香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

ECE3080
Microprocessors and Computer Systems

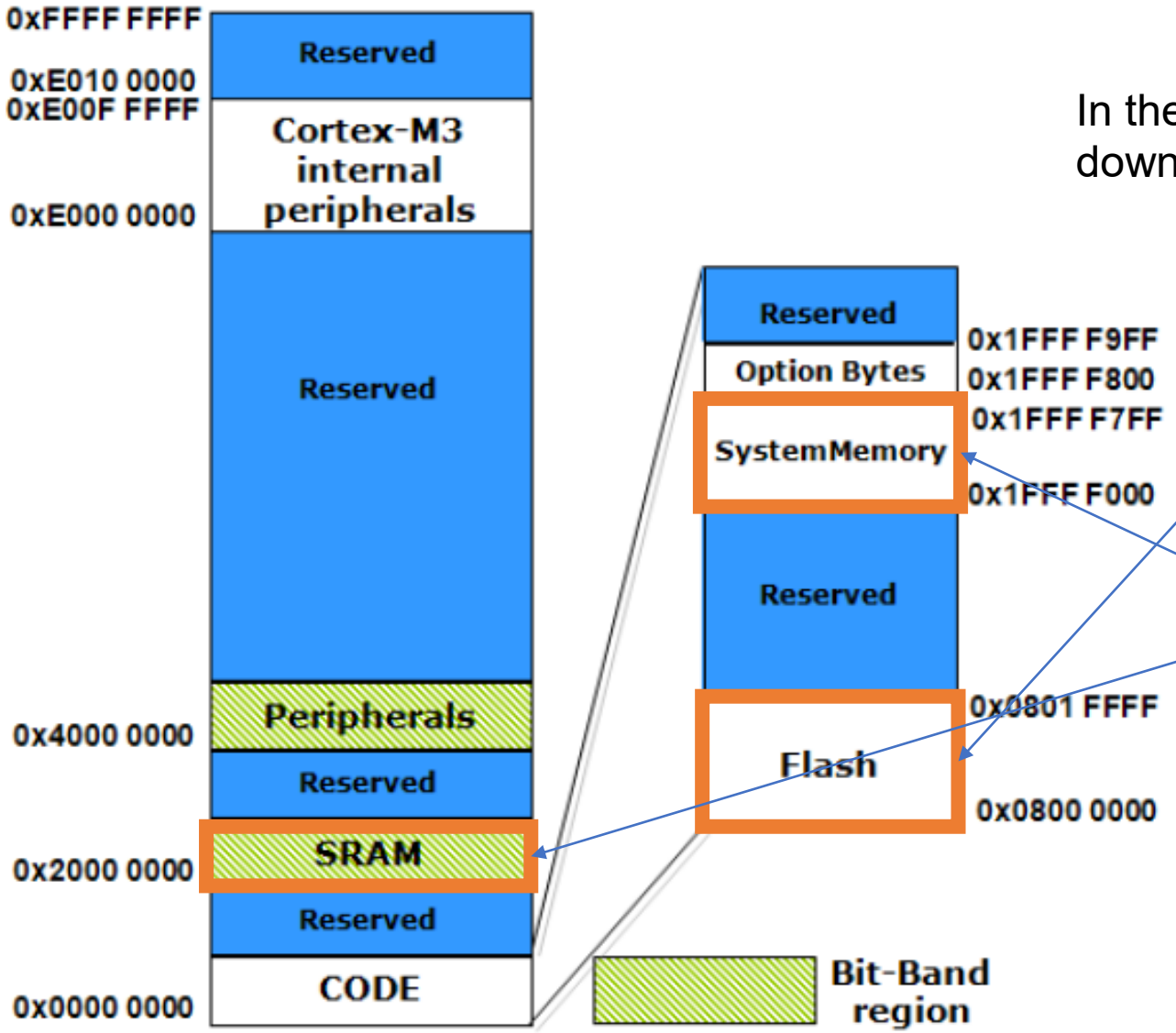
Embedded System Programming

Instructor：Tin Lun LAM

E-mail: tllam@cuhk.edu.cn

URL: https://myweb.cuhk.edu.cn/tllam

How does Cortex-M3 program start?



In the lab experiments, 'System Memory' is used for downloading program; 'User Flash' for user programs.

BOOT Mode Selection Pins		Boot Mode	Aliasing
BOOT1	BOOT0		
x	0	User Flash	User Flash is selected as boot space
0	1	SystemMemory	SystemMemory is selected as boot space
1	1	Embedded SRAM	Embedded SRAM is selected as boot space

The STM32 memory map follows the Cortex standard. The first 2K of memory is mapped from FLASH, System Memory or SRAM depending on the condition of the boot pins.

Booting from system memory (bootloader mode)

- For bootloader mode, the external BOOT1= “0” and BOOT0= “1” .
- Code is downloaded over USART1 port into User FLASH memory.
- The system memory block will appear as (alias) starting at 0x00000000.
- In bootloader mode, after a reset, bootloader code (rather than the code in the User FLASH) will be executed.
- Note: A bootloader application for the PC can be downloaded from the ST website. This program will communicate with the bootloader code and can be used to erase and reprogram the User FLASH memory.

Table 1. Boot pin configuration

Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
X	0	User Flash memory	User Flash memory is selected as the boot space
0	1	System memory	System memory is selected as the boot space
1	1	Embedded SRAM	Embedded SRAM is selected as the boot space

Booting from SRAM

- When BOOT1= “1” and BOOT0= “1”, the internal SRAM is mirrored at 0x00000000.
- This allows programs under development to be downloaded and executed from the internal SRAM ->

  speeds up the download process and saves continual burning of the FLASH memory.

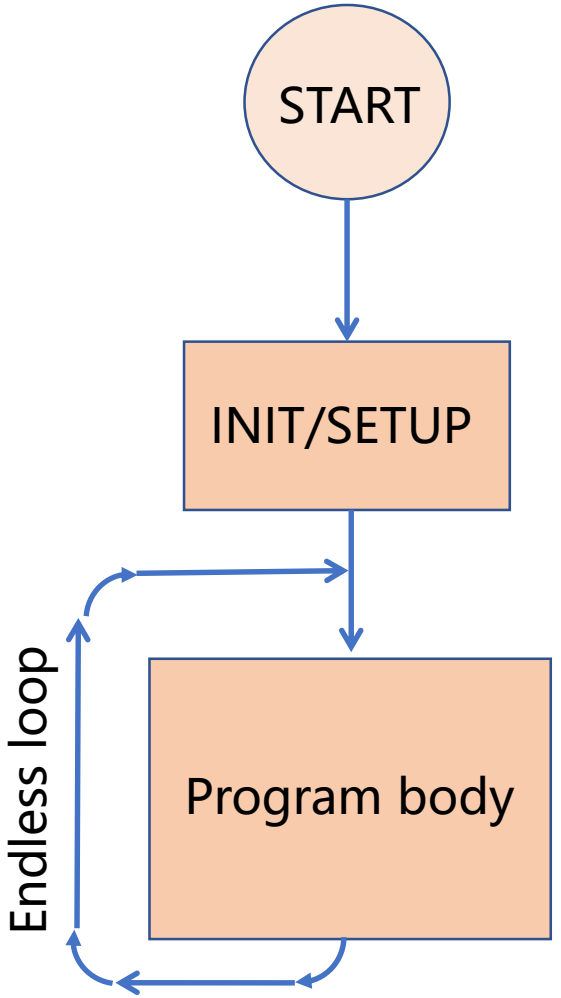
Note: Flash memory program/erase (P/E) cycle: typical 1,000 to 100,000 cycles (depending on the types, like MLC/SLC). SSD uses endurance (in TB) to indicate its life span. For instance, Samsung 850 Evo's 120GB and 250GB capacities have an endurance rating of 75TB. This means if you write 40GB per day to the drive every day (may overwrite the old one), it can last for 5 years.

Table 1. Boot pin configuration

Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
X	0	User Flash memory	User Flash memory is selected as the boot space
0	1	System memory	System memory is selected as the boot space
1	1	Embedded SRAM	Embedded SRAM is selected as the boot space

Embedded system Program flow

- This is a typical embedded system program -- *starts after power on and never ends*.
- Before the program body, there should be some initialization steps.
- Note: This type of program flow will be used in your labs experiments and mini-project.
- Q: What will happen if the endless loop is not there?
- A: The processor will execute the meaningless codes in memory space after the end of your program code.



Endless loop

```
while (1)
{



}
```

Programming in Cortex-M3

- Assembly language, C language, or other high-level languages like National Instruments LabVIEW, can be used.
- The software can be written entirely in C language.
- Assembly language or a combination of C and assembly language can also be used.
- The procedure of building and downloading the resultant image files to the target device mainly depends on the tool chain used.
- As most microcontroller vendors provide device driver libraries written in C to control peripherals, programming in C for Cortex-M3 processor is even easier. (remember the discussion in lab tutorial-1: using function library vs. setting registers)
- For beginners, it is recommended to use C language for software development. (implement your own complex routines in assembly language can be error prone and less portable).

Contents from Text book

Programming in Cortex-M3

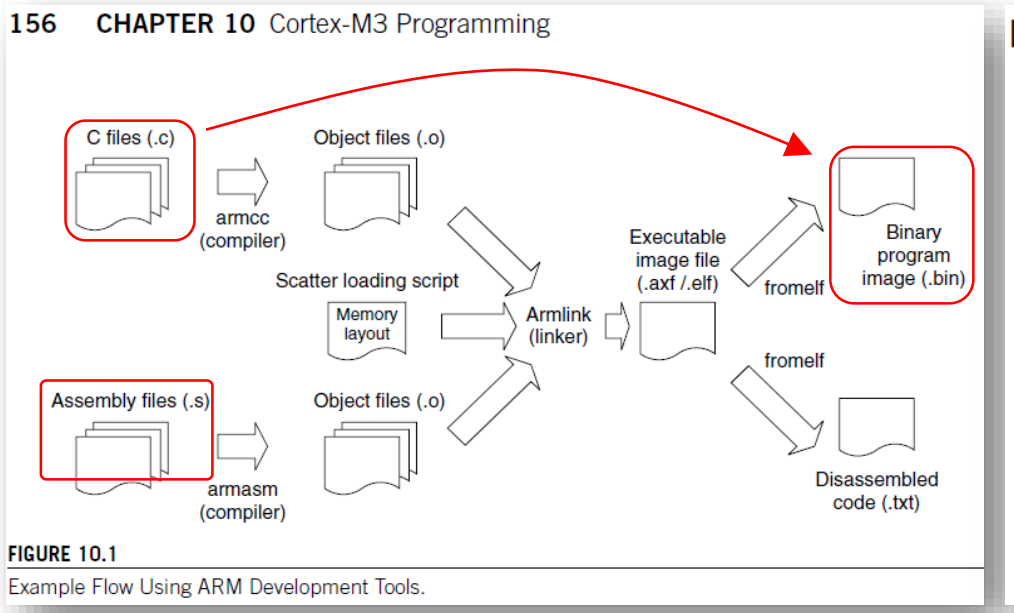
- Cases that you may want to include Assembly codes in your C program
  - Functions that cannot be implemented in C, such as direct manipulation of stack data or special instructions that cannot be generated by the C compiler in normal C-code
  - Timing-critical routines
  - Tight memory requirements; part of the program written in Assembly to reduce memory size

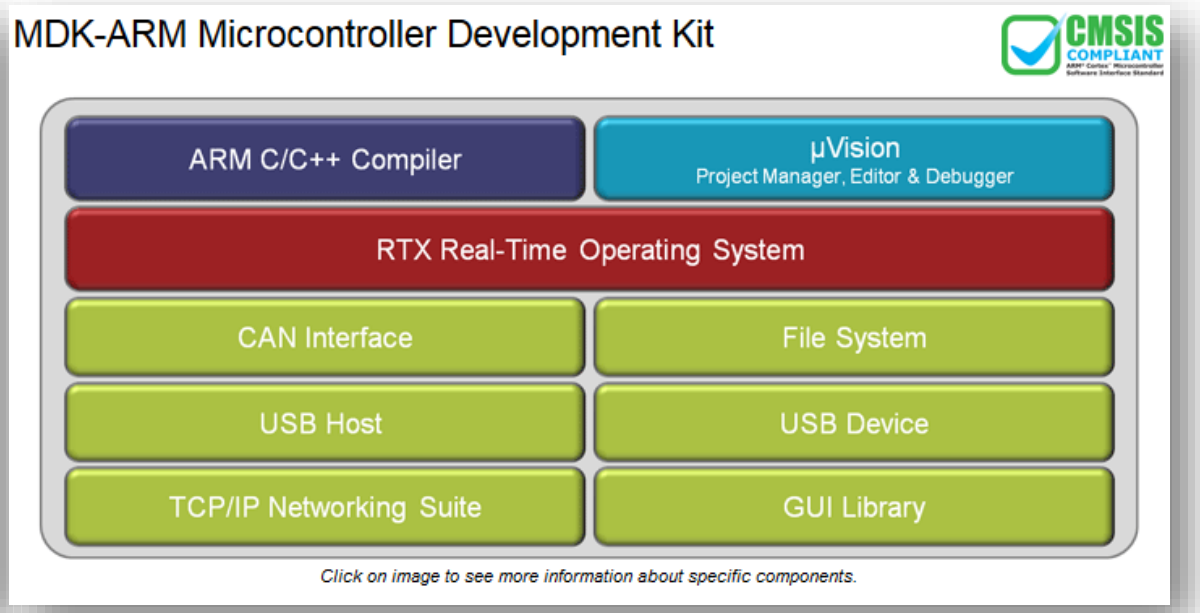
Contents from Text book

Programming in Cortex-M3

- We use KEIL for lab experiments and project.
- Compiler, assembler, linker and others powerful functions are included in MDK-ARM.
- From '.c' to '.bin', you just have to press an icon in uVision.

KEIL™ Tools by ARM





CMSIS (ARM Cortex Microcontroller Software Interface Standard)

- There are quite a number of vendors providing different embedded software solutions, including codecs, data processing libraries, and various software and debug solutions.
- Need to standardize the software interfaces across all Cortex-M silicon vendor products.
  → The CMSIS was developed by ARM to provide a standardized software interface.

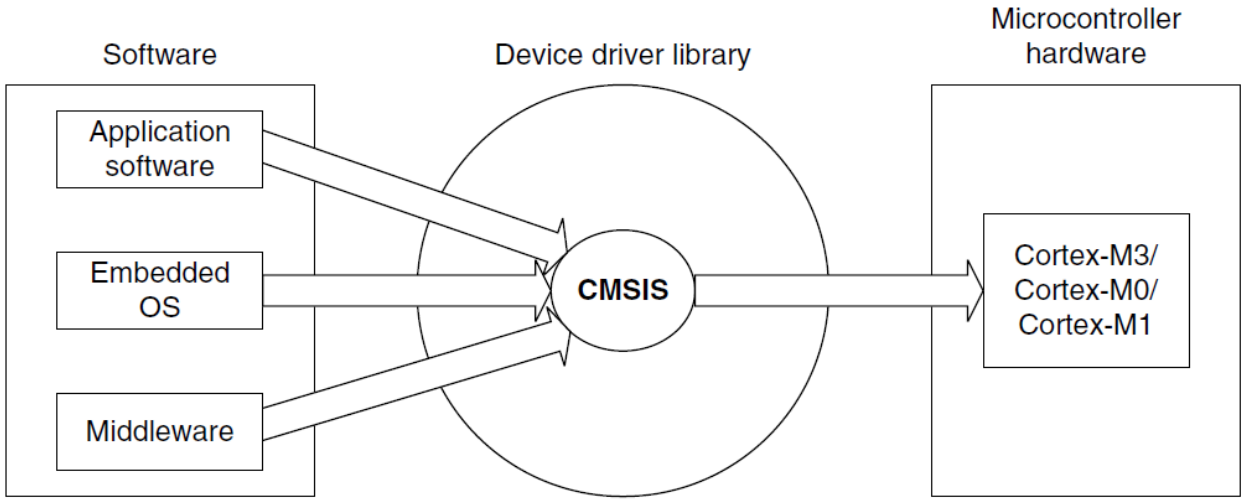


FIGURE 10.6

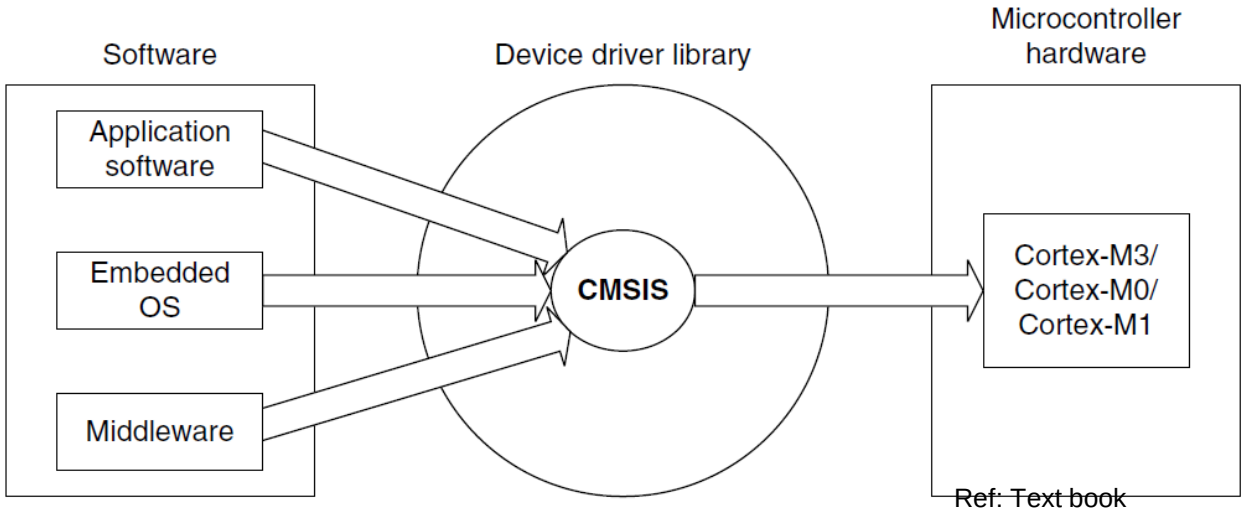
CMSIS Provides a Standardized Access Interface for Embedded Software Products.

CMSIS

The aims of CMSIS are to:

- improve software portability and reusability;
- enable software solution suppliers to develop products that can work seamlessly with device libraries from various silicon vendors;
- allow embedded developers to develop software quicker with an easy-to-use and standardized software interface;
- allow embedded software to be used on multiple compiler products;
- avoid device driver compatibility issues when using software solutions from multiple sources.

(Note that we will focus on programming using register setting, a more fundamental approach, rather than CMSIS library.)



Ref: Text book

FIGURE 10.6

CMSIS Provides a Standardized Access Interface for Embedded Software Products.

Register – Example General Purpose I/O

9.5 GPIO and AFIO register maps

The following tables give the GPIO and AFIO register map and the reset values.

Refer to *Table 3 on page 50* for the register boundary addresses.

Table 59. GPIO register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	GPIOx_CRL	CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]		CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
	Reset value	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
0x04	GPIOx_CRH	CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[1:0]		CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
	Reset value	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
0x08	GPIOx_IDR	Reserved																IDRy															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	GPIOx_ODR	Reserved																ODRy															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	GPIOx_BSRR	BR[15:0]																BSR[15:0]															
	Reset value	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	0	0
0x14	GPIOx_BRR	Reserved																BR[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x18	GPIOx_LCKR	Reserved															LCKK	LCK[15:0]															
	Reset value																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

CNFy[1:0]: Port x configuration bits (y= 0 .. 7)

These bits are written by software to configure the corresponding I/O port.

Refer to *Table 20: Port bit configuration table*.

In input mode (MODE[1:0]=00):

00: Analog mode

01: Floating input (reset state)

10: Input with pull-up / pull-down

11: Reserved

In output mode (MODE[1:0] > 00):

00: General purpose output push-pull

01: General purpose output Open-drain

10: Alternate function output Push-pull

11: Alternate function output Open-drain

MODEy[1:0]: Port x mode bits (y= 0 .. 7)

These bits are written by software to configure the corresponding I/O port.

Refer to *Table 20: Port bit configuration table*.

00: Input mode (reset state)

01: Output mode, max speed 10 MHz.

10: Output mode, max speed 2 MHz.

11: Output mode, max speed 50 MHz.

How to modify only some bits in a memory?

- Q: How about changing a bit in one memory?
  - You need to use Read-Modify-Write method with a register, as shown on the right. It takes some number of cycles to read, to set/clear a bit, and then to write.

Scenario:
Without bit-banding

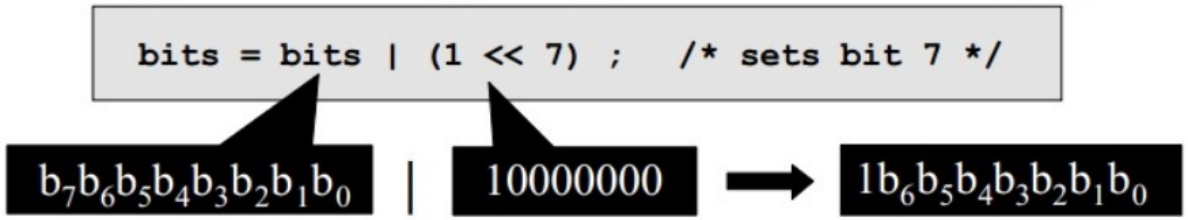
Pictures from Hitex

Basic of bitwise operations

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Setting bits, Inverting bits

Let's say we have variable called **bits** and we have asked to set the bit-7. This can be achieved by writing this single line of code.



Setting Bits using Bitwise Operators

```
bits = bits | (1 << 7) ; /* sets bit 7 */
```

This would usually be written more succinctly as:

```
bits |= (1 << 7) ; /* sets bit 7 */
```

Inverting (toggling) is accomplished with bitwise-XOR. In following, example we'll toggle bit-6.

```
bits ^= (1 << 6) ; /* toggle bit 6 */
```

Clearing Bits

Let's say if we want to clear bit-7. This can be accomplished using bitwise-AND operator

```
bits &= ~(1 << 7) ; /* clears bit 7 */
```

(1 << 7) → 10000000

~(1 << 7) → 01111111

Clear Bits using Bitwise Operators

Mask must be as wide as the operand! if bits is a 32-bit data type, the assignment must be 32-bit:

```
bits &= ~(1L << 7) ; /* clears bit 7 */
```

Note:

|= "or equal" operator

&= "and equal" operator

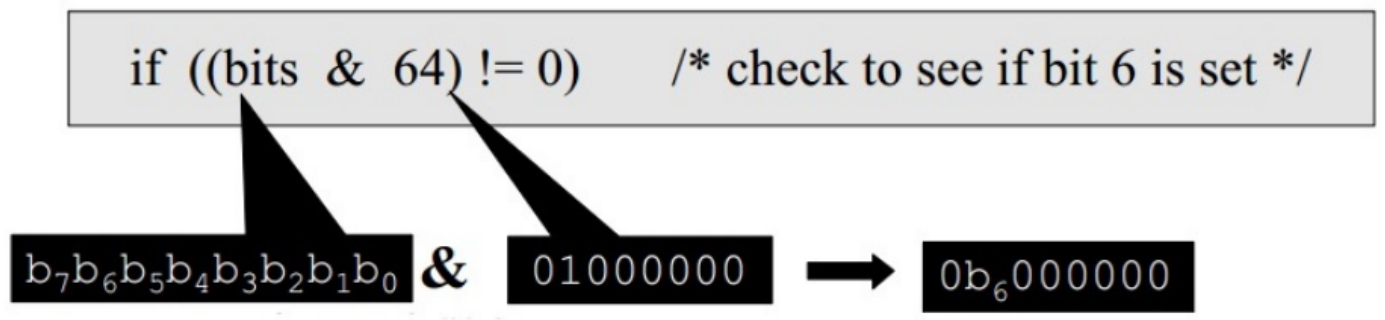
https://binaryupdates.com/bitwise-operations-in-embedded-programming

Basic of bitwise operations

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Testing Bits

Form a mask with **1** in the bit position of interest, in this case bit-**6**. Then bitwise AND the mask with the operand. The result is non-zero if and only if the bit of interest was 1:



Test Bits using Bitwise Operators

```
if ((bits & 64) != 0) /* check to see if bit 6 is set */
```

Same as:

```
if (bits & 0x64) /* check to see if bit 6 is set */
```

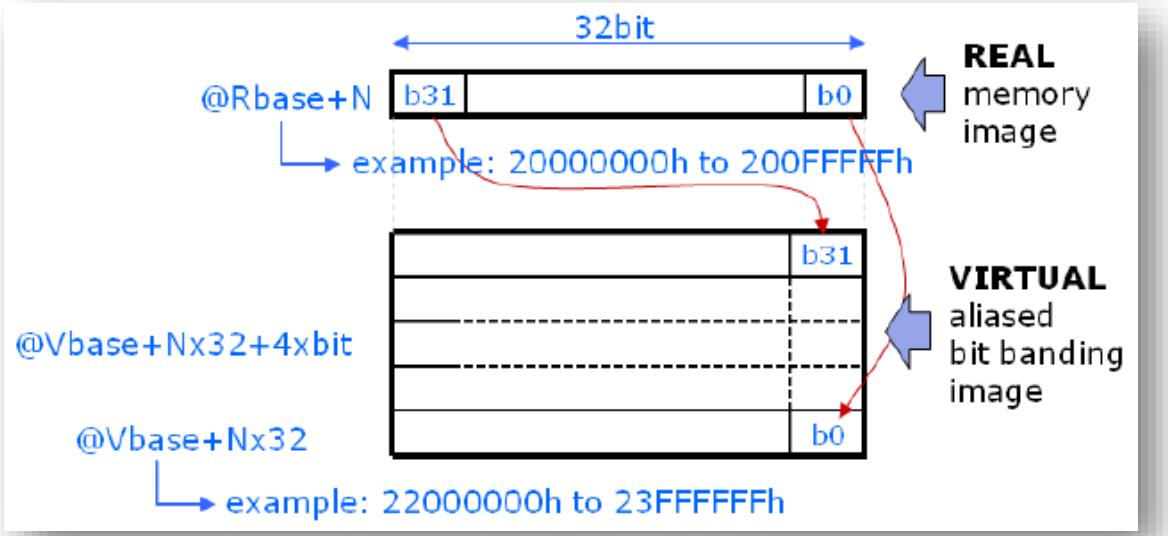
Same as:

```
if (bits & (1 << 6)) /* check to see if bit 6 is set */
```

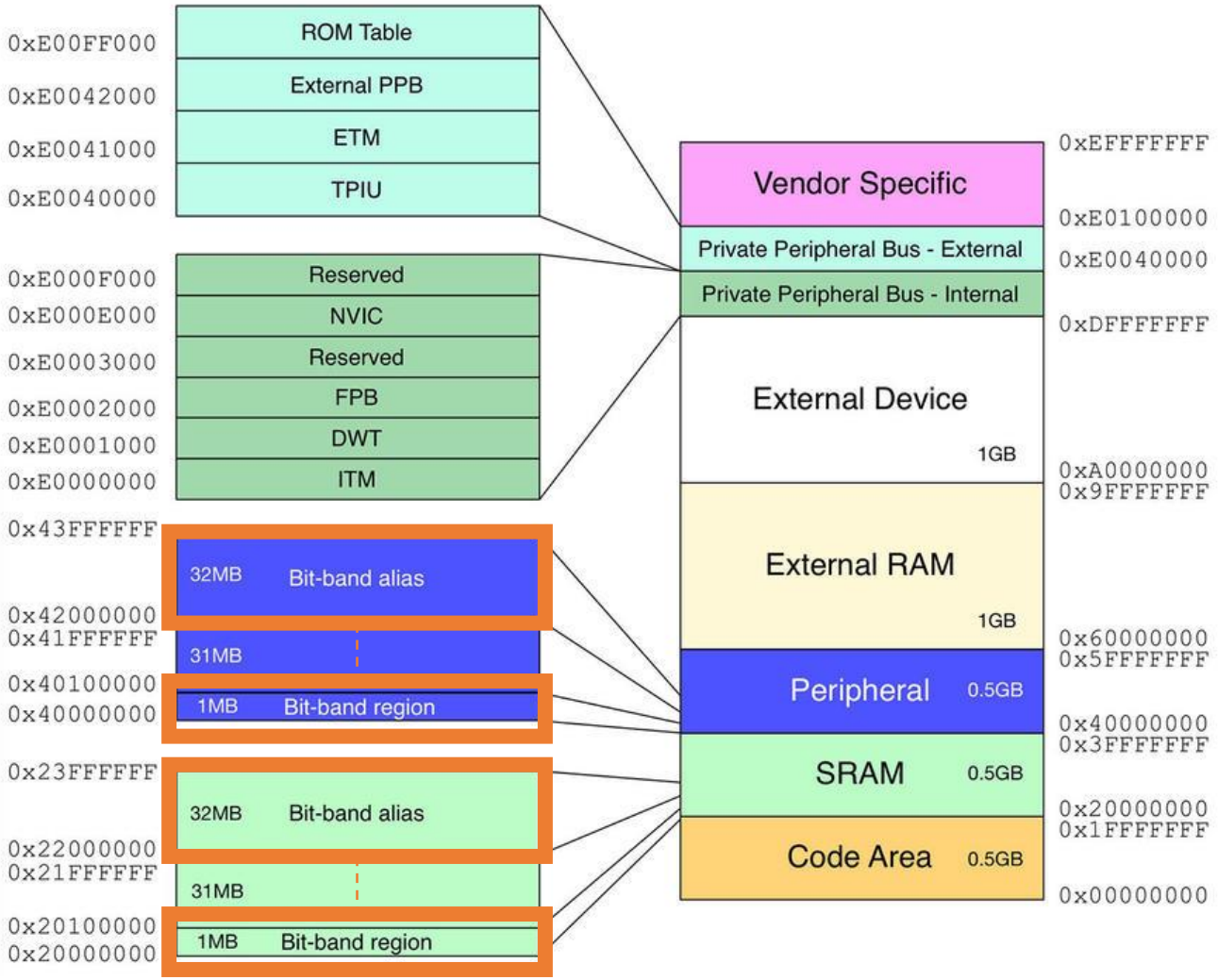
https://binaryupdates.com/bitwise-operations-in-embedded-programming

What is Bit-banding?

- Bit-banding operation allows a single load/store operation to access (read/write) to a single data bit in memory. (compare with Read-Modify-Write 3 steps operation)
- Bit-band is supported in two predefined memory regions (called bit-band regions), located in the first 1 MB of the SRAM region and the peripheral region, respectively.



1MB expand to 32MB



Fixed memory map for ARM Cortex-M cores

Advantages of Bit-Band operation

Avoid data conflict when resource are shared by multiple processes.

e.g., Consider a simple output port with bit 0 used by a main program and bit 1 used by an interrupt handler (interrupt subroutine).

without bit-band

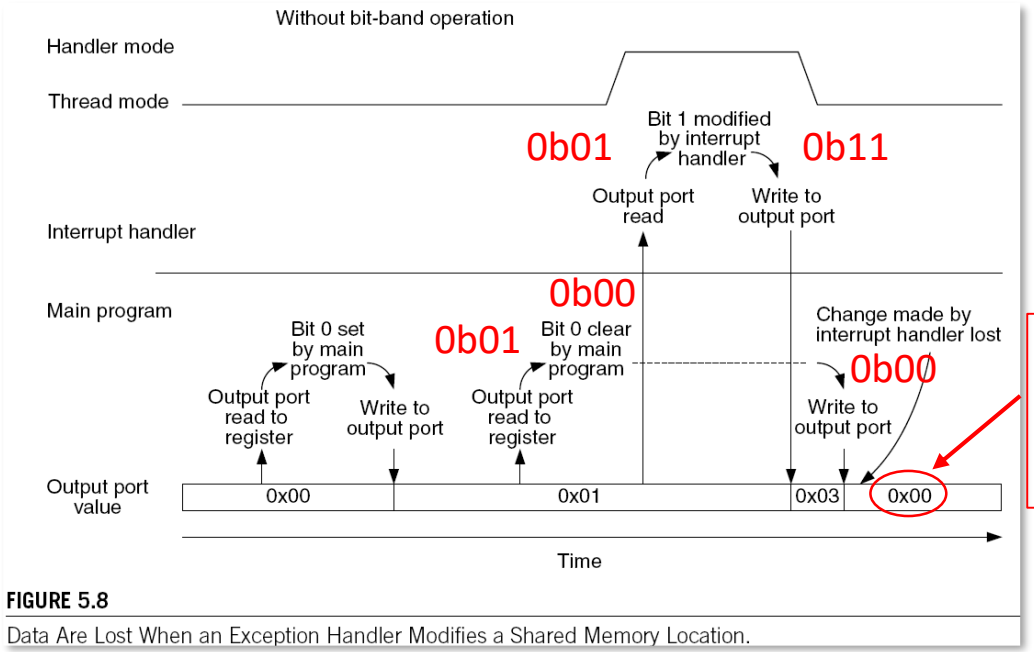


FIGURE 5.8
Data Are Lost When an Exception Handler Modifies a Shared Memory Location.

What should be the correct data here?

Bit-0 set and then reset by main program (using read/modify/write), and bit-1 set by handler (using read/modify/write), just right after the previous reset takes place.

Advantages of Bit-Band operation

- With bit-band, this kind of race condition can be avoided because the READ-MODIFY-WRITE is carried out at the hardware level and is atomic (the two transfers cannot be pulled apart) and interrupts cannot take place between them (see Figure 5.9).

with bit-band

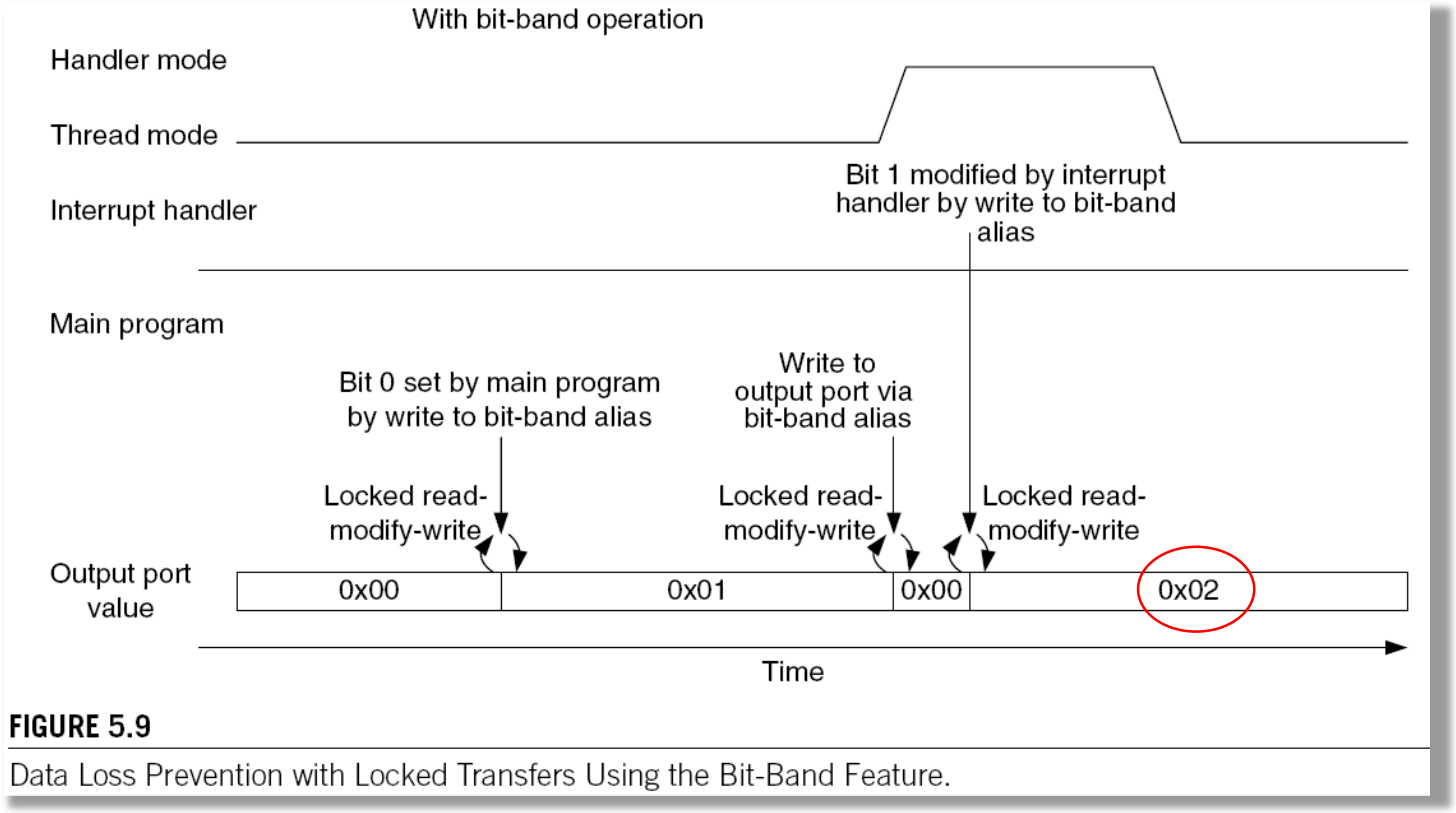


FIGURE 5.9

Data Loss Prevention with Locked Transfers Using the Bit-Band Feature.

Bit-Band Operations in C Programs

- Q: Can we do bit-banding in C?
- A: No native support of bit-band operation in most C compilers.
- To use the bit-band feature in C, the simplest solution is to separately declare the address and the bit-band alias of a memory location.

```
#define DEVICE_REG0          *((volatile unsigned long *) (0x40000000))
#define DEVICE_REG0_BIT0     *((volatile unsigned long *) (0x42000000))
#define DEVICE_REG0_BIT1     *((volatile unsigned long *) (0x42000004))
...
DEVICE_REG0 = 0xAB;          // Accessing the hardware register by
normal
                             // address
...
DEVICE_REG0 = DEVICE_REG0 | 0x2;       // Setting bit 1 without using
                                       // bit-band feature
...
DEVICE_REG0_BIT1 = 0x1;      // Setting bit 1 using bit-band feature
                             // via the bit-band alias address
```

(From Textbook)

Bit-Band Operations

A mapping formula shows how to reference each word in the alias region to a corresponding bit in the bit-band region. The mapping formula is:

$$bit\_word\_addr = bit\_band\_base + (byte\_offset \times 32) + (bit\_number \times 4)$$

where:

bit_word_addr is the address of the word in the alias memory region that maps to the targeted bit.

bit_band_base is the starting address of the alias region

byte_offset is the number of the byte in the bit-band region that contains the targeted bit

bit_number is the bit position (0-7) of the targeted bit.

Example:

The following example shows how to map bit 2 of the byte located at SRAM address 0x20000300 in the alias region:

0x22006008 = 0x22000000 + (0x300*32) + (2*4).

Writing to address 0x22006008 has the same effect as a read-modify-write operation on bit 2 of the byte at SRAM address 0x20000300.

Reading address 0x22006008 returns the value (0x01 or 0x00) of bit 2 of the byte at SRAM address 0x20000300 (0x01: bit set; 0x00: bit reset).

For more information on Bit-Banding, please refer to the *Cortex™-M3 Technical Reference Manual*.

Pictures from RM0008

End