Chapter 3

The Proper Use of Assembly Directives

As for any language, the syntactic aspect of a listing is of crucial importance so that the compiler (in the case of higher level language structures) or the assembler (for listings written in assembly language) understands the rest of the characters that it will read and that make up the program. If we consider the previous examples, an assembler would have trouble generating the corresponding executable code: it lacks a lot of information. Only a few instructions, without any context, were transcribed in the examples that have previously been presented. Where is the code entry point, where does the program end, where is the code located in memory, what are the constants or variables?

This chapter aims to define this context through the description of assembly directives.

3.1. The concept of the directive

An assembly directive is a piece of information that appears word-for-word in the listing and which is supplied by the assembler to give the construction rules of the executable. These lines in the source file, though an integral part of the listing and necessary for its coherence, do not correspond directly to any line of code. These pieces of information will therefore not appear during the disassembly of the code and will be also be missing if a *hacker* tries to reverse engineer it. Disassembly is a process that consists of converting the code (that is, the collection of words read in memory) into the corresponding symbols and, if necessary, the operands (numeric or register). The coding of an instruction being biunivocal, there is no particular problem in re-transcribing a code extracted from memory into primitive assembly

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

language. The symbolic layer does not give an understanding of that which is programmed – it will still be quite incomprehensible! For example, the disassembly of the code in Example 2.2 gives the following lines:

EXAMPLE 3.1.– *Example of disassembly*

```
0x080001A0     BL.W 0x080001A8

0x080001A4     B 0x080001A4

0x080001A6     NOP

0x080001A8     NOP

0x080001AA     BX LR
```

In this example, the first column indicates the value of the address where the instruction is stored. By glancing at these addresses, we can see that an instruction is coded on either two bytes (Thumb-type code) or four bytes (Thumb-2 type code). The second column corresponds to the decoding of the read instruction and the third to the accompanying potential arguments.

3.1.1. *Typographic conventions and use of symbols*

In the following text, the following typographic conventions have been adopted:

– *slim italics* denote a sequence of characters that you must choose;

– words written in CAPITALS are compulsory. We write them in capitals to make them obvious, but they can also be written in lower case;

– ***bold italics*** indicate a field where the value is to be chosen from a set list;

– the areas not written in brackets { } are compulsory. The other fields are thus optional, but we never write the brackets.

In order to write your programs, you will need to define the symbols (a constant, a label to identify a line of code, a variable, etc.). We can also use the term identifier for these user-defined names. A symbol will always be associated with a numerical value by the assembler and the linker: in the same way as the directives defined above, a symbol will never be explicitly included in the final code. The readability of your programs, by nature very low in assembly language, is therefore directly linked to the semantics of your symbols. It is consequently better to be generous with characters in order to make it more explicit. From a syntactic point of view, a symbol must also obey the following rules if it is to be accepted by the assembler:

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

– the name of a symbol must be unique within a given module;

– the characters may be upper or lower case letters, numbers or "underscores". The assembler is case-sensitive;

– *a priori*, there is no maximum length;

– the first character cannot be a number;

– the key words (mnemonics, directives, etc.) of the language are reserved;

– if it proves necessary to use a palette of more important characters (in order to mix your code with a compiler, for example), it is possible to do this by surrounding the symbol with | (these do not become part of the symbol). For example |*.text*| is a valid symbol and the assembler will memorize it (in its symbol table) as *.text*.

3.2. Structure of a program

Writing a program in assembly language, in its simplest form, implies that the user can, in a source file (which is just a text file and so a simple series of ASCII characters):

– define the sequence of code instructions, so that the assembler will be able to translate them into machine language. This sequence, once assembled and given Cortex-M3 Harvard structure, will be stored in CODE memory;

– declare the data it will use, by giving it an initial or constant value, if necessary. This allows the assembler to give orders to reserve the necessary memory space, by initializing everything that is predestined to fill the DATA memory, when appropriate.

REMARK 3.1.– A third entity is required for the correct running of a program: the system stack. This is not fixed in size or memory location. This implies that somewhere in the listing there is a reservation for this specific area and at least one instruction for the initialization of the stack pointer (SP) responsible for its management. In using existing development tools, this phase is often included in a file (written in assembly language) that contains a number of initializations. Indeed, the μcontroller hosting Cortex-M3 must also undergo a number of configuration operations just after a reset; the initialization of the stack in this file is consequently not aberrant. The advanced programmer will, however, have to verify that the predefined size of the system stack is not under- or over-sized relative to its application.

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

3.2.1. *The AREA sections*

A program in assembly language must have at least two parts, which we will refer to as sections, that must be defined in the listing by the *AREA* directive:

– one section of code containing the list of instructions;

– one section of data where we find the description of the data (name, size, initial value).

REMARK 3.2.– Unlike in higher level languages where the declaration of variables can be more or less mixed with instructions, assembly language requires a clear separation.

From the point of view of the assembler, a section is a contiguous zone of memory in which all of the elements are of the same logical nature (instructions, data, and system stack).

The programmer uses the AREA directive to communicate with the assembler in order to show the beginning of a section. The section naturally terminates at the beginning of another section, so there is no specific marker for the end of a section.

The body of a section is made up of instructions for the *CODE* part or of various place reservations (whether initialized of not) for the *DATA* section.

The general definition syntax of a section, as it must be constructed in a source file, is:

```
AREA   Section_Name     { ,type } { ,attr } ...

       …  Body of the section:
       …  definitions of data
       …  or instructions, according to the case
```

So let us explain the four fields:

– AREA: the directive itself;

– *Section_Name*: the name that you have given to the section in keeping with the rules set out earlier;

– ***type***: ***code*** or ***data***: indicates the type of section being opened;

– a suite of non-compulsory options. The principal options are:

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

- ***readonly*** or ***readwrite***: indicates whether the section is accessible to read only (the default for CODE sections) or to read and write (the default for DATA sections),

- ***noinit*:** indicates, for a DATA section, that it is not initialized or initialized at 0. This type of section can only contain rough memory reservations or reservations for data initialized at 0, and

- ***align = n*** with a value between 0 and 31. This option indicates how the section should be placed in memory. The section will be aligned with a 2^n modulo address, or in other terms it signifies that the least significant *n* bits of the first address of the section would be at 0.

There are other options which, if they need to be put in place, show that you have reached a level of expertise beyond the scope of this book.

In a program, a given section can be opened and closed several times. It is also possible that we might like to open different sections of code or data that are distinct from each other. Finally, all of these various parts combine automatically. That is the role of the linker, which should therefore take into account the various constraints so that it can assign memory addresses for each constituent section of the project.

3.3. A section of code

A section of code contains instructions in symbolic form. One instruction is written on one line, according to the following syntax:

{ label } ***SYMBOL*** *{* ***expr*** *}{* ***,expr*** *}{* ***,expr*** *}* { ; comment}

The syntax of all assembly language is rigorous. At most, we can place one instruction on one line. It is also permitted to have lines without an instruction containing a label, a comment or even nothing at all (to space out the listing in order to make it more readable). Finally, please note that a label must always be placed in the first column of the line.

3.3.1. *Labels*

A label is a symbol of your choice that is always constructed according to the same rules. The label serves as a marker, or an identifier, for the instruction (or the data). It allows us to go to that instruction during execution by way of a jump (or branch).

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

Below is an example of a piece of program of limited algorithmic interest, but that contains a loop and consequently a label. With each loop, the program increments the *R3* register by a value contained in *R4*. The number of loops carried out is also calculated by an incrementation (unitary in this case) in register *R0*. The loop is carried out as long as it does not exceed the capacity (so as long as the flag *C* of register *xPSR* remains at 0).

EXAMPLE 3.2.– *Jump to a label*

```
       MOV     R0,#0          ; Initialization of the counter
       MOV     R3, #4         ; Initial value
       MOV     R4, #35       ; Incrementation
Turnal ADD     R0,#1          ; Incrementation of the loop counter
       ADDS    R3,R4          ; Incrementation of the value
       BCC     Turnal         ; Conditional branch
Inf    B       Inf
```

The symbol # denotes an immediate addressing, i.e. it precedes a constant to use as it is (in this case for loading the register). The last instruction (*BCC*) is a conditional jump. The *B* signifies a *Branch* request and the suffix *CC* means *Clear Carry*. Therefore the processor only carries out the jump if the previous operation (addition with flag allocation) does not cause an overshoot and the switching of flag *C* to 1. The rendezvous point (*Turnal*) is the operand that follows the *BCC* instruction and corresponds to the label positioned a few lines higher.

A version of a jump without conditions is present in the last line of this example. The instruction jumps to the label that marks that same instruction. This therefore corresponds to an infinite loop.

When we place a label on an empty line, it serves to mark the first instruction (or piece of datum) that follows it. For the assembler, **labels are equivalent to addresses**. The numerical value of a label is the value of the address that it represents. This label can also be the address of an instruction or a piece of data.

REMARK 3.3.– In all assembly languages, the concept of label or constant has been expanded: it is possible to attribute a value to a label thanks to the equals (EQU) directive, which we will talk about in Chapter 4. It is the equivalent of a #define in C language. The label takes on the meaning of a numerical size written in symbolic form, which is not necessarily an address.

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

Mahout, Vincent. Assembly Language Programming : ARM Cortex-M3, John Wiley & Sons, Incorporated, 2012. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=1143595.
Created from hud on 2020-09-28 03:18:56.

3.3.2. *Mnemonic*

We use *mnemonic* to mean the symbolic name of an instruction. This name is set and the series of mnemonics makes up the instruction set.

In the ARM world, since the ARMV4 version of the architecture, there have been two distinct instruction sets: the ARM set (coded in 32 bits) and the Thumb set (coded in 16 bits). The size for coding with the Thumb set being smaller, this set offers fewer possibilities than the full ARM set. The advantage that can be drawn from this compression is a more compact code. For many ARM processors, there is the option to make them work in Thumb or ARM mode, according to the needs and appropriate constraints of the project.

Since architecture version ARMV6, ARM has introduced a second subtlety by introducing a Thumb-2 version for the instruction set. In this second version of the Thumb appellation, the 16-bit set is expanded with some 32-bit instructions to combine performance and density, as shown in Figure 3.1[1].

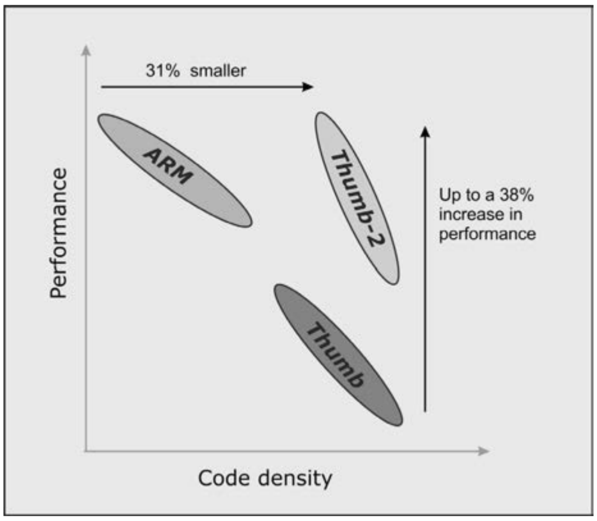

Figure 3.1. *Performances and density of the different instruction sets*

Cortex-M3 exclusively uses the Thumb-2 set. It is therefore impossible to switch to ARM mode (this is why the bit *T* in the *Execution Program Status Register* [EPSR] (see section 2.3.5) is always set to 1).

1 This diagram is extracted from ARM commercial documents, and is thus to be understood as such.

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

The Thumb-2 set understands 114 different mnemonics (excluding communication instructions with a potential coprocessor). A quick classification allows us to pick out:

– 8 mnemonics for branch instructions;

– 17 for basic arithmetic instructions (addition, comparison, etc.);

– 9 for logical shift;

– 9 for multiplication and division;

– 2 for saturation instructions;

– 4 for change of format instructions (switching from eight to 16 bits, for example);

– 10 for specific arithmetic instructions;

– 2 for *xPSR* register recovery;

– 24 for unitary read/write memory operations;

– 12 for multiple read/write memory operations;

– 17 for various instructions (*WAIT*, *NOP*, etc.).

REMARK 3.4.– Be careful not to make a mistake regarding the complexity of operations. These are all operations that only handle integers (signed or unsigned). The novice should not be surprised to not find, with these processors, capabilities for handling floating point numbers, for example. Similarly, for everything related to the algorithmic structure or management of advanced data structures, it will be necessary to break down even the smallest operation in order to adapt to assembly language.

3.3.3. *Operands*

Instructions act on and/or with the operands provided. An instruction can, depending on the case, have zero to four operands, separated by commas. Each operand is written in the form of an expression *Expr* that is assessed by the assembler. Generally, arithmetic and logical instructions take two or three operands. The case with four operands is relatively rare in this instruction set. In the case of the Thumb-2 set, apart from read/write instructions, the operands are of two types: immediate values or registers. For read/write instructions, the first operand will be a register and the second must be a memory address. Different techniques exist for specifying these addresses; such techniques correspond to the idea of *addressing mode*, which will be explained later (see section 4.3). It is therefore necessary to bear in mind that, because of the load/store architecture of Cortex-M3, this address

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

will always be stored in a register. All memory access will require prior recovery of the address of the target to be reached in a register.

Let us amend the previous program so that the initial value corresponds to the contents of a byte labeled *Bytinit*. As it is now known, let us add the declaration of sections to this program.

EXAMPLE 3.3.– *The memory operand*

```
;****************************************************************
; DATA Section
;****************************************************************
  AREA MyData, DATA, align = 2
Bytinit DCB 0x124
;****************************************************************
; CODE Section
;****************************************************************
  AREA MyCode, CODE, readonly, align = 3
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; main subroutine
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
main    PROC
        LDR R6,=Bytinit      ; Address load ①
        MOV R0,#0            ; Initialization of the counter
        LDRB R3, [R6]        ; Load of the initial value ②
        MOV R4, #35          ; Incrementation
Turnal  ADD R0,#1            ; Incrementation of the loop counter
        ADDS R3,R4           ; Incrementation of the value
        BCC Turnal           ; Conditional branch
Inf     B Inf
;*******************************************************************
```

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

In this example (see also Figure 3.2), we can see that in order to modify *R3* with the stored value of the memory block named *Bytinit*, it is necessary to pass through an additional register (*R6*). This code recovers, in a first step (①), the address of the variable (*LDR R6,=Bytinit,*). Then in a second step (②) with "indirect addressing", it loads *R3* with the stored value (*LDRB R3, [R6]*). Register *R6*, once initialized, acts like a pointer to the memory zone to be read. We can also note in passing that only the least significant byte is copied, as the *LRDB* instruction carries out the transfer of one byte. On the other hand, the instruction concerns the whole of the *R3* register, so the 24 most significant bits of this register are set to zero during the transfer. The final content of the 32-bit *R3* register is effectively equivalent to the unsigned numbers stored on a byte in *Bytinit*.

REMARK 3.5.– It is important to note that the label *Bytinit* is not in itself the data. It is merely a marker of the data – its address.

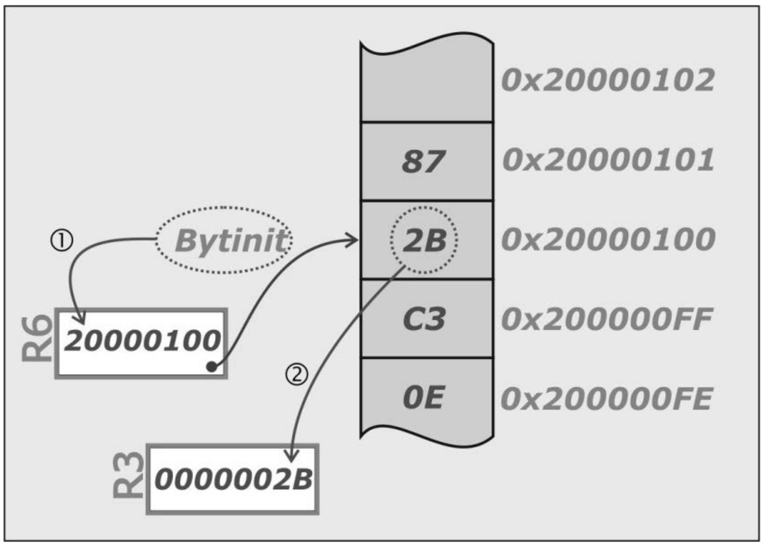

Figure 3.2. *Use of a register as a pointer*

3.3.4. *Comments*

A comment must begin with ; (a semicolon), even if it is the only thing on the line. It always finishes the code at the end of the line. If we want to write a comment requiring several lines, each line must begin with ; (a semicolon). Comments can be placed anywhere in the program, provided that they are at the end of the current line. It is worth remembering that a listing is more often read than written… even by

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

Mahout, Vincent. Assembly Language Programming : ARM Cortex-M3, John Wiley & Sons, Incorporated, 2012. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=1143595.
Created from hud on 2020-09-28 03:18:56.

those who wrote it. It is thus particularly useful when revisiting code written several weeks, months, years, etc., earlier for it to have comments, so do not hesitate to provide them!

3.3.5. *Procedure*

In the assembly language presented here, all instructions must be written inside a procedure. It is therefore normal that the first line following the opening of a section looks like a line where the mnemonic is replaced by *PROC*, as in the following example.

EXAMPLE 3.4.– *The declaration of a procedure*

```
     AREA New_Section, CODE, readonly, align = 2
MyFunct PROC
        ...     ; body of the procedure (instructions)
        ...
        ENDP
```

We call a sequence of instructions a "procedure". We can distinguish it from a subprogram (or subroutine) when the last instruction allows us to return to a calling program. We will return to the details of how to write and use procedures in section 7.1, but for now let us focus on the basics: the call to a procedure.

The most generic form for the call is a *Branch and Link* (*BL MyFunct*), with the return corresponding to a re-allocation of the instruction pointer by the instruction *BX LR* (Brach and eXchange), as shown in Example 2.2.

As with C language, there is a *principal procedure*, the first to be launched after a μcontroller initialization sequence known as the *main*. It is not standard, as it is in C language. It is perfectly possible to replace the standard initialization library with your own library, making the initialization work. This new library could then make the call to the entry point of the application program – the entry point whose name could then be freely chosen by the programmer.

REMARK 3.6.– It is possible to replace the PROC/ENDP pair with the FUNCTION/ENDFUNC pair, knowing that the language makes no distinction between procedure and function, unlike certain higher-level languages.

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

3.4. The data section

A set of directives allows us to reserve memory space that will be used by the program to store data. It is also possible assign such space an initial value, if needed. This manipulation seems simple but it needs to be looked at more closely.

3.4.1. *Simple reservation*

This just means reserving a memory space and optionally giving it a name.

```
{ label } SPACE expr
```

expr is the amount (expressed as a number of bytes) of memory that we wish to allocate. This zone will be set at 0 by default.

The numerical expressions are the quantities that are presented directly to the assembler. In the case we are considering here, it is the number of bytes being reserved but later it may be an initial value, an immediate value to be given to a register, etc. This quantity can be expressed in different bases:

– decimal base: the default base, as neither a prefix nor a suffix is required;

– hexadecimal base: as in C language, this base is selected when the quantity is prefixed with 0x (for example, the value 255 would be 0x00FF). An alternative is to use the &;

– any base (between 2 and 9): the syntax is base_digits, where base is the chosen base and digits are the characters (from 0 to *base*-1) representing the characters. (For example in base 5: 5_1213 represents the value $183 = 1 \times 5^3 + 2 \times 5^2 + 1 \times 5^1 + 3 \times 5^0$). This possibility turns out to be interesting for the binary base, in which the programmer can easily express a value (of a register, for example) for which he or she knows the bits to locate. For example, to put the three- and five-weighted bits of a byte at 1, the programmer must specify: 2_00101000 = 2_101000 = 0*x*28 = 40;

– ASCII "base": it is not useful to learn the ASCII table by heart. By surrounding a single character with single inverted commas ('), the assembler will understand that it must take the ASCII value of that character as the value of the expression. It is also possible to construct a chain of characters (to create a message, for example) by placing them within double inverted commas ("). Example 3.6 illustrates this technique with the declaration and initialization of the variable *Chain*.

Note that this expression can be a simple literal calculation. For example *LotsOfWords SPACE 12*4+3* would reserve 51 bytes. The calculation is obviously

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

done by the assembler (and so the processor of the development host computer) when generating the code (and not at any time by Cortex-M3!). The principal arithmetic operators (+, -, *, /, >> (right shift), << (left shift), & (and), | (or), etc.) allow us to construct these literal expressions. The advanced user will find an exhaustive and detailed list of existing operators and their precedence level (relative priority), knowing that brackets are allowed, in the technical documentation [ARM 10].

3.4.2. *Reservation with initialization*

Different directives exist to create memory zones containing specific values (initialized variables). Below is the list with the possible arguments:

```
{ label1 }  FILL expr {,value{,valuesize} }

{ label2 }  DCB expr1 {,expr2}{,expr2}…

{ label3 }  DCD{U} expr1 {,expr2}{,expr3}…

{ label4 }  DCW{U} expr1 {,expr2}{,expr3}…

{ label5 }  DCQ{U} expr1 {,expr2}{,expr3}…
```

The reservation for *FILL* is the counterpart of the directive *SPACE* but with initialization. We specify that we want to reserve *expr* initialized bytes with the value *value*, which is coded on *valuesize* bytes. For it to be completely consistent, the size of the reservation (*expr*) must be a multiple of the value of *valuesize*.

REMARK 3.7.– All unsigned values can be stored in a larger format than that which is strictly necessary for their size; just add 0 in front of the useful bits. The assembler cannot know *a priori* in what size it must code the information.

The other four directives specify the size of the variable with the last letter of the mnemonic: B = byte, W = half-word (2 bytes), D = word (four bytes) and Q = double word (8 bytes). The data are aligned by default: a type W datum starts on an even address (least significant bit at 0), and type D and Q data are on modulo 4 addresses (the two least significant bits at 0).

The number of reservations therefore corresponds to a number of given initial values. The first initialization is compulsory. Any following this are optional. The following example uses these different directives with different bases and several

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

forms of literal expressions. Figure 3.3 shows the memory mapping resulting from this example, taking 0x20000000 as the base address.

EXAMPLE 3.6.– *Reservations with initialization*

```
Bytinit DCB 0x24,2_0111
Chain   DCB "toto",’*’,10,13
HalfW   DCW 47, 12+3
Word    DCD 3,0xFFFF * 3
Byte    DCB 0xAB >> 2        ; two right shifts
Danger  DCWU 0x1234
```

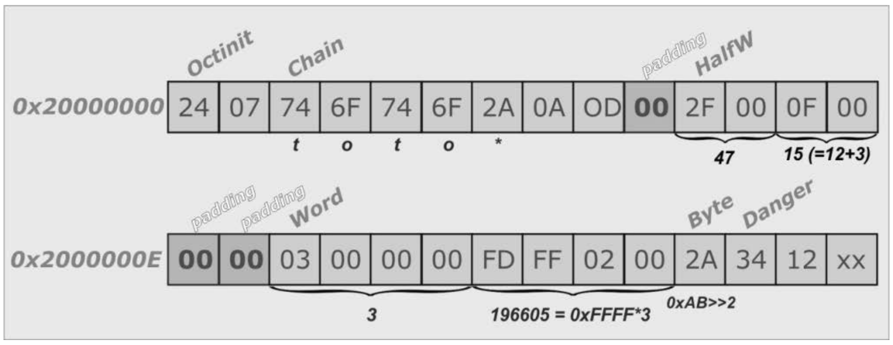

Figure 3.3. *Mapping of the example of reservations with initialization*

In the example, three padding bytes are added by the assembler to allow proper alignment of the data. A byte was placed at 0x20000009 because *HalfW* was placed at an even address (0x2000000A) and two bytes were added at 0x2000000E because the two least significant bits of the *Word* address were at 0 (0X20000010). In contrast, the half-word *Danger* finds itself unaligned at 0x20000019, since the directive *DCWU* was used to create it.

The U option means that the alignment is unnecessary, but that can be excessively dangerous! In fact, the assembler aligns its reservations on an even address for half-words and a doubly even (divisible by four) address for words. However, this alignment is not strictly necessary. Cortex-M3 allows unaligned memory accesses. Such accesses are not at all optimized, since they will need to make two reading (or writing, depending on the direction of the instruction) cycles.

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

It is primarily for the sake of optimality that alignment is in force at the level of memory reservations by the assembler. However, ARM has provided the option to prohibit unaligned access, physically speaking. This is done by setting the weight three bit (UNALIGN_TRAP) in the *Configuration Control Register* to 1. Thus positioned, any access to a half-word on an odd address, or a word at a non-doubly even address, would cause a *usage fault* (see section 8.2.1.4). If your processor is configured in this state, it becomes essential to manage it rigorously (notably, regarding the alignment of its pointers).

3.4.3. *Data initialization: the devil is in the details*

What happens in the life of an initialized variable? In a normal development cycle, we can imagine that this variable is born in the loading of the program to memory. The *loader* (program in charge of the transfer from the development toolchain to the target containing Cortex-M3) will write the planned initial value to the memory address corresponding to this variable. Then, if the variable is "variable", it will change and no longer be equal to its initial value. Let us now suppose that the user restarts the target (*RESET*). The program restarts without going through the *loader* step. What guarantees that the initial value will be re-transcribed? *A priori* in assembly language, nothing; however, it turns out that in the majority of cases (because it is often paired with a C compiler) the assembler has anticipated this. The mechanism put in place is based on a doubling of initialization zones: a first random access memory (RAM)-type address space for the variables themselves and a duplicate in read-only memory (ROM)-type (usually just after the code) containing the initial values. It is then enough to have a routine at its disposal that recovers the tables of different zone addresses and then operates a copy of one over the other. A slight adaptation of this method also allows us to reinitialize the variables at zero. These maneuvers, although very useful, may not appear very clearly. So in the case of the ARM-MDK (Microcontroller Development Kit), this is done through the use of a dynamic library in the code, which is accomplished by adding a nice but nevertheless obscure *IMPORT*||*Lib$$ Request$$armlib*|| to the main file header. This initialization mechanism is revisited in detail in Chapter 9.

3.5. Is that all?

Obviously not. There are also certain directives for the assembler that are not completely explained in this chapter. Some of them (*EQU*, *RN* and *LTORG*) are discussed in the next chapter as they deal mainly with data manipulation (and thus operand management). Others concern the sharing (*IMPORT, EXPORT*) of variables or procedures, which is the subject of Chapter 9. Yet others will not be mentioned here because interest in their use is somewhat limited (INFO, TTL, etc.).

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

Mahout, Vincent. Assembly Language Programming : ARM Cortex-M3, John Wiley & Sons, Incorporated, 2012. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=1143595.
Created from hud on 2020-09-28 03:18:56.

Those that are left are listed below. They are more specific (in the sense that their use is not generally necessary in producing an executable) but they can be very useful.

3.5.1. *Memory management directives*

```
ALIGN {expr}
```

where *expr* must be a value of any power of 2 (from 2^0 to 2^{32}). If *expr* is not specified, its value is taken to be 2. *ALIGN* is used to position data (or instructions) that follow this directive on a modulo *expr* aligned address. This directive inserts zeroes (in the DATA zone) or *NOP* instructions (in the CODE zone) in order to align the code. Its usefulness is clear for resolving potentially fatal memory unalignment problems. The use of an ALIGN 1 is possible but presents no interest. *ALIGN* can take other optional arguments to insert an additional offset alignment.

A typical example of the use of this directive is that of the reservation of memory space in order to make a user stack (also known as a *heap*). The reservation is carried out by the *SPACE* directive, which does not manage alignment. If access to this stack is only used for reading/writing on 32 bits, it is fine to position the base address of the stack on a doubly even value. Example 3.7 shows how this reservation can be done using a forced alignment.

EXAMPLE 3.7.– *Reservation for an user stack*

```
Dim             EQU 64

Variable        DCB 0           ; unused reservation just for creating
                                ; an unalignment in the memory

                Align 4         ; Forcing of alignment for

                                ; the next reservation

MyStack         SPACE 4*Dim     ; reservation of Dim words

Summit_Stack                    ; label on the top of the stack
```

```
COMMON symbol{,size{,alignment}}
```

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

This directive allows us to reserve (marked by the name *symbol*) a common memory zone of the size *size* (always in bytes) and aligned following the same syntax as that indicated for the use of *AREA* for the *alignment* option. The memory spaces thus created will be reserved by the linker in a zone set to 0 and pooled. This means that in the end the linker will only reserve a single communal zone within which the most important *COMMON* zone will be included. One of the advantages of having this directive is to optimize memory management in sizable projects, particularly for the allocation of memory dynamic in an operating system.

```
{label} DCI{.W} expr{,expr}
```

This directive allows us to reserve memory locations with initialization (*expr*), as with directives *DCB, DCD*…, but in the *CODE* zone rather than the *DATA* zone. We can, for example, use this directive to create jump tables to carry out indexed accesses to a set of procedures (see section 6.2.4).

```
REQUIRE8 {bool}
PRESERVE8 {bool}
```

Here, *bool* can have a value of *TRUE* or *FALSE*. The absence of *bool* would be equivalent to *TRUE*. These two directives allow us to specify to the assembler that we want to get or retain a modulo 8 alignment for the system stack. It is of little use in assembly language. The programmer manages his or her own stacks on the stack system, and does not have to worry about this 8-byte alignment. The usefulness of this directive becomes apparent when the code written in assembly language is interfaced with code written in C language. The compiler can, in some cases (use of floating-point numbers, for example), "require" this functional principle for the system stack. It is therefore normal to find a way to specify such a systematic alignment and thus to have the directives that it produces.

3.5.2. *Project management directives*

The *ROUT* directive allows the management and limitation of the visibility of local labels. This is useful for avoiding interference between two similar labels. A local label is a label with the form *n*{*name*}, where *n* is a number (between 1 and 99) that is normally forbidden because theoretically a label must start with a letter. Consequently, we reference the label as *%n*{*name*}. As the name is optional, the label may just be a number. The *ROUT* directive, then, is used to position the visibility border of local labels.

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

EXAMPLE 3.8.– *Use of local labels*

```
Procedure1     ROUT                ; ROUT doesn't mean routine ! !
               …                   ; Code
3Procedure1    …                   ; Code
               …                   ; Code
               BEQ %4Procedure1    ; No ambiguity
               …                   ; Code
               BGE %3              ; The target is 3Procedure1
               …                   ; Code
4Procedure1    …                   ; Code
               …                   ; Code
Others         ROUT                ; Border of visibility
               BEQ %3              ; The target is now 3Others
               …                  ; Code
3Others        …                  ; Code
               ROUT               ; Border but without the option [name]
               …                  ; Code
3              …                  ; Code   Possible too…
                                  ;        ... but not very readable
               B %3               ; Branch to the label 3 ...
```

```
GET filename
INCBIN filename
```

This allows us to insert (GET) the file *filename* into the current file. In the case of INCBIN, the file is assumed to be binary (either raw data or a type .obj executable file).

```
KEEP { symbol }
```

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

indicates to the assembler that it should keep the symbols of the referenced label in its symbol table. In effect, without this notice, the assembler does not keep track of local markers and only retains the most significant elements such as, for example, the addresses of procedures.

The following directives allow us to set conditions for the inclusion of parts of the listing in a project. What does this mean? Let us suppose that you were to develop some code that can be developed into several versions for a system. Let us assume that 90% of the code is common to all versions, and that the remaining 10% corresponds to options. How can we manage this? Either you have as many versions of the project as you have versions of the system, or you have only one project in which some parts are only included if, during the assembly phase, you indicate which version it is.

EXAMPLE 3.9.– *Use of conditioned assembly*

```
        GBLS Version              ;Declaration of a key (global string)
Version SETS ? ? ?                ; with ? ? ? "Classic" or "Design"
        AREA MyCode, CODE         ;Opening of a CODE section
main    PROC
        …                         ;Common part of the code
        BL Function1
Infinity B Infinity               ;end of the program

        AREA MyCode, CODE         ;Opening of a CODE section
;oooooooooooooooooooooooooooooooooo
IF Version = "Classic"
Function1 PROC
           …                      ; code for this first version
           ENDP
;oooooooooooooooooooooooooooooooooo
ELSEIF Version = "Design"
Function1 PROC
```

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

```
        ….                    ; code for this second version

        ENDP

ENDIF

;oooooooooooooooooooooooooooooooo
```

In Example 3.9, the function *Function1* exists in two versions. It is clear that, if nothing is added to the code, the linker will be confused to find two identical references to the same function, and would not know which to choose to finish the instruction *BL Function1*. As the inclusion of one or other of the two versions is conditioned by the structure *IF..ELSEIF..ENDIF*, however, the assembler will test to find out which part of the code should be included in the project. The other part is simply ignored. Be careful not to get confused: Cortex-M3 will never see these algorithmic structures. Likewise, it will never know of the existence of the *Version* assembly key that allowed the selection to take place. Only the assembler knows of this key, which should therefore not be confused with the idea of a variable (we could talk about environmental variables, but that term can be confusing). The programmer must, when generating the software version that he or she wants to provide, supply the key and replace the ??? with *Classic* or *Design*.

There are various types of assembly keys: Boolean (*L*), arithmetic (*A*) and string (*S*). Their creation is carried out by a *GBLx* directive (with *x* = *L*, *A* or *S*) for a global creation (which is true for all files in the project) or a *LCLx* directive for a local creation (true for only the current file). A *SETx* directive allows us to modify the value of this key. In the rest of the code, the conditioning is located inside the following structure:

```
IF Logic_Exp
…               ; code to include if Logic_Exp is true
ELSE
…               ; code to include when Logic_Exp is false
ENDIF
```

3.5.3. *Various and varied directives*

```
ASSERT Logic_Exp
```

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

The above directive allows us to display a message (such as an error message) during the second assembly phase (an assembler takes several passes to carry out its work). If *Logic_Exp* is false:

```
name CN expr
```

allows us to rename a register in a possible coprocessor:

```
{ label } DCFSU fpliteral{,fpliteral}…
```

which allows us to reserve initialized memory space for floating points. This implies that the program has access to floating-point calculation libraries and/or that a floating-point arithmetic unit type coprocessor is attached to the Cortex-M3 structure.

```
ENTRY
```

The above defines the entry point of the program. This directive is essential if you create all of the code. If you use a processor initialization library, it will integrate entry point management. In this case, you should avoid using this directive.

```
IMPORT          EXPORT
```

The two directives above allow us to share a symbol (name of a procedure, of a variable, etc.) between several files of a single project. Their advantages and uses will be elaborated on in section 9.1.5.

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

Copyright © 2012. John Wiley & Sons, Incorporated. All rights reserved.

Mahout, Vincent. Assembly Language Programming : ARM Cortex-M3, John Wiley & Sons, Incorporated, 2012. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/hud/detail.action?docID=1143595.
Created from hud on 2020-09-28 03:18:56.