EIE3105: Introduction to ARM and C Programming for ARM

Dr. Lawrence Cheung

Semester 1, 2021/22

Topics

- ARM Cortex-M3 Architecture
- STM32F103RBT6 Architecture
- C Programming for ARM

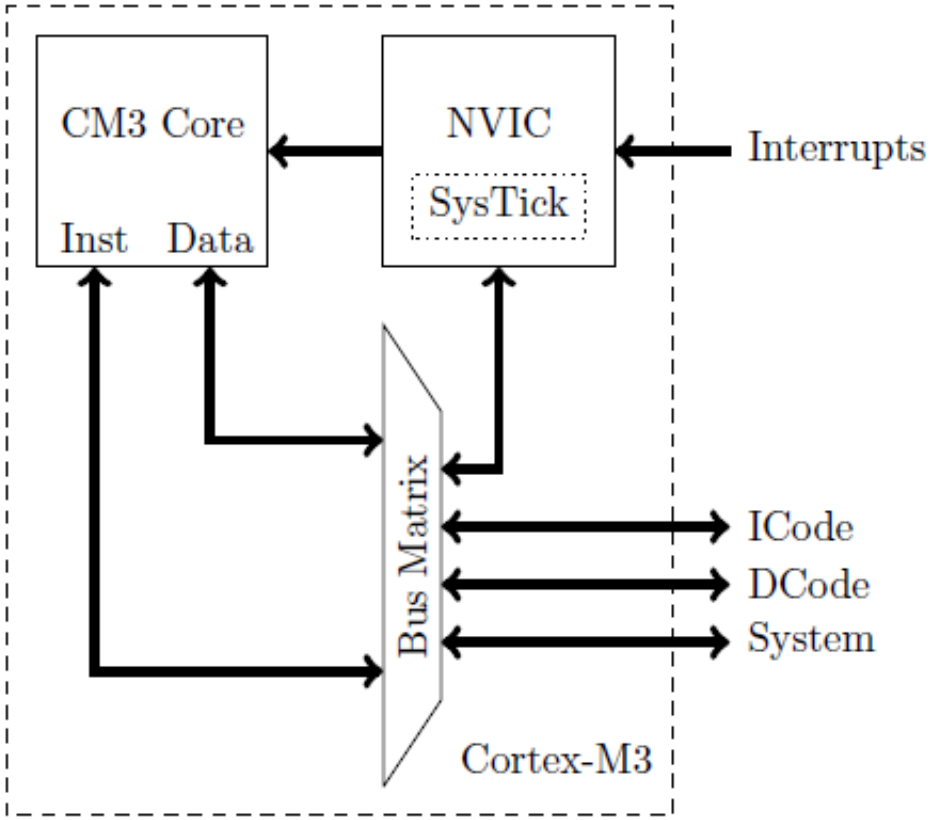
ARM Cortex-M3 Architecture

- The ARM Cortex-M is a group of 32-bit RISC ARM processor cores licensed by ARM Holdings for microcontroller use (from Wikipedia).

Lowest power and area	Performance efficiency	Highest performance
Cortex-M23 TrustZone in smallest area, lowest power	Cortex-M33 Flexibility, control and DSP with TrustZone	Cortex-M7 Maximum performance, control and DSP
Cortex-M0+ Highest energy efficiency	Cortex-M4 Mainstream control and DSP	
Cortex-M0 Lowest cost, low power *Freely available for design and simulation via* DesignStart	Cortex-M3 Performance efficiency	

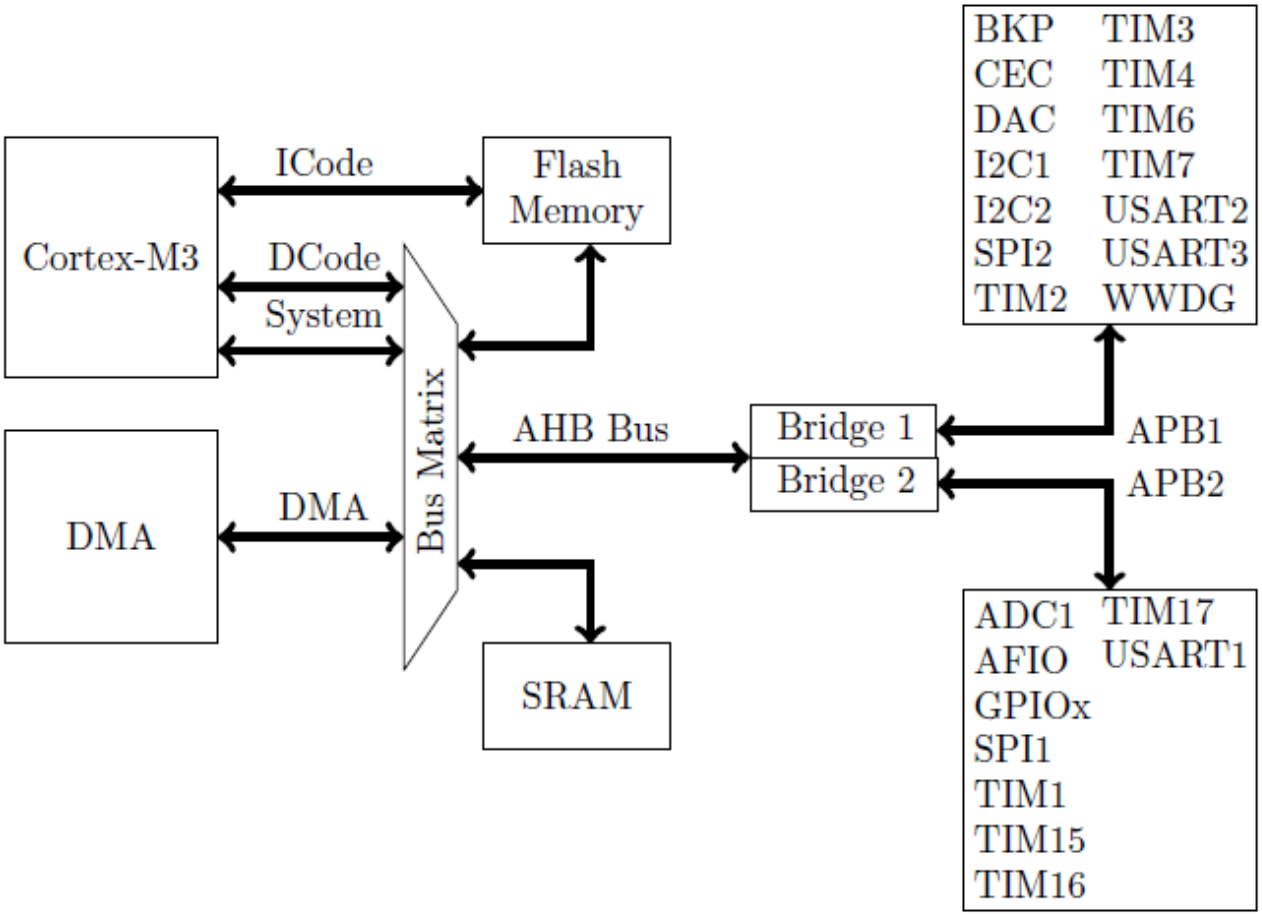
ARM Cortex-M3 Architecture

- Simplified ARM Cortex-M3 architecture



STM32F103RBT6 Architecture

- Simplified STM32F1xx architecture



STM32F103RBT6 Architecture

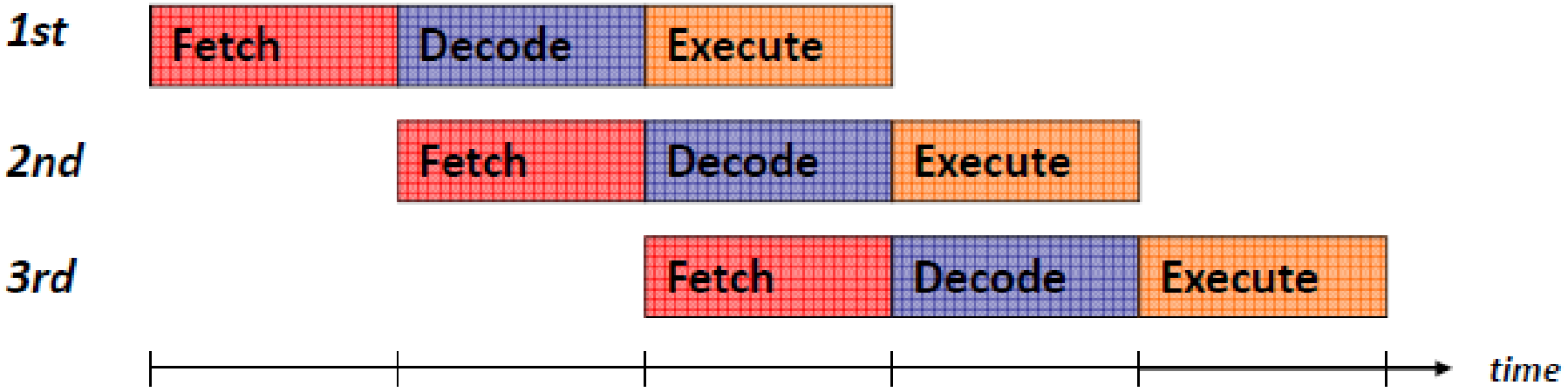
- ICode bus: the instruction bus
- DCode bus: the data bus (flash memory)
- System bus: the peripherals bus
- DMA bus: manage the access of CPU Dcode and DMA (Direct Memory Access) to SRAM, Flash memory and peripherals.
  - Direct memory access (DMA) is a feature of computer systems that allows certain hardware subsystems to access main system memory (Random-access memory), independent of the central processing unit (CPU).

STM32F103RBT6 Architecture

- BusMatrix: manage the access between the system bus and the DMA bus.
- AHB/APB bridges: provide full synchronous connections between the AHB and the two APB buses.
  - AHB: AMBA High-performance Bus
    - AMBA: Advanced Microcontroller Bus Architecture
  - APB: Advanced Peripheral Bus
- NVIC: Nested Vector Interrupt Controller

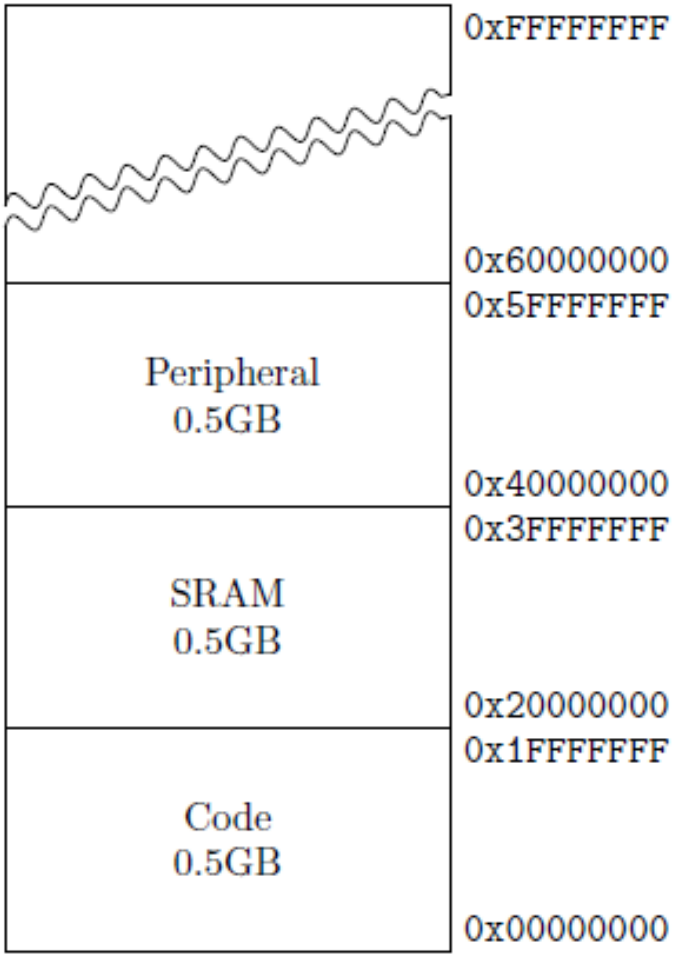
STM32F103RBT6 Architecture

- Processor architecture
  - Harvard: it uses separate interfaces to fetch instructions and data.
  - Processor is not memory starved: it permits accessing data and instruction memories simultaneously.
  - 3-stage pipeline for instruction executions



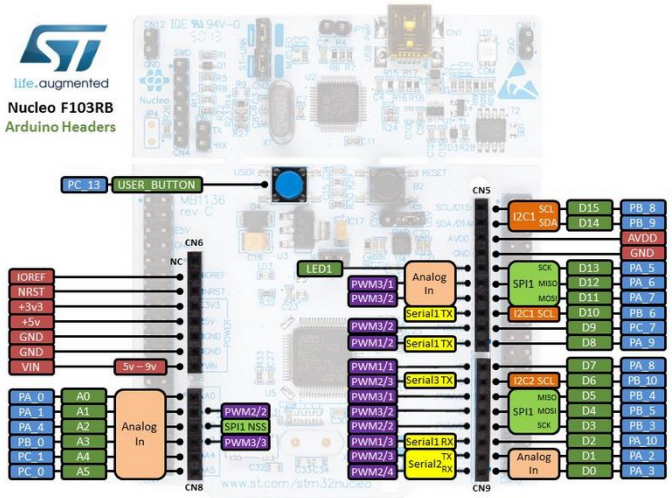
STM32F103RBT6 Architecture

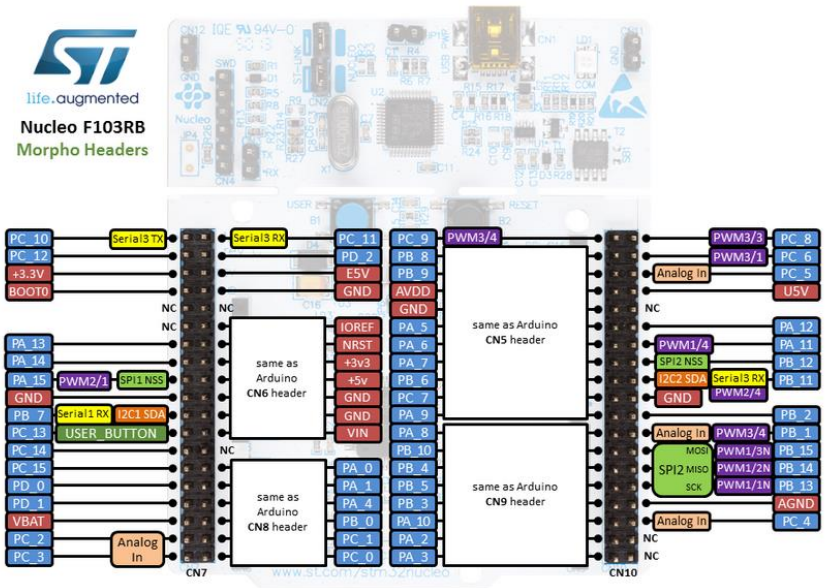
- Memory address space
  - 4GB address space
  - The SRAM and Peripheral areas are accessed through the system bus.
  - The “Code” region is accessed through the ICode (instructions) and DCode (constant data) buses.



STM32F103RBT6 Architecture

- Key features
  - CPU frequency: 72 MHz
  - 128-KB Flash, 20-KB SRAM
  - Two extension connectors: Arduino Uno and ST Morpho





STM32F103RBT6 Architecture

- Timers
  - 3 General-purpose timers
  - 1 Advanced-control timer
- Communication
  - SPI (2), I^2C (2), USART (3), USB (1), CAN (1)
- GPIOs (51) with external interrupt capability
- 12-bit synchronized ADC (2)
  - Number of channels: 16
- Package: LQFP64
  - A surface mount integrated circuit package

C Programming for ARM

- #define
  - The #define directive allows the definition of macros within your source code.
  - These macro definitions allow constant values to be declared for use throughout your code.
  - Example:

```
#define NAME "TechOnTheNet.com"
#define AGE 10
```

C Programming for ARM

- Pointers
  - Powerful, but difficult to master
  - Can be used to perform pass-by-reference.
  - Can be used to create and manipulate dynamic data structures.
  - Close relationship with arrays and strings
    - `char *` pointer-based strings

C Programming for ARM

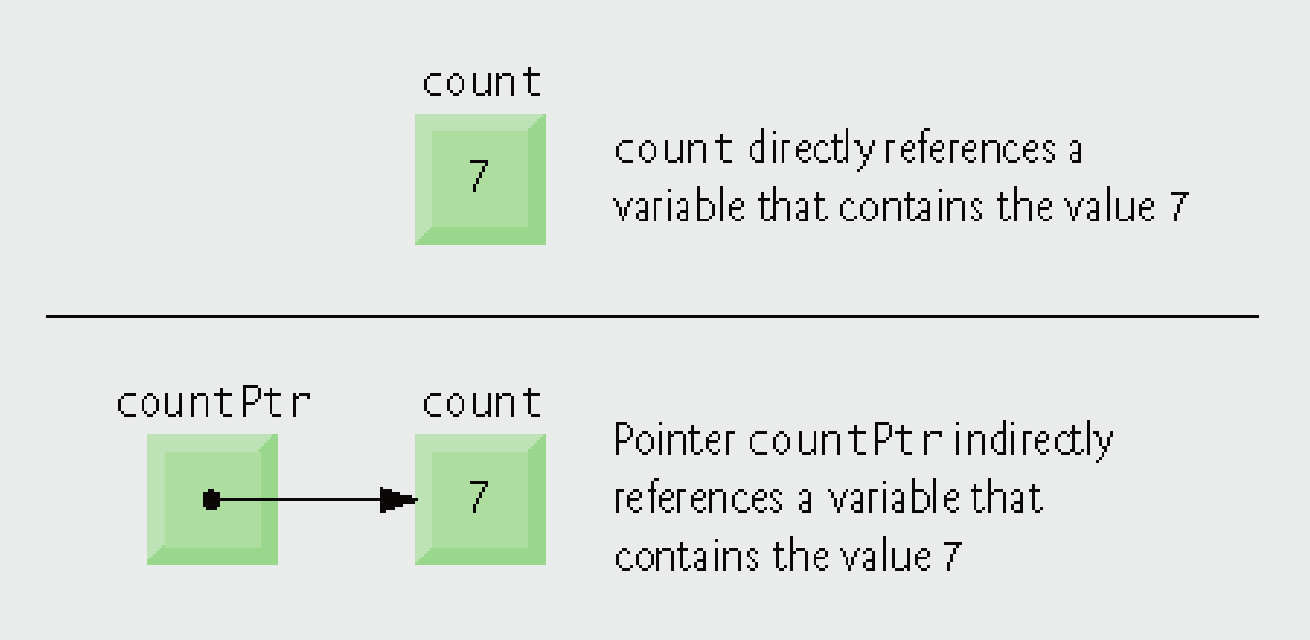
- Pointer variables
  - Contain memory addresses as values.
    - Normally, variable contains specific value (direct reference).
    - Pointers contain address of variable that has specific value (indirect reference).
- Indirection
  - Referencing value through pointer

C Programming for ARM

- Pointer declarations
  - `*` indicates variable is a pointer
    - Example: `int *myPtr;`
    - Declares pointer to `int`, of type `int *`.
    - Note that the name of the pointer variable is `myPtr` but not `*myPtr`.
  - Multiple pointers require multiple asterisks.
    - `int *myPtr1, *myPtr2;`

C Programming for ARM

- Pointer initialization
  - Initialized to `0`, `NULL`, or an address
  - `0` or `NULL` points to nothing (null pointer).

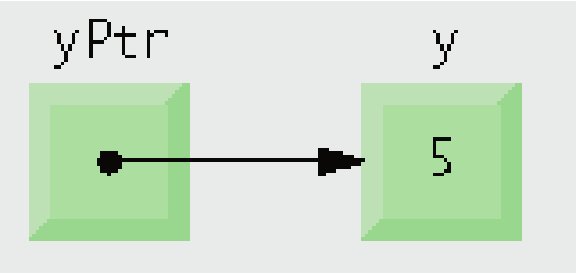


C Programming for ARM

- Address operator (`&`)
  - Returns memory address of its operand.
  - Example: Assign the address of variable `y` to pointer variable `yPtr`.

```
int y = 5;
int *yPtr;
yPtr = &y;
```

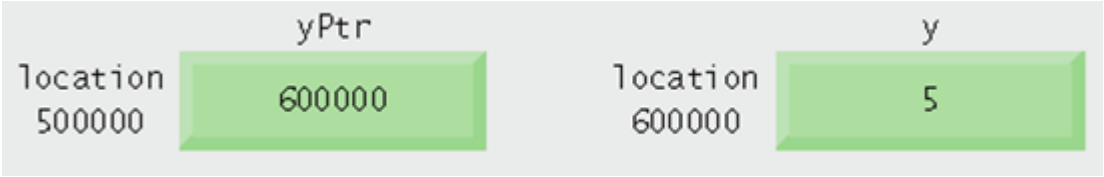
  - Variable `yPtr` "points to" `y`.
  - `yPtr` indirectly references variable `y`'s value.



C Programming for ARM

- `*` operator
  - Also called indirection operator or dereferencing operator
  - Returns synonym for the object its operand points to.
  - `*yPtr` returns `y` (because `yPtr` points to `y`).
  - Dereferenced pointer is an *lvalue*.

```
*yPtr = 5;
```



- `*` and `&` are inverses of each other.
  - Will “cancel one another out” when applied consecutively in either order.

C Programming for ARM

- Example

```
1  // Fig. 8.4: fig08_04.cpp
2  // Using the & and * operators.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int a; // a is an integer
10    int *aPtr; // aPtr is an int * -- pointer to an integer
11
12    a = 7; // assigned 7 to a
13    aPtr = &a; // assign the address of a to aPtr
```

C Programming for ARM

- Example

```
14
15    cout << "The address of a is " << &a
16       << "\nThe value of aPtr is " << aPtr;
17    cout << "\n\nThe value of a is " << a
18       << "\nThe value of *aPtr is " << *aPtr;
19    cout << "\n\nShowing that * and & are inverses of "
20       << "each other.\n&*aPtr = " << &*aPtr
21       << "\n*&aPtr = " << *&aPtr << endl;
22    return 0; // indicates successful termination
23 } // end main
```

```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

C Programming for ARM

- `int a = 7; int *aPtr; aPtr = &a;`

Location (address)	Variable Name	Value
2000	aPtr	0012F580
0012F580	a	7

- `aPtr = &a = 0012F580`
- `*aPtr = a = 7`
- `&aPtr = 2000`

C Programming for ARM

- `&*aPtr = &(*aPtr) = &a = 0012F580`
- `*&aPtr = *(&aPtr) = *(2000) = 0012F580`
- `*aPtr = 9` $\Rightarrow$ `(automatically) a = 9`
- `a = 9` $\Rightarrow$ `(automatically) *aPtr = 9`

C Programming for ARM

- Structure
  - A collection of variables of different data types under a single name.
  - It is similar to a class in that, both holds a collection of data of different data types.
- Example: Store some information about a person: his/her name, mobile phone number and gender (M/F).
- Definitely you can create different variable names to store these information separately.

C Programming for ARM

- A better approach is to have a collection of all related information under a single name `Person`, and use it for every person.

```
struct Person
{
      char name[20];
      int phoneNumber;
      char gender;
};
```

C Programming for ARM

- Declare a structure variable

```
Person boy;
```

- Access its member (field)

```
boy.phoneNumber = 12345678;
```

- Use pointers to access data within a structure

```
Person *ptr;
ptr = &boy;
ptr->phoneNumber = 87654321;
```

C Programming for ARM

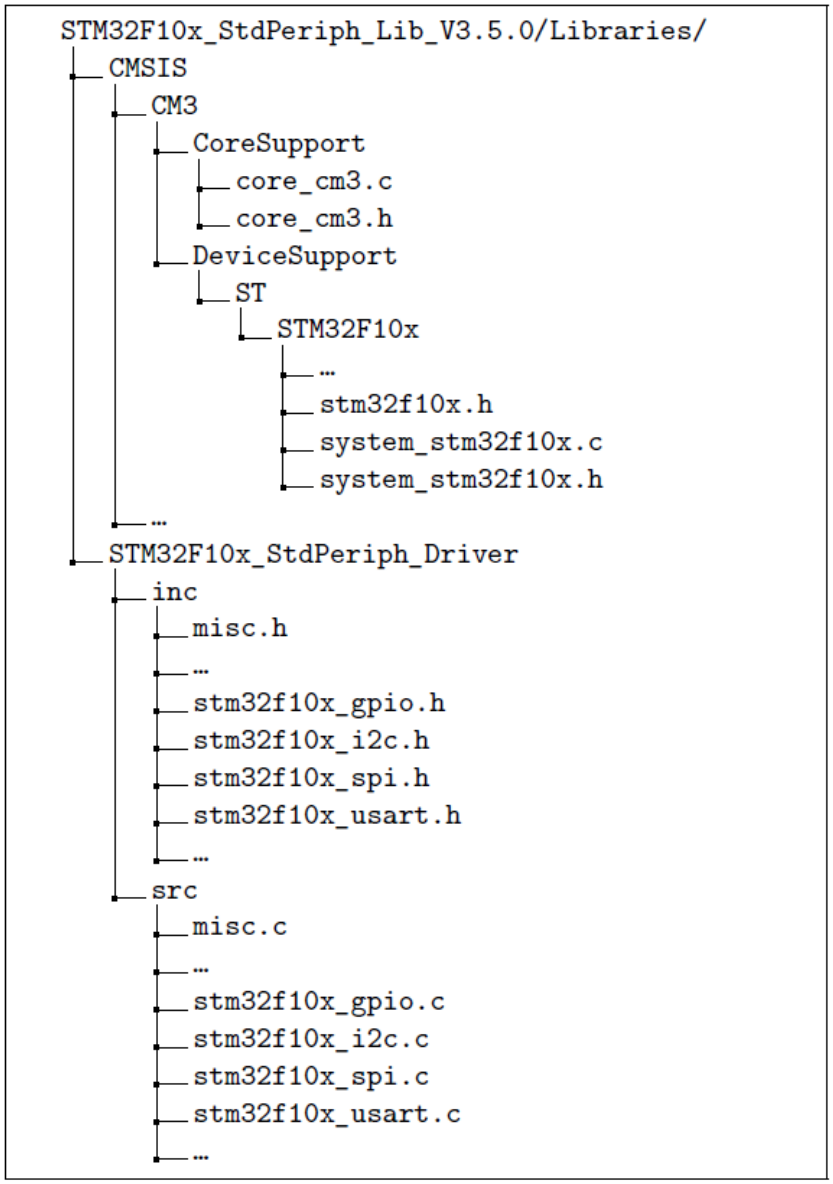
- Example 1: Flash a LED connected to the pin PA5.

```
#include "stm32f10x.h"                        // Device header

void delay(int t) {
    int i;
    for(i = 0; i < t; i++) GPIOA->BSRR |= 0x01; // do something to PA0
}

int main(void) {
    RCC->APB2ENR |= RCC_APB2Periph_GPIOA; //PA5, enable APB2 peripheral clock
    GPIOA->CRL &= ~0x00F00000;  //clear the setting
    GPIOA->CRL |=  0 << 22 | 2 << 20; //GPIO_Mode_Out_PP, GPIO_Speed_2MHz
    while(1) {
      GPIOA->BSRR |= 0x20;
      delay(500000);
      GPIOA->BRR  |= 0x20;
      delay(500000);
    }
}
```

C Programming for ARM

```
#include "stm32f10x.h"
```

- STM32F10x family header file
  - You must include it.



C Programming for ARM

```
RCC->APB2ENR |= RCC_APB2Periph_GPIOA;
```

- RCC = Reset and Clock Control
- Manage system and peripherals clocks and resets
- Resets
  - System reset
  - Power reset
  - Backup domain reset

C Programming for ARM

- Clocks
  - 2 internal oscillators
  - 2 external oscillators (crystal or RC)
  - 3 Phase-locked loops (PPLs)
  - Flexible peripheral clock sources
- All GPIO peripherals works on APB2 bus.
- Bits 2 to 8 are used to enable clock for each GPIO port (form GPIOA to GPIOG).

C Programming for ARM

RCC

- Define in `stm32f10x.h`

```
-> #define RCC ((RCC_TypeDef *) RCC_BASE)
-> #define RCC_BASE (AHBPERIPH_BASE + 0x1000)
-> #define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)
-> #define PERIPH_BASE ((uint32_t)0x40000000)
 /*!< Peripheral base address in the alias region */
```

C Programming for ARM

`RCC->APB2ENR`

- Define in `stm32f10x.h`

```
typedef struct
{
  __IO uint32_t CR;
  __IO uint32_t CFGR;
  __IO uint32_t CIR;
  __IO uint32_t APB2RSTR;
  __IO uint32_t APB1RSTR;
  __IO uint32_t AHBENR;
  __IO uint32_t APB2ENR;
  …
} RCC_TypeDef;
```

C Programming for ARM

`uint32_t`

- unsigned int (32 bits)

```
#define     __IO    volatile
/*!< Defines 'read / write' permissions */
```

- `volatile` is a qualifier that is applied to a variable when it is declared.
- It tells the compiler that the value of the variable may change at any time-without any action being taken by the code the compiler finds nearby.

C Programming for ARM

- A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:
  - Memory-mapped peripheral registers
  - Global variables modified by an interrupt service routine
  - Global variables within a multi-threaded application

C Programming for ARM

`RCC->APB2ENR`

8.3.7 APB2 peripheral clock enable register (RCC_APB2ENR)

Address: 0x18

Reset value: 0x0000 0000

Access: word, half-word and byte access

No wait states, except if the access occurs while an access to a peripheral in the APB2 domain is on going. In this case, wait states are inserted until the access to APB2 peripheral is finished.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	USART1EN	Res.	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	Reserved		IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	Res.	AFIO EN
	rw		rw	rw	rw	rw			rw	rw	rw	rw	rw		rw

C Programming for ARM

```
RCC_APB2Periph_GPIOA;
```

- Definition at line 498 of file stm32f10x_rcc.h.

```
/* STM32F10X_CL */
#define RCC_APB2Periph_GPIOA    ((uint32_t)0x00000004)
```

- Alternative way:

```
#define  RCC_APB2ENR_IOPAEN     ((uint32_t)0x00000004)
/*!< I/O port A clock enable */
```

C Programming for ARM

```
GPIOA->CRL &= ~0x00F00000;  //clear the setting
-> #define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
-> #define GPIOA_BASE (APB2PERIPH_BASE + 0x0800)

typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

C Programming for ARM

- CRH is used to set type/and or speed of pins 8 – 15 of the port.
- CRL is used to set type/and or speed of pins 0 – 7 of the port.
- Accessed as a 32 bit word, with 4 bits representing the state of each pin.
- Out of these 4 bits, the low 2 bits are MODE, and high 2 bits are CNF.

```
GPIOA->CRL &= ~0x00F00000;
```

- Clear Pin 5 of Port A

C Programming for ARM

Port bit configuration table

Configuration mode		CNF1	CNF0	MODE1	MODE0	PxODR register
General purpose output	Push-pull	0	0	01 10 11		0 or 1
	Open-drain		1			0 or 1
Alternate Function output	Push-pull	1	0			don't care
	Open-drain		1			don't care
Input	Analog	0	0	00		don't care
	Input floating		1			don't care
	Input pull-down	1	0			0
	Input pull-up					1

Output MODE bits

MODE[1:0]	Meaning
00	Reserved
01	Max. output speed 10 MHz
10	Max. output speed 2 MHz
11	Max. output speed 50 MHz

C Programming for ARM

```
GPIOA->CRL |=  0 << 22 | 2 << 20;
//GPIO_Mode_Out_PP, GPIO_Speed_2MHz
```

- Bit 20 to 21: PA5 Output Mode
- Mode = 2 (10): Max. output speed 2 MHz
- Bit 22 to 23: PA5 Configuration Mode
- Mode = 0 (00): General purpose output (Push-pull)

C Programming for ARM

- GPIO speed
  - Basically, this controls the slew rate (the rise time and fall time) of the output signal.
  - The faster the slew rate, the more noise is radiated from the circuit (EMI, Electromagnetic Interference).
  - It is good practice to keep the slew rate slow, and only increase it if you have a specific reason.
    - Example: Output high frequency signals but using low speed mode: output signal distortion

C Programming for ARM

- More examples
  - USART port: If the maximum frequency is 115.2 kHz, 2MHz should be enough.
  - I2C port: if 400 kb/s is used but the acceptable range is 10 times, 10MHz is more stable.
  - SPI interface: If the output is 9MHz or 18 MHz is used, 50MHz should be used.

C Programming for ARM

- IDR – Input Data Register
  - Used to read input of entire 16 pins of port at once.
  - Accessed as a 32 bit word whose lower 16 bits represent each pin.
  - The pins being read must be set to INPUT mode by using CRL/CRH or pinMode() before using this.
- ODR-Output Data Register
  - Used to write output to entire 16 pins of port at once.
  - Accessed and written as a 32 bit word whose lower 16 bits represent each pin.
  - The pins being read must be set to OUTPUT mode by using CRL/CRH or pinMode() before using this.

C Programming for ARM

- BRR – Bit Reset Register (Clear)
  - 32 bit word.
  - Lower 16 bits have 1's where bits are to be set to "LOW".
  - Upper 16 bits have 1's where bits are to be set to "HIGH".
- BSRR – Bit Set Reset Register (Set)
  - BSRR is like the complement of BRR.
  - Lower 16 bits have 1's where bits are to be set to "HIGH".
  - Upper 16 bits have 1's where bits are to be set to "LOW".

C Programming for ARM

- LCKR – Lock Register
  - Bit 16 : Lock key
  - 0: Port configuration lock key not active
  - 1: Port configuration lock key active
  - Bits 15:0 LCKy: lock bit y
  - 0: Port configuration not locked
  - 1: Port configuration locked

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															LCKK
															rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

C Programming for ARM

- GPIO
  - PA0 to PA15
  - PB0 to PB15
  - PC0 to PC15
    - PC13 to PC15 have limitations:
      - Only one can be an output at the same time
      - 2 MHz
      - No current source (e.g., to drive LEDs)
  - PD0 to PD2
  - Total: 51 pins

C Programming for ARM

- Example 2: Flash a LED connected to the pin PA5 (using Functions and System clock tick).

```
#include "stm32f10x.h"                    // Device header

static __IO uint32_t msTicks;

void DelayMs(uint32_t ms)
{
    msTicks = ms; // Reload us value
    while (msTicks); // Wait until msTicks reaches zero
}

// SysTick_Handler function will be called every 1 ms
void SysTick_Handler()
{
    if (msTicks != 0)
        msTicks--;
}
```

C Programming for ARM

```
int main(void) {
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); // Enable APB2

    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_5;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    SystemCoreClockUpdate(); // Update SystemCoreClock value

    // Configure the SysTick timer to overflow every 1 ms
    SysTick_Config(SystemCoreClock / 1000);

    while(1) {
        GPIO_WriteBit(GPIOA, GPIO_Pin_5, Bit_SET);
        DelayMs(500);
        GPIO_WriteBit(GPIOA, GPIO_Pin_5, Bit_RESET);
        DelayMs(500);
    }
}
```

C Programming for ARM

- The timer is a multiple of the system clock.
  - Define in ticks/second (number of system clock cycles per second)
  - Here we configure it for an 1 ms interrupt.
  - Every 1 ms, the timer triggers a call to the `SysTick_Handler`.

C Programming for ARM

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
```

- STM32 peripherals are organized into three distinct groups
  - APB1 peripherals: I2C devices, USARTs 2 – 5 and SPI devices
  - APB2 devices: GPIO ports, ADC controllers and USART 1
  - AHB devices: DMA controllers and external memory interfaces

C Programming for ARM

- Clocks to various peripherals

```
RCC_APB1PeriphClockCmd(uint32_t RCC_APB1PERIPH,
                       FunctionalState NewState)
RCC_APB2PeriphClockCmd(uint32_t RCC_APB2PERIPH,
                       FunctionalState NewState)
RCC_AHBPeriphClockCmd(uint32_t RCC_AHBPERIPH,
                       FunctionalState NewState)
```

- Example:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |
                       RCC_APB2Periph_GPIOB, ENABLE);
```

C Programming for ARM

- Clock distribution constant names (`stm32f1xx_rcc.h`)

APB1 Devices	APB2 Devices
RCC_APB1Periph_BKP	RCC_APB2Periph_ADC1
RCC_APB1Periph_CEC	RCC_APB2Periph_AFIO
RCC_APB1Periph_DAC	RCC_APB2Periph_GPIOA
RCC_APB1Periph_I2C1	RCC_APB2Periph_GPIOB
RCC_APB1Periph_I2C2	RCC_APB2Periph_GPIOC
RCC_APB1Periph_PWR	RCC_APB2Periph_GPIOD
RCC_APB1Periph_SPI2	RCC_APB2Periph_GPIOE
RCC_APB1Periph_TIM2	RCC_APB2Periph_SPI1
RCC_APB1Periph_TIM3	RCC_APB2Periph_TIM1
RCC_APB1Periph_TIM4	RCC_APB2Periph_TIM15
RCC_APB1Periph_TIM5	RCC_APB2Periph_TIM16
RCC_APB1Periph_TIM6	RCC_APB2Periph_TIM17
RCC_APB1Periph_TIM7	RCC_APB2Periph_USART1
RCC_APB1Periph_USART2	
RCC_APB1Periph_USART3	
RCC_APB1Periph_WWDG	
AHB Devices	
RCC_AHBPeriph_CRC	RCC_AHBPeriph_DMA

C Programming for ARM

- GPIO initialization
  - In header file `stm32f10x_gpio.h`

```
typedef struct
{
    uint16_t GPIO_Pin;
    GPIOSpeed_TypeDef GPIO_Speed;
    GPIOMode_TypeDef GPIO_Mode;
}GPIO_InitTypeDef;
```

C Programming for ARM

- GPIO_Pin = the GPIO pin to be configured
  - Values: GPIO_Pin_0 to GPIO_Pin_15
  - Value: GPIO_Pin_ALL (all pins selected)
- GPIO_Speed = maximum frequency selection
  - Values: GPIO_Speed_10MHz, GPIO_Speed_2MHz and GPIO_Speed_50MHz

C Programming for ARM

– GPIO_Mode = configuration mode

Function	Library Constant
Alternate function open-drain	`GPIO_Mode_AF_OD`
Alternate function push-pull	`GPIO_Mode_AF_PP`
Analog	`GPIO_Mode_AIN`
Input floating	`GPIO_Mode_IN_FLOATING`
Input pull-down	`GPIO_Mode_IPD`
Input pull-up	`GPIO_Mode_IPU`
Output open-drain	`GPIO_Mode_Out_OD`
Output push-pull	`GPIO_Mode_Out_PP`

C Programming for ARM

- Input floating: no hardware conflicts will occur when the system is powering up.
- Analog: analog input (directly input the signal into the CPU)
- Alternative Functions: any outputs required by the peripheral must be configured to an “alternative mode”.
  - The Tx pin (data out) for USART1 is configured as follows

```
GPIO_InitStruct.GPIO_PIN = GPIO_Pin_9;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_Init(GPIOA, &GPIO_InitStruct);
```

C Programming for ARM

`SystemCoreClockUpdate()`

- Update `SystemCoreClock` variable according to Clock Register Values.

```
SysTick_Config(SystemCoreClock / 1000);
```

- Configure the clock for 1 ms interrupt.

C Programming for ARM

- GPIO operations

```
void GPIO_Init(GPIO_TypeDef * GPIOx,
               GPIO_InitTypeDef * GPIO_InitStruct)
```

  - Initialize the GPIOx peripheral according to the specified parameters in the GPIO_InitStruct.
    - Values of GPIOx: GPIOA to GPIOG

```
void GPIO_WriteBit(GPIO_TypeDef * GPIOx,
                   uint16_t GPIO_Pin, BitAction BitVal)
```

  - Set or clear the selected data port bit.
    - BitVal = specify the value to be written
    - Values: Bit_RESET (clear the pin), BIT_SET (set the pin)

C Programming for ARM

- All GPIO operations

```
void GPIO_Init(GPIO_TypeDef* GPIOx,
               GPIO_InitTypeDef* GPIO_InitStruct);
uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx,
                              uint16_t GPIO_Pin);
uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx);
uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef* GPIOx,
                               uint16_t GPIO_Pin);
uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx);
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin,
                   BitAction BitVal);
void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal);
```

Reference Readings

- Chapter 2 – *The Definitive Guide To The ARM Cortex-M3*, Joseph Yiu, 2nd edition, Newnes, 2010.
- Chapter 2, 3 and 4 – *Discovering the STM32 Microcontroller*, Geoffrey Brown, 2012.
- http://www.longlandclan.yi.org/~stuartl/stm32f10x_stdperiph_lib_um
- RM0008 Reference Manual (STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 320bit MCUs)
- Datasheet – STM32F103x8, STM32F103xB

End