Chapter 1

Overview of Cortex-M3 Architecture

A computer program is usually defined as a sequence of instructions that act on data and return an expected result. In a high-level language, the sequence and data are described in a symbolic, abstract form. It is necessary to use a compiler to translate them into machine language instructions, which are only understood by the processor. Assembly language is directly derived from machine language, so when programming in assembly language the programmer is forced to see things from the point of view of the processor.

1.1. Assembly language versus the assembler

When executing a program, a computer processor obeys a series of numerical orders – instructions – that are read from memory: these instructions are encoded in binary form. The collection of instructions in memory makes up the *code* of the program being executed. Other areas of memory are also used by the processor during the execution of code: an area containing the *data* (variables, constants) and an area containing the *system stack*, which is used by the processor to store, for example, local data when calling subprograms. Code, data and the system stack are the three fundamental elements of all programs during their execution.

It is possible to program directly in machine language – that is, to write the bit instruction sequences in machine language. In practice, however, this is not realistic, even when using a more condensed script thanks to hexadecimal notation (numeration in base 16) for the instructions. It is therefore preferable to use an *assembly language*. This allows code to be represented by symbolic names, adapted to human understanding, which correspond to instructions in machine language.

Assembly language also allows the programmer to reserve the space needed for the system stack and data areas by giving them an initial value, if necessary. Take this example of an instruction to copy in the no. 1 general register of a processor with the value 170 (AA in hexadecimal). Here it is, written using the syntax of assembly language studied here:

EXAMPLE 1.1.– *A single line of code*

```
MOV R1, #0xAA  ; copy (move) value 170 (AA in hexa)
               ; in register R1
```

The same instruction, represented in machine language (hexadecimal base), is written: E3A010AA. The symbolic name *MOV* takes the name *mnemonic. R1* and *#0xAA* are the *arguments* of the instruction. The semicolon indicates the start of a commentary that ends with the current line.

The *assembler* is a program responsible for translating the program from the assembly language in which it is written into machine language. Upon input, it receives a source file that is written in assembly language, and creates two files: the object file containing machine language (and the necessary information for the fabrication of an executable program), and the printout assembly file containing a report that details the work carried out by the assembler.

This book deals with assembly language in general, but focuses on processors based on Cortex-M3, as set out by Advanced RISC Machines (abbreviated to ARM). Different designers (Freescale, STmicroelectronics, NXP, etc.) then integrate this structure into μcontrollers containing memory and multiple peripherals as well as this processor core. Part of the documentation regarding this processor core is available in PDF format at www.arm.com.

1.2. The world of ARM

ARM®

ARM does not directly produce semiconductors, but rather provides licenses for microprocessor cores with 32-bit RISC architecture.

This Cambridge-based company essentially aims to provide semiconductors for the embedded systems market. To give an idea of the position of this designer on this market, 95% of mobile telephones in 2008 were made with ARM-based processors. It should also be noted that the A4 and A5 processors, produced by Apple and used in their iPad graphics tablets, are based on ARM Cortex-Type A processors.

Since 1985 and its first architecture (named ARM1), ARM architectures have certainly changed. The architecture upon which Cortex-M3 is based is called ARMV7-M.

ARM's collection is structured around four main families of products, for which many licenses have been filed[1]:

– the ARM 7 family (173 licenses);

– the ARM 9 family (269 licenses);

– the ARM 10 family (76 licenses);

– the Cortex-A family (33 licenses);

– the Cortex-M family (51 licenses, of which 23 are for the M3 version);

– the Cortex-R family (17 licenses).

1.2.1. *Cortex-M3*

Cortex-M3 targets, in particular, embedded systems requiring significant resources (32-bit), but for these the costs (production, development and consumption) must be reduced. The first overall illustration (see Figure 1.1) of Cortex-M3, as found in the technical documentation for this product, is a functional diagram. Although simple in its representation, every block could perplex a novice. Without knowing all of the details and all of the subtleties, it is useful to have an idea of the main functions performed by different blocks of the architecture.

1.2.1.1. *Executive units*

These units make up the main part of the processor – the part that is ultimately necessary to run applications and to perform them or their software functions:

– CM3CORE: This is the core itself. This unit contains different registers, all of the read/write instruction mechanisms and data in the form of the arithmetical and

1 Numbers from the third quarter of 2010.

logical unit for the proper execution of different instructions. The functioning of this block will be explained in detail in Chapter 2. It is necessary to understand its mechanism in order to write programs in assembly language.

– Nested Vector Interrupt Controller (NVIC): Cortex-M3 is intended to be embedded in a μcontroller, which includes peripheral units to allow interfacing with the outside world. These units can be seen as independent micromachines. The exchanges between them and Cortex-M3 must consequently be rhythmic and organized so that the sequence of tasks complies with rules (the concept of priorities) and determinism set in advance by the programmer. NVIC plays the role of "director". It is in charge of receiving, sorting and distributing the different interrupt requests generated by the collection of μcontroller units. It also manages events that threaten the smooth running of the code being executed (reset, memory bus problem, division by 0, etc.).

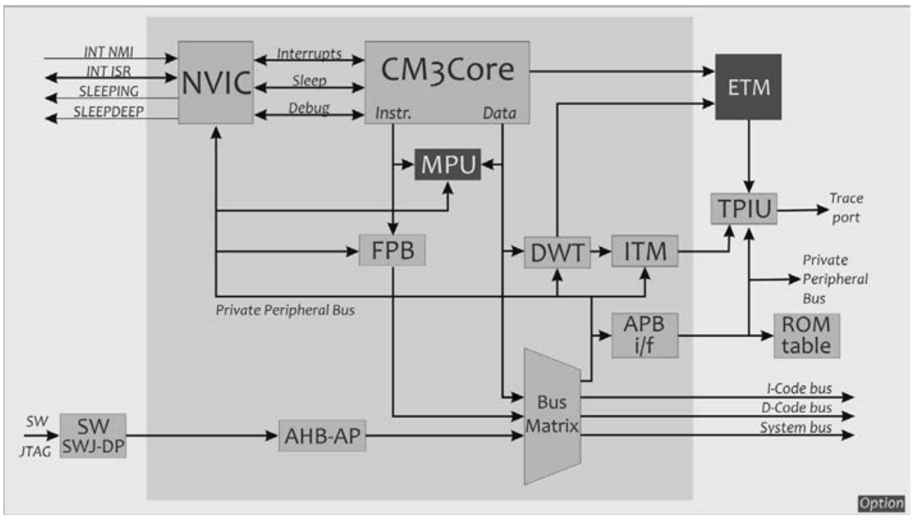


Figure 1.1. *Cortex-M3 functional diagram*

– Memory Protection Unit (MPU): This block is optional – a designer using Cortex-M3 to make their μcontroller can choose not to implement this function. This block allows the allocation of specific read and/or write privileges to specific memory zones. In this way, when different independent software tasks are executed in parallel (or more precisely in sharing the common resources of the processor), it is possible to allocate a memory zone to each task that is inaccessible to the other tasks. This mechanism therefore allows programmer to secure memory access. It usually goes hand-in-hand with the use of an operating system (real-time or otherwise) for the software layer.

– Bus matrix: This unit is a kind of gigantic intelligent multiplex. It allows connections to the external buses:

- the *ICode* bus (32-bit AHB-Lite type[2]) that carries the memory mappings allocated to the code and instructions;

- the *DCode* bus (also 32-bit AHB-Lite type) that is responsible for reading/writing in *data* memory zones;

- the *System* bus (again 32-bit AHB-Lite type), which deals with all system space access;

- the *Private Peripheral Bus* (PPB): all peripherals contained in the μcontroller are added to the Cortex-M3 architecture by the designer. ARM designed a specific bus to allow exchanges with peripherals. This bus contains 32 bits, but in this case it is the *Advanced Peripheral Bus* (APB) type. This corresponds to another bus protocol (which you may know is less efficient than AHB type, but it is more than sufficient for access to peripheral units). It should be noted that the bus matrix plays an important role in transmitting useful information to development units, which are mentioned in the next section.

1.2.1.2. *Development units*

The development of programs is an important and particularly time-consuming step in the development cycle of an embedded application. What is more, if the project has certification imperatives, it is necessary that tools (software and/or material) allowing maximum monitoring of the events occurring in each clock cycle are at its disposition. In Cortex-M3, the different units briefly introduced below correspond to these monitoring functions. They are directly implanted in the silicon of the circuit, which allows them to use these development tools at a material level. An external software layer is necessary, however, to recover and process the information issued by these units. The generic idea behind the introduction of hardware solutions is to offer the programmer the ability to test and improve the reliability of (or certify) his or her code without making any changes. It is convenient (and usual) to insert some *print ("Hello I was here")* into a software structure to check that the execution passes through this structure. This done, a code modification is introduced, which can modify the global behavior of the program. This is particularly true when time management is critical for the system, which, for

2 Advanced High-performance Bus (AHB) is a microcontroller bus protocol brought in by ARM.

embedded systems controlled by a *μ*controller, is almost always the case. The units relating to monitoring functions in Cortex-M3 include:

– Flash Patch and Breakpoint (FPB): the FPB is the most basic function for this process. It is linked to the concept of a stopping point (breakpoint), which imposes a stop on a line of code (that is to say, an instruction in the case of assembly language) located beforehand. This unit is used to mark instructions so that when they come into effect, the processor puts itself into a particular mode of operation: *debug* mode. Development software (and other pieces of software that use it) can therefore observe the state of the processor and directly influence the running of the program in progress.

– Data Watchpoint and Trace (DWT): the concept of a "point of observation" is the counterpart to the concept of a stopping point for the data. The DWT stops the program running when it works on marked data rather than a marked instruction. This action can be in reading, writing, passing on values, etc. The unit can also send requests to the *ITM* and *ETM* units.

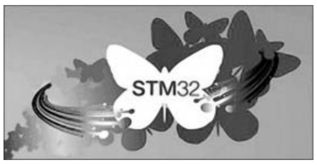
– Embedded Trace Macrocell (ETM): the concept of trace includes the capacity for the hardware to record a sequence of events during program execution. The recovery of these recordings allows the programmer to analyze the running of the program, whether good or bad. In the case of ETM, only information on instructions is stored in a first in, first out (FIFO)-type structure. As with the MPU unit, this unit is optional.

– Instrumentation Trace Macrocell (ITM): this unit also allows the collection of *trace* information on applications (software, hardware or time). The information is more limited than with the ETM unit but is nevertheless very useful in isolating a vicious bug. This is especially true if the optional ETM is not present.

– Advanced High-performance Bus Access Port (AHB-AP): this is an (Input/Output) port within Cortex-M3 that is designed to debug. It allows access to all records and all addressable memory space. It has priority in the arbitration policies of the *bus matrix*. This unit is connected upstream by the Serial Wire JTAG (Joint Test Action Group) port (SW/JTAG), which is the interface (with its physical layers) that connects to the outside world, equipped with its part of the JTAG probe. The JTAG protocol is a standardized protocol used by almost all semiconductor manufacturers.

– Trace Port Interface Unit (TPUI): the TPUI plays the same role in *trace* functions as the SW/JTAG plays in debug functions. Its existence is principally linked to the fact that it is necessary to sort the external world recordings collected by the ITM. There is an additional challenge, however, when an ETM unit is present, as it must also manage the data stored there. Its secondary role is therefore to combine and format this double stream of data before transmitting it to the port. In the outside world, it is necessary to use a *Trace Port Analyzer* to recover the data.

1.2.2. *The Cortex-M3 core in STM32*



As already stated, ARM does not directly make semiconductors. The μcontroller core designs are sold under license to designers, who add all of the peripheral units that make up the "interface with the exterior". For example, the STM32 family of μcontrollers, made by STMicroelectronics, contains the best selling μcontrollers using Cortex-M3. Like any good family of μcontrollers, the STM32 family is available in many versions. In early 2010, the STMicroelectronics catalog offered the products shown in Figure 1.2.

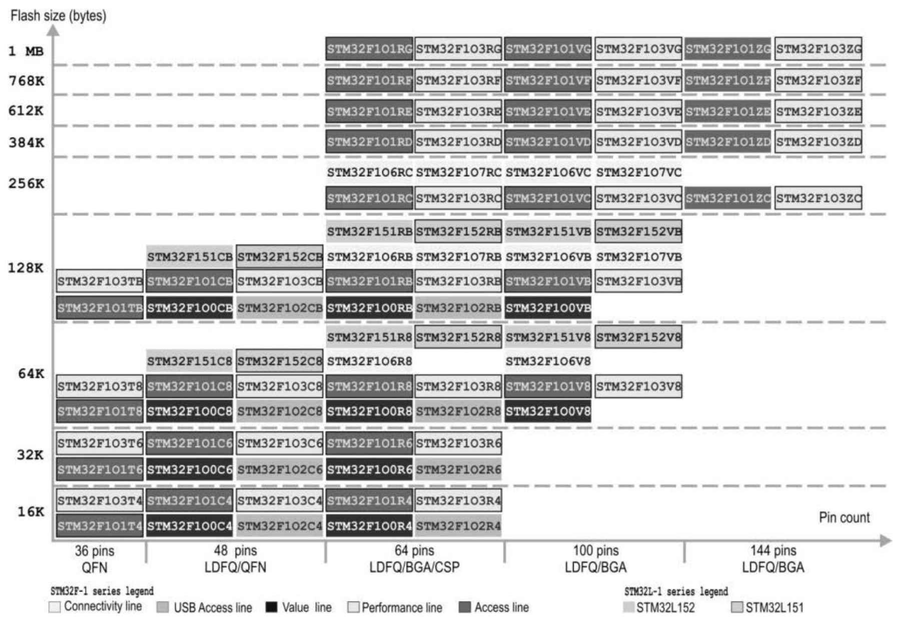


Figure 1.2. *STM32 family products*

1.2.2.1. *Functionality*

The choice of the right version of μcontroller can be a significant step in the design phase of a project: based on the needs (in terms of function, number of Input/Output, etc.) but also on the proper constraints (cost, consumption, size, etc.), each version of the processor will be more or less well adapted to the project. Again, the purpose of this book is not to go into detail on the function of the *peripheral* aspects of the μcontroller. It is, however, useful to have an overall view of the circuit that will eventually be programmed, so you are not surprised by such basic things as, for example, addressing memory. From a functional point of view, Figure 1.3 shows how STMicroelectronics has "dressed" Cortex-M3 to make a processor (the STM32F103RB version is shown here). The first important remark is that, inevitably, not all functions offered by this μcontroller are available simultaneously. Usually several units share the output pins. So, depending on the configuration that the programmer imposes, certain functions must *de facto* be unavailable for the application. Only a profound knowledge of a given processor will allow the programmer to know, *a priori*, whether the chosen version will be sufficient for the needs of the application being developed. The issue of choosing the best version is therefore far from trivial.

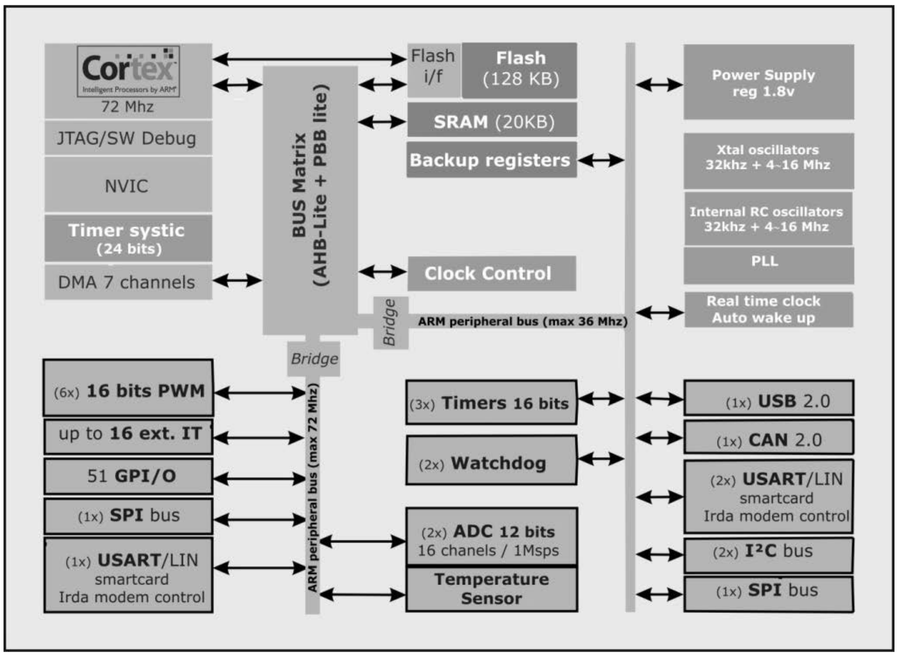

Cortex
Intelligent Processors by ARM
72 Mhz
JTAG/SW Debug
NVIC
Timer systic (24 bits)
DMA 7 channels
BUS Matrix (AHB-Lite + PBB lite)
Flash i/f
Flash (128 KB)
SRAM (20KB)
Backup registers
Clock Control
Bridge
ARM peripheral bus (max 36 Mhz)
Power Supply reg 1.8v
Xtal oscillators 32khz + 4-16 Mhz
Internal RC oscillators 32khz + 4-16 Mhz
PLL
Real time clock Auto wake up
Bridge
(6x) 16 bits PWM
up to 16 ext. IT
51 GPI/O
(1x) SPI bus
(1x) USART/LIN smartcard Irda modem control
ARM peripheral bus (max 72 Mhz)
(3x) Timers 16 bits
(2x) Watchdog
(2x) ADC 12 bits 16 chanels / 1Msps
Temperature Sensor
(1x) USB 2.0
(1x) CAN 2.0
(2x) USART/LIN smartcard Irda modem control
(2x) I²C bus
(1x) SPI bus

Figure 1.3. *Functional description of the STM32F103RB*

In looking at the outline of the STM32F103RB processor, we can see that it has ARM design elements in its processor core, namely:

– a power stage and so clock circuits. This makes it possible to put the processor into "sleep" mode and allow access to different frequencies (which is interesting for timer management in particular);

– the addition of *Flash* and *RAM*. The amount of memory present in the case is one of the variables that fluctuates the most, depending on which μcontroller is chosen. The STM32F103RB version has 128 KB of *Flash* memory which, to put it in perspective, can represent up to 65,000 lines of code written in assembly language (assuming that an instruction is coded on 16 bits);

– managers developed in system time, including the *Systick* 24-bit timer and an automatic wake-up system when the processor has gone into "sleep" mode. These units have an undeniable usefulness during the use of the real-time kernel.

STMicroelectronics then added the following functions:

– time and/or counting management peripherals;

– analog signal management peripherals. These are analogical/numerical converters for acquiring analog quantities, and represent the PWM (Pulse Width Modulation) managers for sending assimilable signals to the analog quantities;

– digital input/output peripherals. These 51 General Purpose Input Output (GPIO) peripherals are the TTL (Transistor-Transistor Logic) input/output ports and the other 16 signals involving digital input that can transmit an interrupt request;

– communication peripherals. Different current protocols are present in this chip for communications via a serial link (Universal Synchronous Asynchronous Receiver Transmitter (USART)), Universal Serial Bus (USB) or an industrial bus (I^2C, Controller–area network (CAN), Serial Peripheral Interface (SPI)).

1.2.2.2. *Memory space*

Memory space management is certainly the most complicated aspect to be managed when developing a program in assembly language. Fortunately, a number of *assembly directives* associated with a powerful *linker* make this management relatively simple. However, it is helpful to have a clear idea of the memory mapping in order to work with full knowledge of the facts while developing (and debugging) the program. In fact, processor registers regularly contain quantities that correspond to addresses. When the program does not behave as desired (a nasty bug, clearly!), it is helpful to know whether the quantities produced are ones that could be expected. Cortex-M3 has a 4 GB consecutive address memory space (32-bit bus). A memory address corresponds to one byte**.** It follows that a *half-word* occupies two addresses and a *word* (32-bits) occupies four memory addresses.

By convention, data storage is arranged according to the *little endian* standard, where the least significant byte of a word or half-word is stored at the lowest address, and we return to the higher addresses by taking the series of component bytes making up the numbers stored in memory. Figure 1.4 shows how, in the *little endian* standard, memory placement of words and half-words is managed.

The architecture is of the *Harvard*-type, which results in a division separating code access from data access. Figure 1.5 shows how this division is planned out. The other zones (*Peripheral, External*, etc.) impose the placement on the addressing space of different units, as presented in Figure 1.3.

One feature that should be noted concerns memory access – *bit banding*. This technique is found in both the *Static Random-Access Memory* (SRAM) zone (between addresses *0×20000000* and *0×2000FFFF* for the *bit-band region* and addresses *0×22000000* and *0×23FFFFFF* for the *alias*) and the *peripheral* zone (between addresses *0×40000000* and *0×4000FFFF* for the *bit-band region* and addresses *0×42000000* and *0×43FFFFFF* for the *alias*). These four zones are schematized by the hatching in the memory-mapping in Figure 1.5. This technique allows the programmer to directly modify (set or reset) bits situated at the addresses within the *bit-banding* zone.

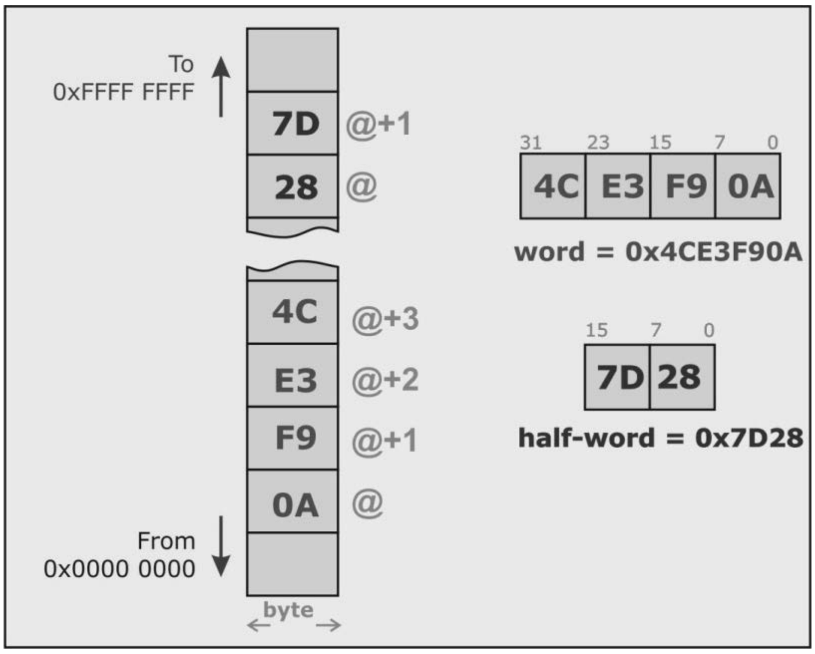


Figure 1.4. *Little-endian convention*

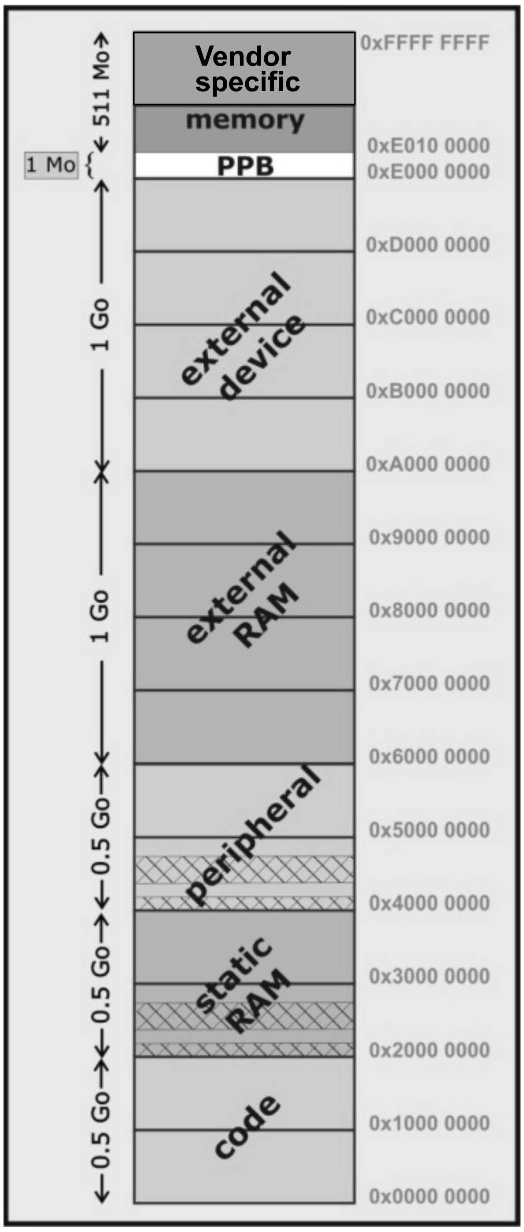


Figure 1.5. *Cortex-M3 memory mapping*

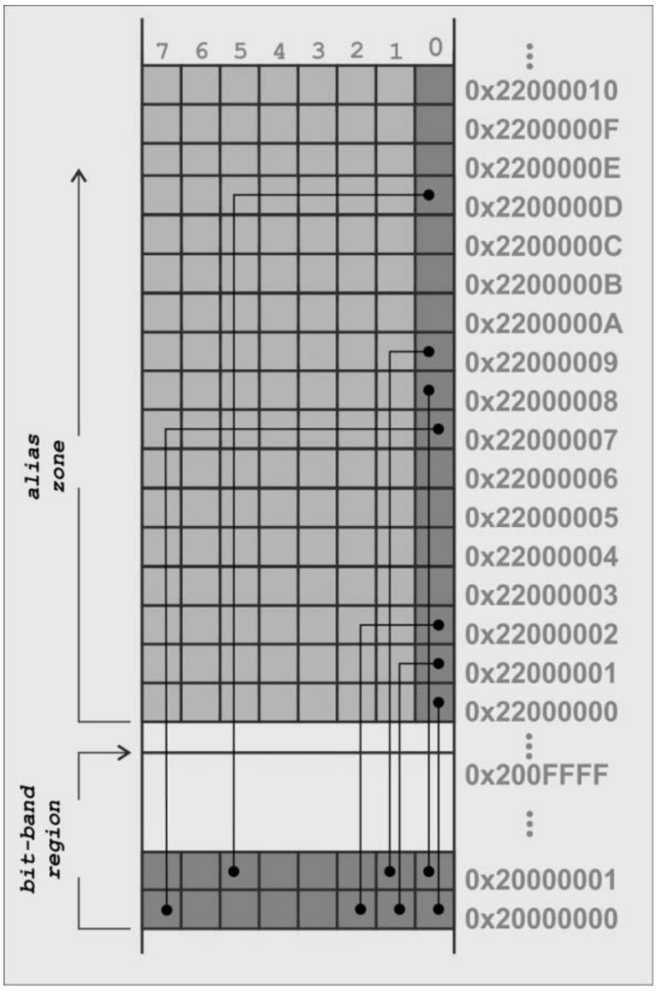


Figure 1.6. *Bit-banding principle for the first SRAM address*

What is the problem? Insofar as the architecture cannot directly act upon bits in memory, if we wish to modify a bit in a memory zone without this feature, it is necessary to:

1) recover the word from memory;

2) modify the bit (by application of a binary mask, for example);

3) rewrite the word to memory.

ARM is designed to match the address of a word (in the alias zone) with a bit (in the bit-banding zone). So when the programmer writes a value in the alias zone, it amounts to modifying the bit-banding bit corresponding to the zero-weight bit that they just wrote. Conversely, reading the least significant bit of a word in the alias zone lets the programmer know the logic state of the corresponding bit in the bit-banding zone (see Figure 1.6). It should be noted that this technique does not use RAM memory insofar as alias zones are imaginary: they do not physically correspond to memory locations – they only use memory addresses, but with 4 GB of possible addresses this loss is of little consequence.