Chapter 2

The Core of Cortex-M3

The previous chapter showed how the programmer could break down the different units present in a μcontroller like STM32 from a functional point of view. It is now time to delve a little deeper into the Cortex-M3 core, and to explain in detail the contents of the CM3CORE box, as shown in Figure 2.1.

It would be pointless to create a replica (which would be incomplete due to the simplification necessary) of the contents of the various reference manuals [ARM 06a, ARM 06b, ARM 06c] that give detailed explanations of Cortex functions. It would also be pointless, however, to claim to program in assembly language without having a reasonably precise idea of its structure. This chapter therefore attempts to present the aspects of the architecture necessary for reasoned programming of a μcontroller.

2.1. Modes, privileges and states

Cortex-M3 can be put into two different operating modes: *thread mode* and *Handler mode*. These modes combine with the privilege levels that can be granted to the execution of a program regarding access to certain registers:

– At the *unprivileged* access level, the executed code cannot access all instructions (those instructions specific to the access of special registers are excluded). It does not generally have access to all functions within the system (Nested Vector Interrupt Controller [NVIC], timer system, etc.). The concern is simply to avoid having code that, by bad management of a pointer for example, would be detrimental to the global behavior of the processor and severely disturb the running of the application.

– At the other end of the spectrum, at the *privileged* level, all of these limitations are lifted.

– *Thread* mode corresponds to the default mode in the sense that it is the mode that the processor takes after being reset. It is the normal mode for the execution of applications. In this mode, both privilege levels are possible.

– The processor goes into *Handler* mode following an exception. When it has finished processing the instructions for this exception, the last instruction allows a return to normal execution and causes the return to *thread* mode. In this mode, the code always has *privileged* level access.

Passage between these three types of functioning can be described by a state machine, see Figure 2.1 [YIU 07]. After being reset, the processor is in *thread* mode with privilege. By setting the least significant bit (LSB) of the *CONTROL* register, it is possible to switch into unprivileged mode (also called *user mode* in the ARM documentation) using software. In unprivileged mode, the user cannot access the *CONTROL* register, and so it is impossible to return to the privileged mode. Just after the launch of an exception (see Chapter 8) the processor switches to *Handler* mode, which necessarily has privilege. Once the exception has been processed, the processor returns to its previous mode. If, during processing of the exception, the LSB of the *CONTROL* register is modified, then the processor can return to the opposite mode to that which was in effect before the launch. The only way to switch to unprivileged *thread* mode, as opposed to privileged *thread* mode, is by going through an exception that expresses itself by passing into *Handler* mode.

This level of protection can appear somewhat minimalist. It is a little like the locking/unlocking of your mobile phone: it takes a combination of keys (known to all) to achieve. It is obviously no use in preventing theft, but it is still useful when the phone is in the bottom of a pocket.

This type of security can only be developed within a global software architecture including an operating system. In a rather simplistic but ultimately quite realistic manner, it is possible to imagine that the operating system (OS) has full access privileges. It can therefore launch tasks in unprivileged *thread* mode, which could guarantee an initial level of security. A second privilege level concerns the functions of the memory protection unit block mentioned previously. Each task can only access the memory regions assigned to it by the OS.

A supplementary element should be taken into account in order to understand the functioning of Cortex-M3. It concerns the internal state of the processor, which can be in either *Thumb* or *debug* state.

The term *Thumb* refers to the set of processor instructions (see Chapter 5) where the associated state (Thumb state) corresponds to normal running. The *debug* state results from a switch to development mode. The execution rate of a program does not follow the same rules (stopping point, observation point, etc.) in this mode, so it is understandable that it results in a particular state. As with any event in the internal mechanisms of a processor, the switch from one state to another is reflected (and/or caused) by switching the values of one or more bits. In this case, it involves two bits (*C_DEBUGEN* and *C_HALT*) located in the *Debug Halting Control and Status Register* (DHCSR).

REMARK 2.1.– Mastery of the different functioning options is not a prerequisite for writing your first programs in assembly language. The preceding brief explanations are only there to help you realize the capacities of the processor. They are also to help you to understand that the observation of the "step-by-step" execution of your program comes from the exploitation of specific processor resources by the development software.

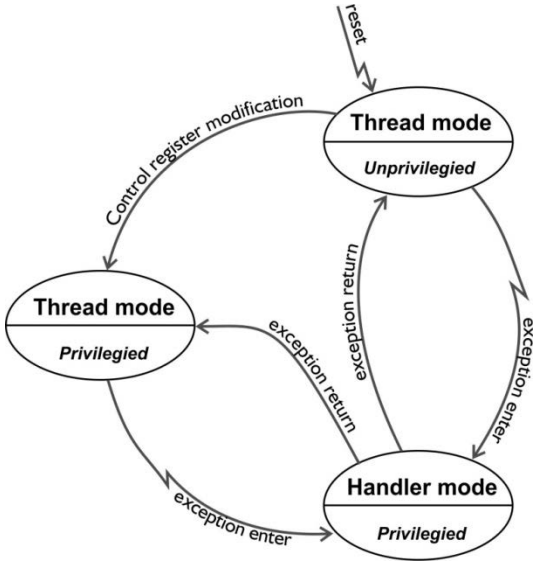

Figure 2.1. *Modes and privileges*

2.2. Registers

The first thing that should be noticed about a processor is that it is made up of various registers. This is without doubt the most pragmatic approach to the extent that modern architectures, such as those of Cortex-M3, are called *load-store* type

architectures. This means that the programs initially transfer data from memory to the registers and performs operations on these registers in a second step. Finally, and when necessary, there is the transfer of the result to memory.

First we must define what a register is. A register, in the primary sense, corresponds to the location in internal memory (in the sense of a series of accessible bits in parallel) of a processor. We should, however, adjust this definition for the simple reason that, in the case of a μcontroller, although there is internal memory (20 KB of *static RAM* in the case of a standard STM32, and that is without taking into account *Flash* memory), this internal memory does not make up a set of registers. An equally inexact definition would be to think of a register as a memory location that does not take part in memory mapping. In effect, all peripherals of a controller can be programmed by means of the values that they are given by the registers; they have no fewer physical addresses that the processor can access than any other memory space. The extensive use of the term "register" in computer architecture means that this term often encompasses, in a rather vague manner, all memory spaces whose content directly affects the functioning of the processor. In this section, we will only consider the registers of the Cortex-M3 core. Their main feature is that they are accessible by the instruction set without submitting a request to the data bus. The *opcode* (which is linked to the coding of the instruction to be executed) of an instruction manipulating the contents of a register must therefore contain the information of the register to be contacted. The execution of an instruction can cause the modification of one or more of these registers without the need to activate the data bus.

Conrtex-M3 has 17 initial registers, all obviously 32-bit:

– *R0* to *R12*: 13 general usage registers;

– *R13*: a stack pointer register, also called *SP*, *PSP (SP_process)* or *MSP (SP_main)*;

– *R14*: link register (LR);

– *R15*: ordinal counter or *PC (program pointer*);

– *xPSR:* a state register (*Program Status Register*, the *x* can be A for Application, I for Interrupt or E for Execution) that is strangely never called R16.

To these 17 registers we must add three special registers (*PRIMASK, FAULTMASK* and *BASEPRI*), which are used for exception handling (see Chapter 8). The 21st that we can add to this list is the *CONTROL* register, which has already been mentioned for its role in privilege levels but which is also a level of the R13 stack pointer.

2.2.1. *Registers R0 to R12*

These 13 registers are used, as we shall see in the passage reviewing the instruction set, as a container for storing the *operands* of an instruction and to receive the results of these operations. ARM distinguishes between the first eight registers R0 to R7 *(low registers)* and the following four (R8 to R12, the *high registers)*. The high registers have employment restrictions with respect to certain instructions. The prudent programmer will primarily use the first eight registers so that they do not have to manage these restrictions.

REMARK 2.2.– These general registers, contrary to other architectures, are only accessible in 32-bit packets. The registers can therefore not be split into two half-words or four bytes. If your application deals with bytes (chains of characters to be more precise), for example, the 24 highest weighted bits would be positioned at 0 in order for their operation to be sensible.

2.2.2. *The R13 register, also known as SP*

R13 is the SP register. As its name suggests, it points (i.e. it contains the address of a memory location) to a place that corresponds to the current location of the *system stack*. This idea of a stack will be explained in more detail later (see section 6.5.6), but for now we will just consider it as a buffer zone where the running program can temporarily store data. In the case of Cortex-M3, this storage zone is doubled, and so the SP register comes in two versions: *PSP (SP_Process)* or *MSP (SP_Main)*. It is important to note that, at any given moment, only one of the two stacks is visible to the processor. So when writing to the stack as in the following example:

EXAMPLE 2.1.– *Saving a register*

```
PUSH R14 ; PC save
         ; MSP or PSP???
```

The writing is done to the visible zone. In terms of this visibility, we can accept that, in *Handler* mode, the current pointer is always MSP. In *thread* mode, even if it can be modified with software, the current pointer is PSP. Thus, without particular manipulation, access to the system stack will be via PSP during the normal course of the program, but when there is an exception the management of the stack is subject to MSP. This division induces a much greater reliability in the functioning of the μcontroller and greater speed during context changes (not forgetting that the

peripherals communicate with Cortex-M3 through interruptions and so an exception is not exceptional, etc.).

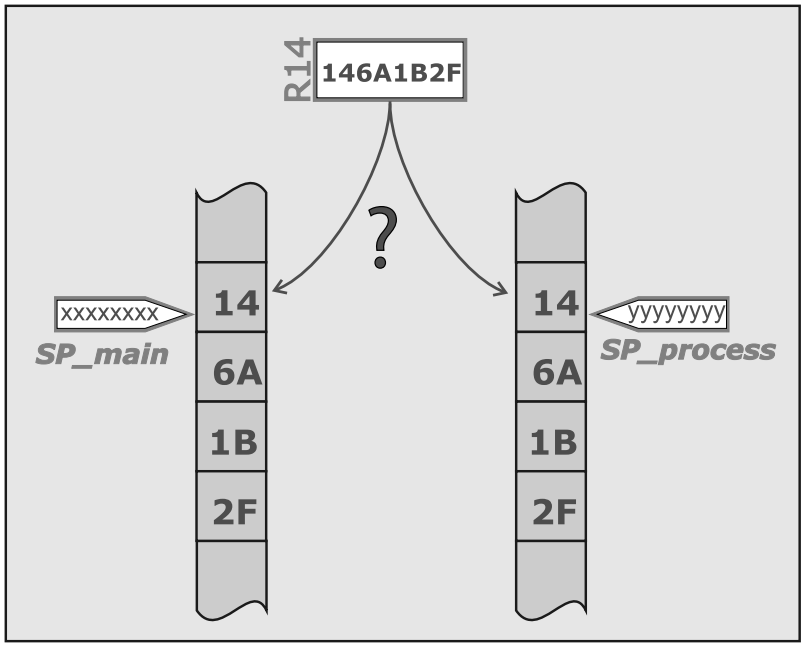

Figure 2.2. *Double stack system*

2.2.3. *The R14 register, also known as LR*

Throughout this register, there are hints of the structure of a program, which will be discussed in Chapter 7. Let us discover, through Example 2.2, one of the basic mechanisms: jumping to a subroutine.

In this example, the programmer has written two procedures (that is the term used for the concept of a subroutine): *Main* and *MyFirst*. The *Main* procedure does nothing but call *MyFirst*, which itself does nothing (the *NOP* (No Operation) instruction has no effect beyond using one machine cycle to run itself). When the processor reaches the *Branch with Link* (BL) instruction, it uses the *LR* to store the next address, where it then modifies the instruction pointer with the routine address. In a symmetrical manner, in the *MyFirst* routine with the *Branch and Exchange* (BX) instruction it will change the contents of the instruction pointer with the value contained in the *LR* and so return to where it started to continue processing the

calling program. It should be noted that, despite the implication, the symbol is not exchanged but simply copied.

EXAMPLE 2.2.– *Utility of the link register LR*

```
 Main    PROC
 Here    BL MyFirst     ; Jump to MyFirst
         ...
         ENDP
         ....
MyFirst  PROC
         NOP            ; No operation..
         BX LR          ; Return to calling program
         ENDP
```

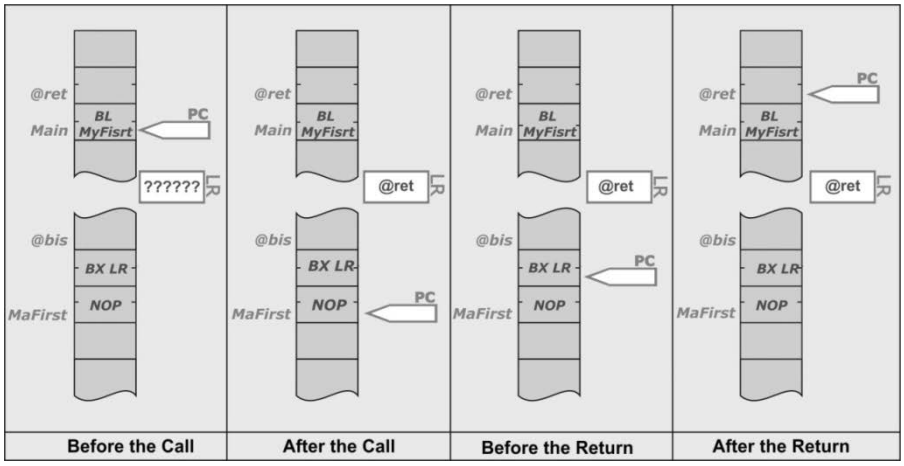

Figure 2.3. *Call to subroutine (use of the LR)*

2.2.4. *The R15 or PC register*

This register can be called "the instruction pointer", "the ordinal counter" or "the *Program Counter* (PC)" but it has a unique role: it contains the memory address where the next instruction to be executed is stored. The LSB of this register is (normally) at 0, assuming that the instructions are coded on 16 bits *(Thumb)* or 32 bits *(Thumb2)*, so they take up at least two consecutive addresses. During the

running of a code sequence, the PC pointer will automatically increment itself (usually two-by-two) in order to then point and recover the rest of the code. The term ordinal counter could imply that its functioning is limited to this simple incrementation. This is not so, and things quickly become complicated, particularly when there is a break in the sequence (in the instance of Example 2.2, where the call to a subroutine caused a discontinuity in the rest of the stored code addresses). In this case, PC would take a value that was not a simple incrementation of its initial value.

The management of this pointer is intrinsically linked to the *pipeline* structure of this *Reduced Instruction Set Computer* (RISC) architecture. This technique allows the introduction of a form of parallelism in the work of the processor. At each moment, several instructions are processed simultaneously, each at a different level. Three successive steps are necessary for an instruction to be completely carried out:

– The *Fetch* phase: recovery of the instruction from memory. This is the step where PC plays its part.

– The *Decode* phase: the instruction is a value encoded in memory (its *opcode*), which requires decoding to prepare it for execution (data recovery, for example).

– The *Execute* phase: execution of the decoded instruction and writing back of the results if necessary.

In these three stages (which are classic in a *pipeline* structure) a first upstream unit must be added: the *PreFetch* unit. This unit (which is part of ARM's expertise) essentially exists to predict what will happen during sequence breaks and to prepare to recover the next instruction by "buffering" about six instructions in advance. This practice optimizes processor speed.

This simplified view of *pipeline* stages deliberately masks its complexity. Again, in line with the objectives of this book, this level of understanding is sufficient to write programs in assembly language. Interested readers can search the ARM documentation to find all of the explanations necessary to go further.

2.2.5. *The xPSR register*

This register contains information regarding the "status" or "state" of the processor. It contains important information – a kind of short report – about what has happened in the processor. It is available in three versions, although it is only one register. The distribution of significant bits, as shown in Figure 2.4, is such that there is no intersection, so it is possible to separate it into three independent subsets:

– *APSR*, with the *A* meaning *Application*: this register contains the flags of the processor. These five bits are essential for the use of conditional operations, since the conditions exclusively express themselves as a logical combination of these flags. Updating of these is carried out by most of the instructions, *provided that the suffix S is specified in the symbolic name of the instruction*. These flags are:

- indicator *C (Carry)*: represents the "carry" during the calculation of the natural (unsigned) quantities. If *C*=1, then there was an overflow in the unsigned representation during the previous instruction, which shows that the unsigned result is partially false. Knowledge of this bit allows for much more precise work;

- indicator *Z (Zero)*: has a value of 1 if the result is zero;

- indicator *N (Negative)*: copies the most significant bit of the result. If the value is signed, *N* being 1 therefore indicates a negative result;

- indicator *V (oVerflow)*: if this has a value of 1, there has been an overflow (or underflow) of signed representation. The signed result is false;

- indicator *Q (Sticky Saturation Flag)*: only makes sense for the two specific saturation instructions *USAT* and *SSAT*: the bit is set at 1 if these instructions have saturated the register used.

– *IPSR*, with the *I* meaning *Interrupt*: in this configuration, this refers to the nine LSBs containing information. These nine bits make up the exception number (*Interrupt Service Routine* or ISR) that will be launched. For example, when the ISR has a value of 2, it corresponds to the launch of a *Non Maskable Interrupt* (NMI) interruption; if it has a value of 5 then there has been a problem with memory access.

– *EPSR*, with the *E* meaning *Execution*: this register stores three distinct pieces of information:

- the 24-bit (T) to indicate whether it is in *Thumb* state – that is, if it is using the *Thumb* instruction set. As this is always the case, this bit is always at 1. We could question the usefulness of this information. As a matter of fact, it is useless in the case of Cortex-M3, but in other architectures this bit can be at 0 to show that the processor is using the *ARM* set and not *Thumb*;

- it uses bit fields [10–15] and [25–26] to store two pieces of overlapping information (the two uses are mutually exclusive): *ICI* or *IT*;

- for *ICI*, this is information that is stored when a read/write multiple (the processor reads/writes several general registers successively, but uses only one instruction) is interrupted. Upon returning from the interruption, the processor can resume its multiple accesses from where it was before;

- for *IT*, there is a particular *If/Then*[1] instruction in the instruction set. The *IT* bits in the execution of this instruction will contain the number (between 1 and 3) of instructions that will be included in an *If/Then* block and a coding of conditions for their execution.

The *EPSR* therefore has relative use for the programmer. Moreover, when we know that it cannot be changed, we can allow ourselves to forget its existence in developing our code.

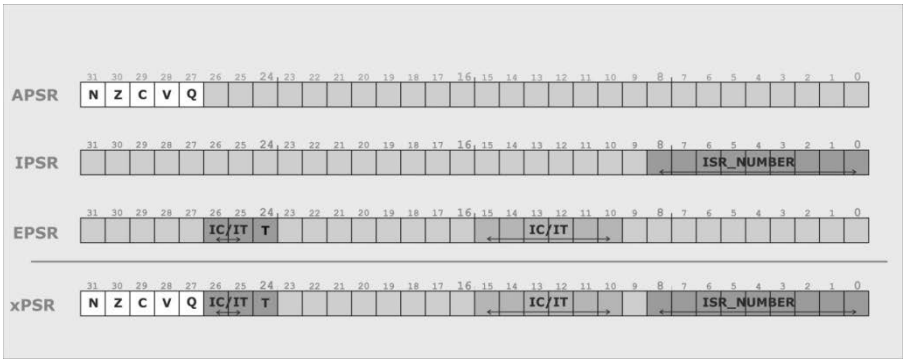

Figure 2.4. *The xPSR register*

From the point of view of assembly language, the following example shows that it is possible to access the three entities separately or to access the whole. It should be noted that this reading can only be done via a specific *MRS* (Move to Register from Special) instruction. Symmetrically, writing of these registers requires the use of a specific *MSR* (Move to Special Register) instruction.

EXAMPLE 2.3.– *Load of the status register*

```
Recup  MRS R4, APSR   ; load APSR part in R4
       MRS R5, IPSR   ; load IPSR part in R5
       MRS R6, EPSR   ; load EPSR part in R6
       MRS R7, PSR    ; load the whole PSR register in R5
```

1 Fortunately, this particular instruction is not the only way to write "If… If not… Then" structure in assembly language. A more systematic approach, by conditional jump, allows the construction of all kinds of algorithmic structures (see section 6.2).