)	74	F0,74
8 (keypad)	75	F0,75
Escape	76	F0,76
NumLock	77	F0,77
F11	78	F0,78
+ (keypad)	79	F0,79
3 (keypad)	7A	F0,7A
- (keypad)	7B	F0,7B
* (keypad)	7C	F0,7C
9 (keypad)	7D	F0,7D
ScrollLock	7E	F0,7E
F7	83	F0,83
Alt (right)	E0,11	E0,F0,11
End	E0,69	E0,F0,69
ArrowLeft	E0,6B	E0,F0,6B
Home	E0,6C	E0,F0,6C
Insert	E0,70	E0,F0,70
Delete	E0,71	E0,F0,71
ArrowDown	E0,72	E0,F0,72
ArrowRight	E0,74	E0,F0,74
ArrowUp	E0,75	E0,F0,75
PageDown	E0,7A	E0,F0,7A
PageUp	E0,7D	E0,F0,7D

Key	Press	Release
PrintScreen	E0,12,E0,7C	E0,F0,7C,E0,F0,12
Pause	E1,14,77,E1,F0,14,F0,77	

APPENDIX

D Compiler error codes

The table below gives the meaning of the error codes which can be output by the compiler implemented in Part 3.

Code		Meaning
1	01_{16}	Out-of-memory
10	$0A_{16}$	Unexpected end of input
11	$0B_{16}$	Printable character expected
12	$0C_{16}$	Quote character expected
20	14_{16}	Unexpected token
21	15_{16}	Integer token expected
22	16_{16}	Identifier token expected
23	17_{16}	End of input expected
24	18_{16}	Instruction opcode expected
25	19_{16}	Comparison operator expected
30	$1E_{16}$	Already defined symbol
31	$1F_{16}$	Already resolved symbol
32	20_{16}	Undefined forward reference
33	21_{16}	Undefined symbol
34	22_{16}	Function name expected
35	23_{16}	Illegal identifier expression
36	24_{16}	Illegal address-of operator argument
37	25_{16}	Unreachable statement
38	26_{16}	Illegal left hand side assignment expression
39	27_{16}	Break statement outside of a loop
40	28_{16}	Unreachable statement
41	29_{16}	Missing return
42	$2A_{16}$	Illegal raw struct type
43	$2B_{16}$	Expression type mismatch

Code		Meaning
44	$2C_{16}$	Illegal raw struct expression type
45	$2D_{16}$	Illegal void expression type
46	$2E_{16}$	Function call argument type mismatch
47	$2F_{16}$	Struct name expected
48	30_{16}	Expression with struct pointer type expected
49	31_{16}	Expression with pointer type expected
50	32_{16}	Expression with integer type expected (left hand side)
51	33_{16}	Expression with integer type expected (right hand side)
52	34_{16}	Different pointer types on left and right hand sides
53	35_{16}	Incorrectly typed add or sub expression
54	36_{16}	Comparison of expression with different types
55	37_{16}	Unexpected return value in void function
56	38_{16}	Missing return value in non-void function
57	39_{16}	Illegal use of null in let statement
58	$3A_{16}$	Function parameter type does not match forward declaration
59	$3B_{16}$	Function parameter count does not match forward declaration
100	64_{16}	Const instruction argument overflow
101	65_{16}	Stack instruction argument overflow
102	66_{16}	Heap instruction argument overflow
103	67_{16}	Branch instruction argument overflow
104	68_{16}	Branch with link instruction argument overflow
105	69_{16}	Too many registers used
106	$6A_{16}$	Static instruction argument overflow
107	$6B_{16}$	Too many function parameters
108	$6C_{16}$	Too many function arguments
109	$6D_{16}$	Return instruction argument overflow
110	$6E_{16}$	Unaligned destination buffer

APPENDIX

E Boot Assistant scripts

The following sections list the source code of the scripts used in Part 2 on the host computer.

E.1 boot_helper.py

```
import serial

# Initialize the serial port to communicate with the SAM-BA program,
# as described in section 20.4.1 of the SAM3X / SAM3A Datasheet.
try:
  serial_port = serial.Serial(port='/dev/ttyACM0', baudrate=115200,
                parity=serial.PARITY_NONE, stopbits=serial.STOPBITS_ONE,
                bytesize=serial.EIGHTBITS, timeout=5, write_timeout=1)
except:
  exit('ERROR: could not open serial port.')


def run(command, verbose=False):
  serial_port.write(bytearray(command.encode('ascii')))
  if command.endswith('#'):
    bytes = bytearray()
    while True:
      byte = serial_port.read()
      if len(byte) != 1:
        exit('ERROR: no response from device.')
      if chr(byte[0]) == '>':
        result = bytes.decode('ascii').lstrip()
        if verbose:
          print(f'{result}>', end='')
        return result
      bytes.append(byte[0])


# Flush the connection and switch the SAM-BA Monitor to ASCII mode.
serial_port.flush()
```

```
run('T#', verbose=True)

if __name__ == '__main__':
  # Main loop (read commands from stdin, run them).
  while True:
    try:
      commands = input().replace('#', '#\').split()
    except (EOFError, KeyboardInterrupt):
      exit('')
    for command in commands:
      if command.strip() == 'exit#':
        exit()
      run(command, verbose=True)
```

E.2 flash_helper.py

```
import os
import re
import stat
import boot_helper

# Check if stdin is a file. If not, we assume it is an interactive terminal.
stdin_from_file = stat.S_ISREG(os.fstat(0).st_mode)


def wait_ready(register):
  while boot_helper.run(f'w{register:08X},#').strip() != '0x00000001':
    pass


class Page:
  def __init__(self, index):
    self._index = index
    self._address = index * 256 + 0x80000
    self._values = []
    self._dirty = False
    print(f'Reading page {self._index}...', end='')
    for i in range(0, 64):
      address = self._address + 4 * i
      value = boot_helper.run(f'w{address:08X},#').strip()
      self._values.append(value[2:])
    print(' Done.')

  def set(self, index, value):
    if value != self._values[index]:
      self._values[index] = value
```

```
      self._dirty = True

  def flash(self):
    if not self._dirty:
      return
    print(f'Writing page {self._index}...', end='')
    for i in range(0, 64):
      address, value = self._address + 4 * i, self._values[i]
      boot_helper.run(f'W{address:08X},{value}#')
    if self._index < 1024:
      command = 0x5A000003 | (self._index << 8)
      boot_helper.run(f'W400E0A04,{command:08X}#')
      wait_ready(0x400E0A08)
    else:
      command = 0x5A000003 | ((self._index - 1024) << 8)
      boot_helper.run(f'W400E0C04,{command:08X}#')
      wait_ready(0x400E0C08)
    for i in range(0, 64):
      address = self._address + 4 * i
      value = boot_helper.run(f'w{address:08X},#').strip()[2:]
      if int(value, 16) != int(self._values[i], 16):
        exit(f'ERROR: page write failed at address {address:08X}')
    print(' Done.')


# Main loop (read commands from stdin, run them).
pages = {}
while True:
  try:
    commands = input().replace('#', '#\').split()
  except (EOFError, KeyboardInterrupt):
    if stdin_from_file:
      print('Done.', end='')
    exit('')
  for command in commands:
    command = command.strip()
    if command == 'exit#':
      exit()
    if command == 'flash#':
      for _, page in sorted(pages.items()):
        page.flash()
      pages = {}
      print('>', end='')
      continue
    if command == 'reset#':
      boot_helper.run('W400E0A04,5A00010B#')  # Set boot from flash.
      wait_ready(0x400E0A08)
```

```
    reset = 'W400E1A00,A500000D#'
    boot_helper.serial_port.write(bytearray(reset.encode('ascii')))
    exit()
  match = re.match(r"W([0-9A-Fa-f]{1,8}),([0-9A-Fa-f]{1,8})#", command)
  if match:
    address, value = int(match.group(1), 16), match.group(2)
    if address >= 0x80000 and address < 0x100000:
      if address % 4 != 0:
        exit(f'ERROR: invalid address {address}.')
      page, word = (address - 0x80000) // 256, (address % 256) // 4
      if page not in pages:
        pages[page] = Page(page)
      pages[page].set(word, value)
      if not stdin_from_file:
        print('>', end='')
      continue
  boot_helper.run(command, verbose=not stdin_from_file)
```

Programming a toy computer from scratch

This book introduces how computer hardware, programming languages and operating systems work, via a practical example which can be understood down to the smallest detail. For this it explains how you can assemble and program a toy computer, in an entirely bottom-up way, without using any existing programming tool. The end result is a toy, monotasking operating system with a command line shell, a text editor, a compiler, and a few utilities, in less than 3300 lines.