Disclaimer

Arm is committed to making the language we use inclusive, meaningful, and respectful. Our goal is to remove and replace non-inclusive language from our vocabulary to reflect our values and represent our global ecosystem.

Arm is working actively with our partners, standards bodies, and the wider ecosystem to adopt a consistent approach to the use of inclusive language and to eradicate and replace offensive terms. We recognise that this will take time. This course contains references to non-inclusive language; it will be updated with newer terms as those terms are agreed and ratified with the wider community.

Contact us at education@arm.com with questions or comments about this course. You can also report non-inclusive and offensive terminology usage in Arm content at terms@arm.com.

arm

Introduction to Embedded Systems

Module Syllabus

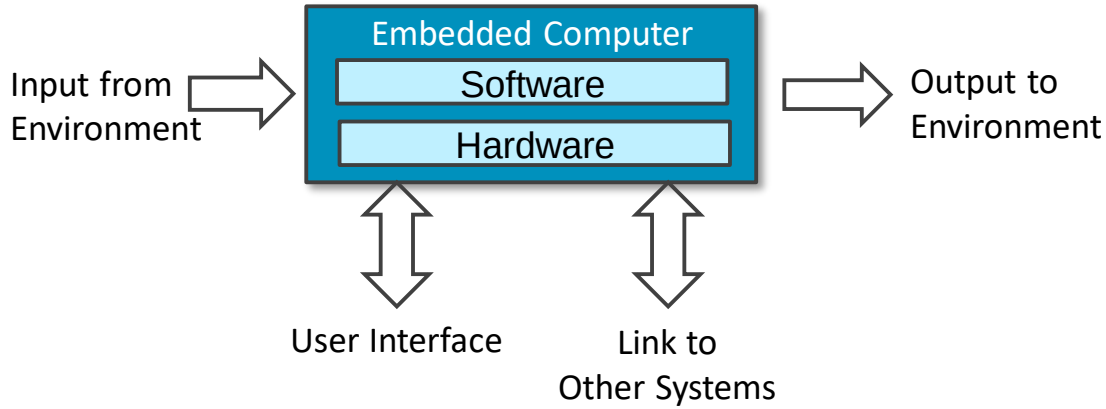
- Introduction to Embedded Systems
  - CPUs vs. MCUs vs. Embedded Systems
  - Examples of Embedded Systems
  - Options for Building Embedded Systems
  - Features of Embedded Systems
- Introduction to the Internet of Things (IoT)
  - What is IoT?
  - Why IoT?
  - Challenges of IoT
- Building Embedded Systems
  - Building Embedded Systems Using MCUs

Introduction to Embedded Systems

- What is an embedded system?
  - Application-specific computer system
  - Interacting with its environment
  - Build into a larger system
  - Often with real-time computing constraints

- What is the motivation for building an embedded system?
  - Better performance
  - More functions and features
  - Lower cost e.g. through automation
  - More dependable
  - Lower power

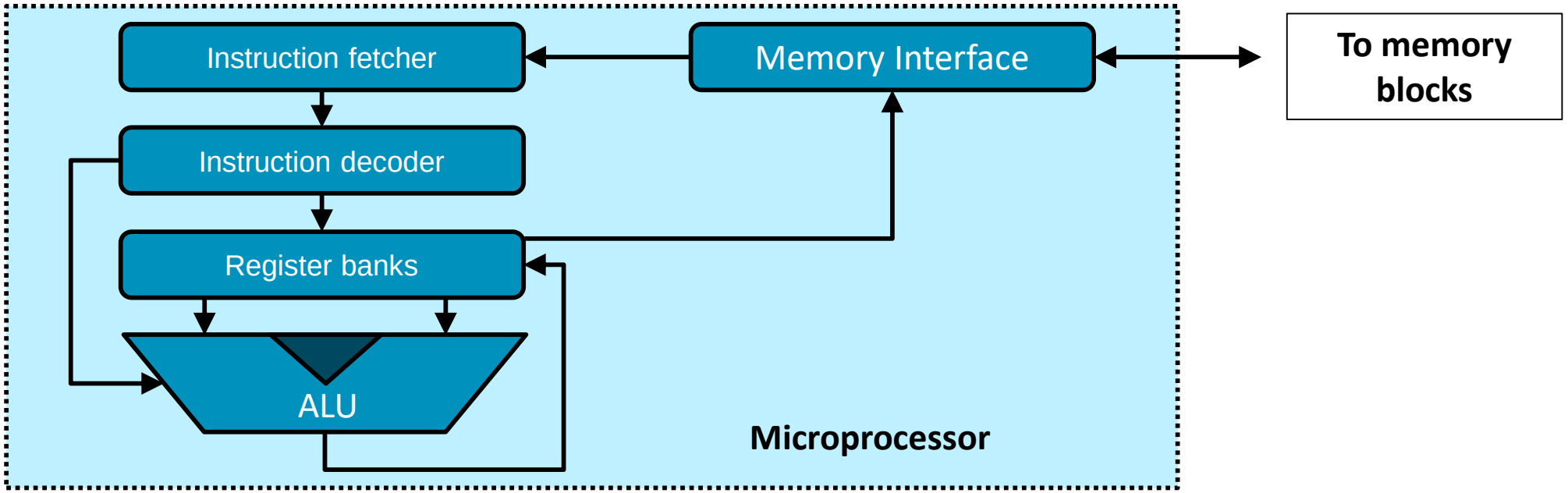
Macroscopic view on a device level



Microscopic view on a functionality level

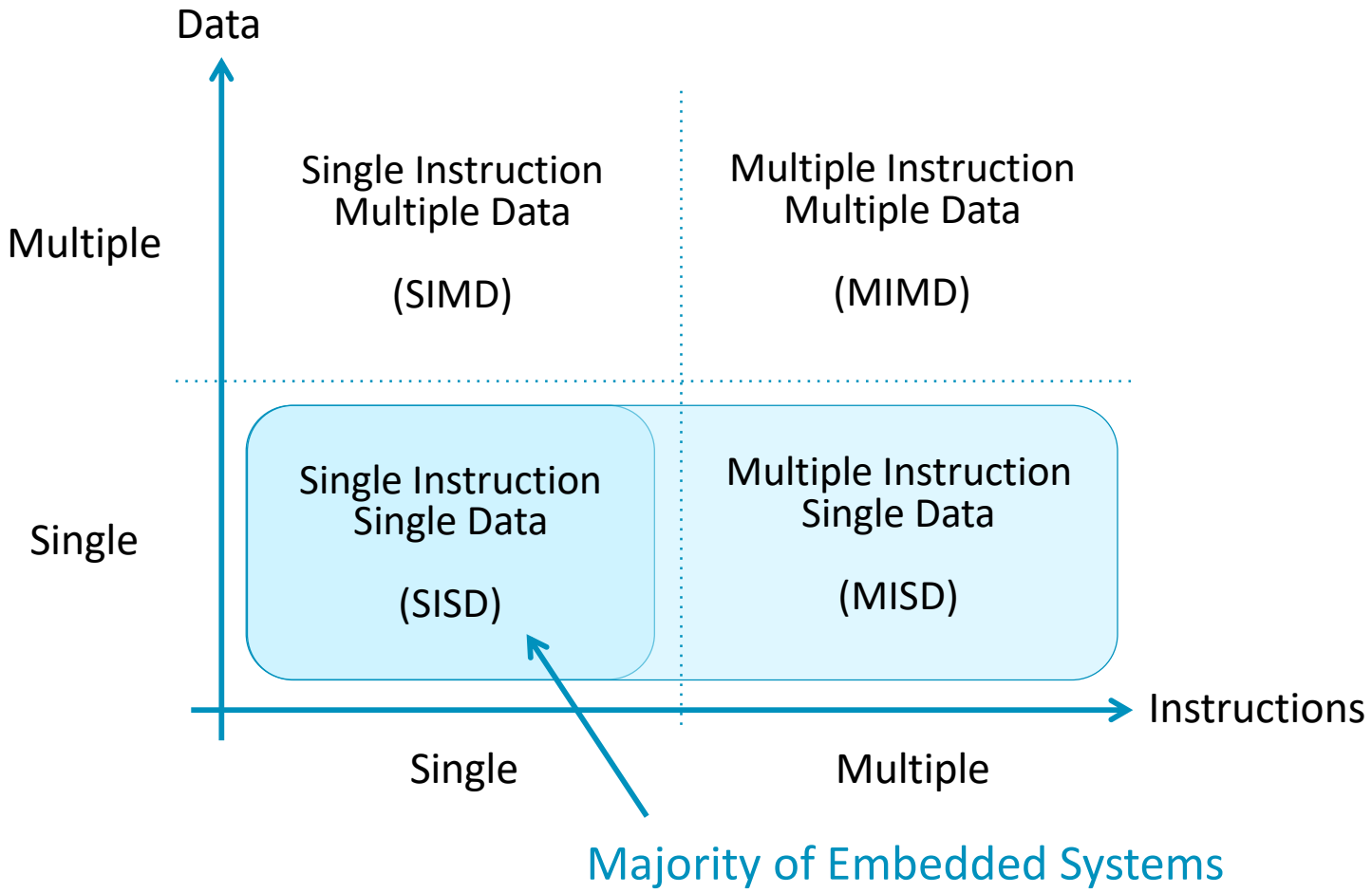
From a Processor to an Embedded System

- Microprocessor (central processing unit, CPU)
  - Typically defined as a single processor core that supports at least instruction fetching, decoding, and executing
  - Normally can be used for general-purpose computing, but needs to be supported with memory and Input/Outputs (IOs)



arm

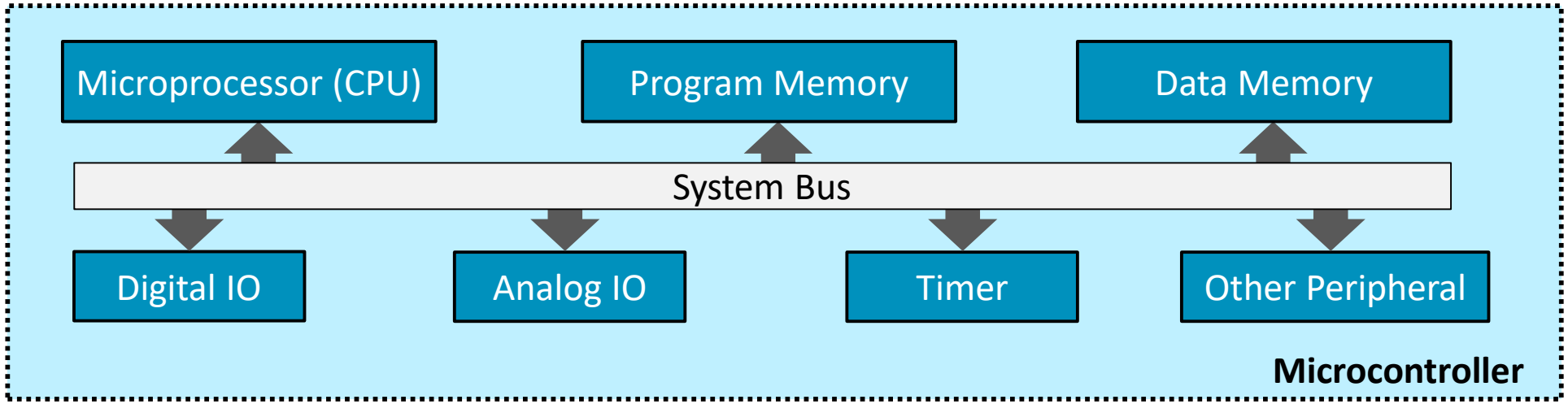
Flynn’s Taxonomy



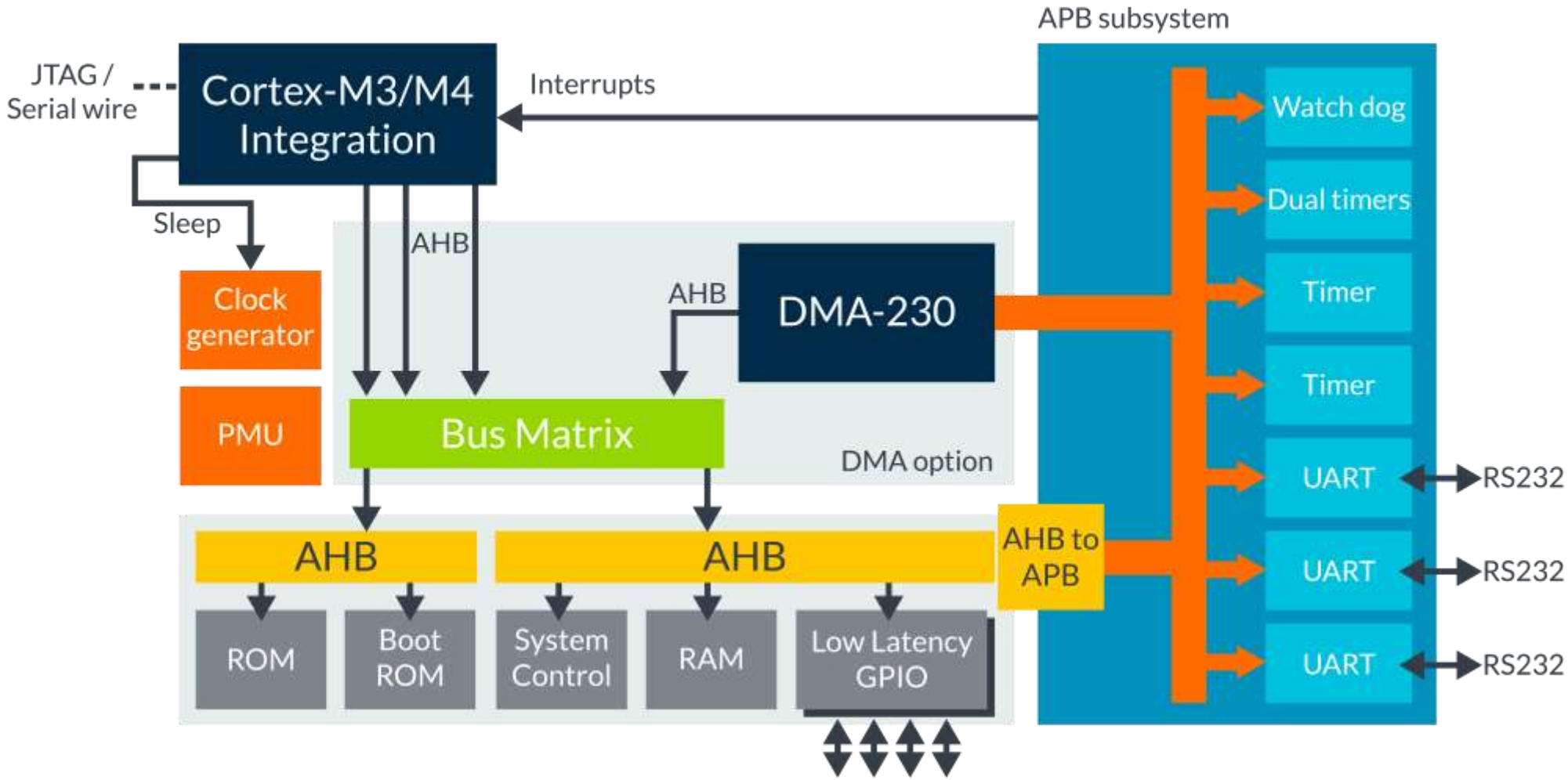
arm

From a Processor to an Embedded System

- Microcontroller (microcontroller unit, MCU)
  - Typically has a single processor core
  - Has memory blocks, digital IOs, analog IOs, and other basic peripherals
  - Typically used for basic control purpose, such as embedded applications



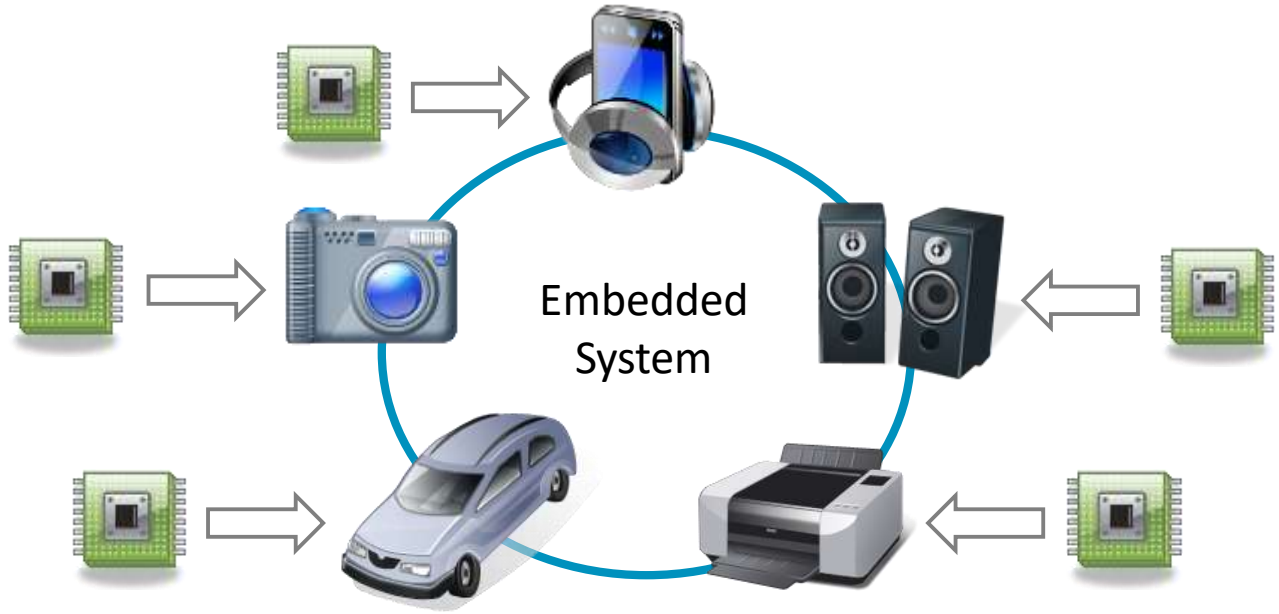
Example of an Arm M4-MCU Architecture



Source: https://developer.arm.com/ip-products/subsystem/corstone-foundation-ip/cortex-m-system-design-kit

From a Processor to an Embedded System

- Embedded system
  - Typically implemented using MCUs
  - Often integrated into a larger mechanical or electrical system
  - May have real-time constraints



arm

Attributes of Embedded Systems

- Interfacing with larger systems and environments
  - Analog signals for reading sensors
    - Typically use a voltage to represent a physical value
  - Power electronics for driving motors, solenoids
  - Digital interfaces for communicating with other digital devices
    - Simple – switches
    - Complex – displays
- Concurrent and reactive behaviours
  - Must respond to sequences and combinations of events
  - Real-time systems have deadlines on responses
  - Typically must perform multiple separate activities concurrently

Attributes of Embedded Systems

- Fault handling
  - Many systems must operate independently for long periods of time
    - Requires them to handle faults without crashing
  - Often, fault-handling code is larger and more complex than the normal-case code

- Diagnostics
  - Help service personnel determine problems quickly

Field Applications of Embedded Systems

- Closed-loop control system
  - Monitor and (pre)process a system state, adjust an output to maintain a desired set point (temperature, speed, direction, etc.)
  - Remove noise, select desired signal features
- Sequencing
  - Step through different stages based on environment and system
- Communications and networking
  - Exchange information reliably and quickly
- Part of a larger system
  - Taking over very specialized functions as part of a larger system, e.g. fault handling, handling networking

Example Embedded System: Bike Computer

Functions:
- Speed measurement
- Distance measurement

Constraints:
- Size
- Cost
- Power and energy
- Weight

Inputs:
- Wheel rotation indicator
- Mode key

Output:
- Liquid crystal display

Use low-performance microcontroller:
- 9-bit, 10 MIPS

arm

Example: Gasoline Automobile Engine Control Unit

Functions:

- Fuel injection
- Air intake setting
- Spark timing
- Exhaust gas circulation
- Electronic throttle control
- Knock control

Many inputs and outputs:

- Discrete sensors and actuators
- Network interface to rest of car

Constraints:

- Reliability in harsh environment
- Cost
- Size

Use high-performance microcontroller:

- E.g. 32-bit, 3 MB flash memory, 50-300 MHz

arm

Introduction to the Internet of Things (IoT)

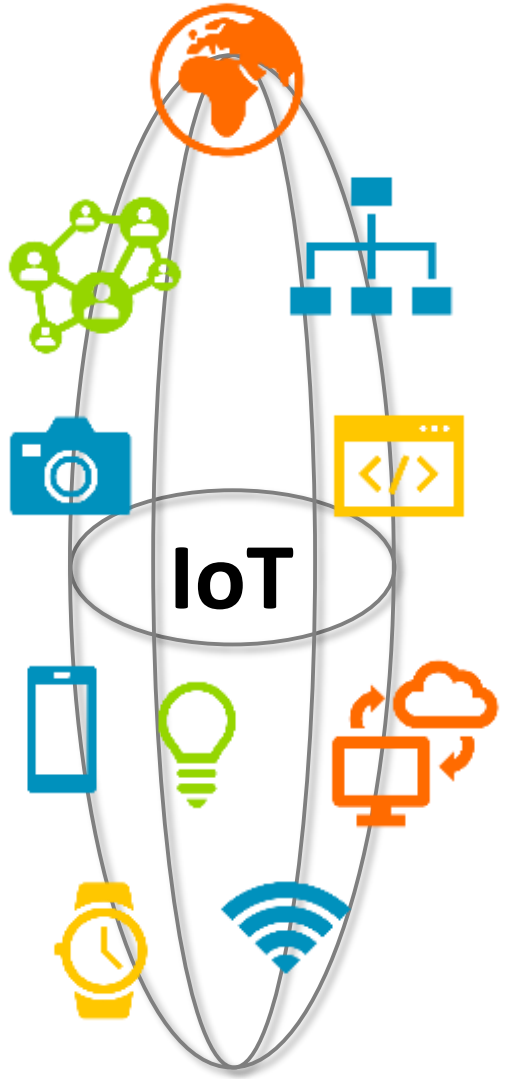
- IoT generally refers to a world in which a large range of objects are addressable via a network

Why IoT?

- Items can have more functionality and become more intelligent
- Items can be managed more easily
- More information becomes available

Objects can include:

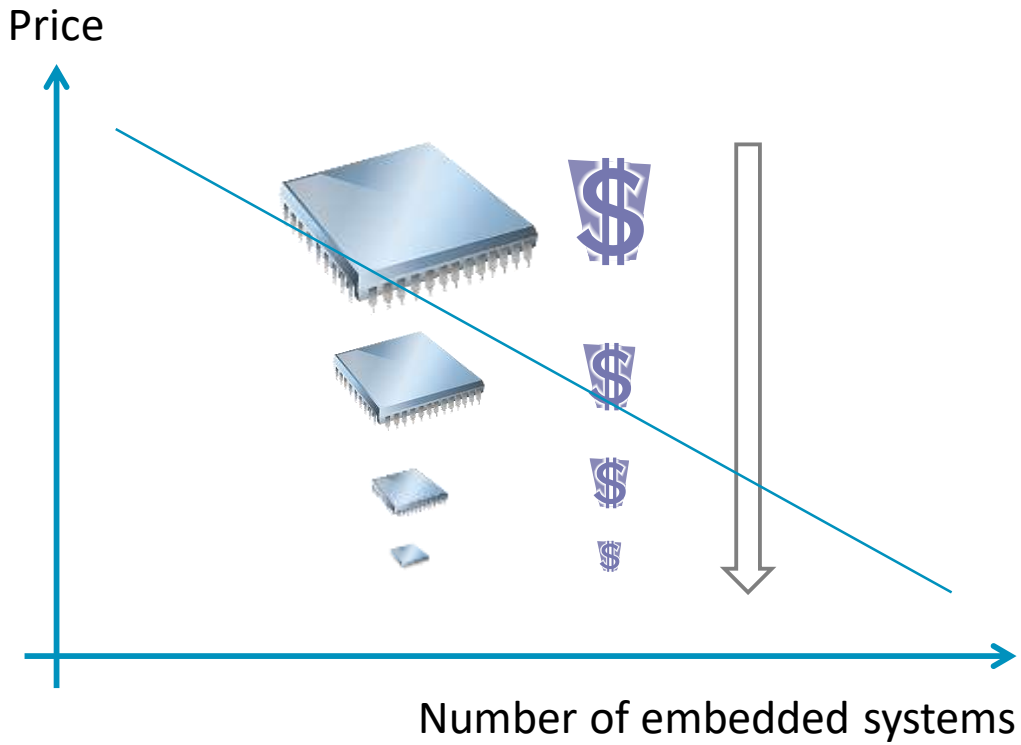
- Smart buildings and home appliances
  - Fridges, TVs, cookers
- Civil engineering structures
  - Bridges, railways
- Wearable devices
  - Smart watches, glasses
- Medical devices
  - Smart inhaler, embedded pills



arm

Internet of Things: Why Now?

- Embedded chips are becoming:
  - Cheaper
  - Smaller
  - Lower power
- Energy harvesting
- Communication is becoming faster and more efficient



arm

Challenges to the Internet of Things

- Large number of chips required
  - Chips have become even cheaper and smaller
- Big data demand
  - Large volume of data will be generated, data centre storage needs to be increased
- Computation requirement
  - High performance e.g. for cloud computing
- Power consumption
  - Low power chips, longer battery life
- Security
  - Large amount of private data needs to be protected
- Standards
  - Official standards are required, such as network protocols

Options for Building Embedded Systems

	Implementation	Design Cost	Unit Cost	Upgrades & Bug Fixes	Size	Weight	Power	System Speed
Dedicated Hardware	Discrete logic	low	mid	hard	large	high	?	very fast
	ASIC	high ($500K/ mask set)	very low	hard	tiny – 1 die	very low	low	extremely fast
	Programmable logic – FPGA, PLD	low to mid	mid	easy	small	low	medium to high	very fast
Software Running on Generic Hardware	Microprocessor + memory + peripherals	low to mid	mid	easy	small to medium	low to moderate	medium	moderate
	Microcontroller (int. memory & peripherals)	low	mid to low	easy	small	low	medium	slow to moderate
	Embedded PC	low	high	easy	medium	moderate to high	medium to high	fast

Microcontroller based embedded system

arm

Benefits of Microcontroller-based Embedded Systems

- Greater performance and efficiency
  - Software makes it possible to provide sophisticated control
- Lower costs for mixed signal-processing systems
  - Less expensive components can be used
  - Manufacturing costs reduced
  - Operating costs reduced
  - Maintenance costs reduced
- More features
  - May not be possible or practical with other approaches
- Better dependability
  - Adaptive system which can compensate for failures
  - Better diagnostics to improve repair time

arm

Impact of Constraints

- Microcontrollers used (rather than microprocessors)
  - Include peripherals to interface with other devices, respond efficiently
  - On-chip RAM, ROM reduce circuit board complexity and cost
- Programming language
  - Programmed in the C language rather than Java (resulting in smaller and faster code – less expensive MCU)
  - Some performance-critical code may be in assembly language
- Operating system
  - Typically no OS, but instead a simple scheduler
  - If OS is used, it is likely to be a lean RTOS

Summary: Building Embedded Systems using MCUs

- In most embedded systems, MCUs are the best solution as they offer:
  - Low development and manufacturing cost
  - Easy porting and updating
  - Light footprint
  - Relatively low power consumption
  - Satisfactory performance for low-end products

In the upcoming labs, we will learn some fundamentals of developing for embedded systems. This will be achieved using an easy-to-start MCU design suite – the Mbed platform.

Coming Next

- Knowledge of embedded systems
  - Hardware mechanisms
    - Introducing the Arm Cortex-M architecture
    - Use interrupts for low power design
  - Software programming
    - Programming basics: C/C++ programming
    - Introducing the Mbed platform
    - Learn to use software libraries: Mbed API

arm

Coming Next

- Develop your own embedded systems
  - Analog IOs: ADC, DAC, PWM
  - Serial communications: UART, I2C, SPI
  - Advanced serial communication: USB, CAN, Bluetooth LE
  - Network: Ethernet, TCP/IP, HTTP
  - Real-time operating system (RTOS)
  - Prototyping applications for IoT

Resources

- Official Mbed website: https://www.mbed.com/en/

arm

The Arm Cortex-M4 Processor Architecture

Module Syllabus

- Arm architectures and processors
  - What is Arm architecture?
  - Arm processor families
  - Arm Cortex-M series
  - Cortex-M4 processor
  - Arm processors vs. Arm architectures
- Arm Cortex-M4 Processor
  - Processor overview
  - Block diagram
  - Registers

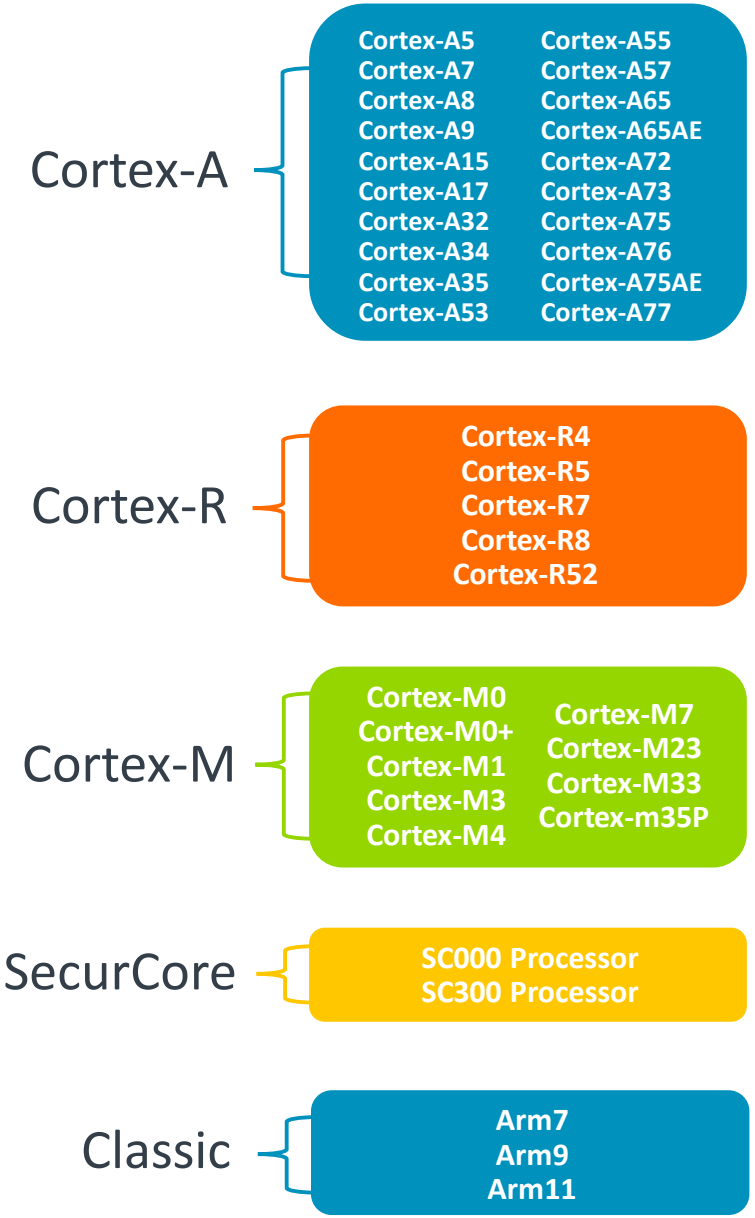
Arm Architectures and Processors

- Arm architecture is a family of RISC-based processor architectures
  - Well known for its power efficiency
  - Widely used in mobile devices, e.g. smartphones and tablets
  - Designed and licensed by Arm to a wide ecosystem of partners

- Arm
  - The company that designs Arm-based processors
  - Arm does not manufacture, but it licenses designs to semiconductor partners who add their own intellectual property (IP) on top of Arm's OP, which they then fabricate and sell to customers
  - Arm also offers IP other than processors, such as physical IPs, interconnect IPs, graphics cores and development tools

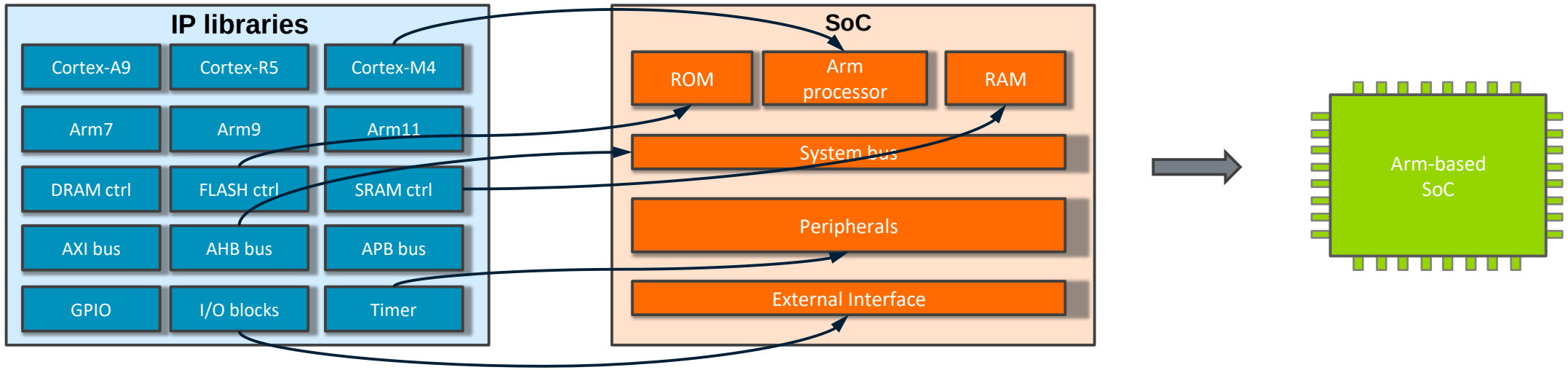
Arm Processor Families

- Cortex-A series (Application)
  - High performance processors capable of full operating system (OS) support
  - Applications include smartphones, digital TV, smart books
- Cortex-R series (Real-time)
  - High performance and reliability for real-time applications
  - Applications include automotive braking systems, powertrains
- Cortex-M series (Microcontroller)
  - Cost sensitive solutions for deterministic microcontroller applications
  - Applications include microcontrollers, smart sensors
  - SecurCore series for high security applications
- Earlier classic processors including Arm7, Arm9, Arm11 families



How to Design an Arm-based SoC

1. Select a set of IP cores from Arm and/or other third-party IP vendors
2. Integrate IP cores into a single-chip design
3. Give design to semiconductor foundries for chip fabrication



Licensable IPs

SoC Design

Chip Manufacture

arm

Arm Cortex-M Series

Cortex-M

Cortex-M0
Cortex-M0+
Cortex-M1
Cortex-M3
Cortex-M4
Cortex-M7
Cortex-M23
Cortex-M33
Cortex-m35P

- Energy-efficiency
  - Low energy cost, long battery life
- Smaller code
  - Lower silicon costs
- Ease of use
  - Faster software development and reuse

- Embedded applications
  - Smart metering, human interface devices, automotive and industrial control systems, white goods, consumer products and medical instrumentation

arm

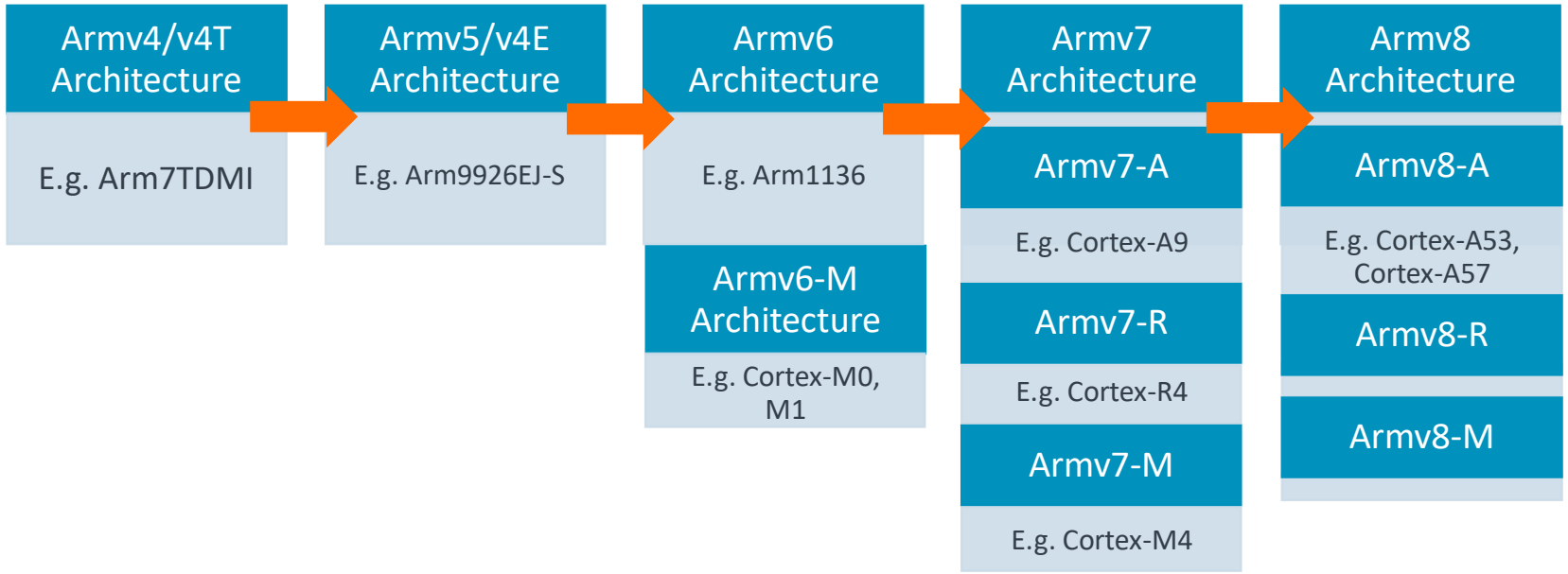
Arm Processors vs. Arm Architectures

Arm architecture

- Describes the details of instruction set, programmer's model, exception model, and memory map
- Documented in the 'Architecture Reference Manual'

Arm processor

- Developed using one of the Arm architectures
- More implementation details, such as timing information
- Documented in processor's 'Technical Reference Manual'



Arm Cortex-M Series Family

Processor	Arm Architecture	Core Architecture	Thumb	Thumb-2	Hardware Multiply	Hardware Divide	Saturated Math	DSP Extensions	Floating Point
Cortex-M0	Armv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M0+	Armv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M3	Armv7-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	No	No
Cortex-M4	Armv7E-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	Yes	Optional
Cortex-M7	Armv7E-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	Yes	Optional

arm

Cortex-M4 Processor Overview

- Cortex-M4 processor
  - Introduced in 2010
  - Designed with a large variety of highly efficient signal processing features
  - Features extended single-cycle multiply accumulate instructions, optimized SIMD arithmetic, saturating arithmetic, and an optional floating-point unit
- High performance efficiency
  - 1.25 DMIPS/MHz (Dhrystone million instructions per second/MHz) at the order of µWatts/MHz
- Low power consumption
  - Longer battery life – especially critical in mobile products
- Enhanced determinism
  - The critical tasks and interrupt routines can be served quickly in a known number of cycles

Cortex-M4 Processor Features

- 32-Bit reduced instruction set computing (RISC) processor
- Harvard architecture
  - Separated data bus and instruction bus
- Instruction set
  - Includes the entire Thumb-1 (16-bit) and Thumb-2 (16/32-bit) instruction sets
- 3-stage + branch speculation pipeline
- Supported interrupts
  - Non-maskable interrupt (NMI) + 1 to 240 physical interrupts
  - 8 to 256 interrupt priority levels

Cortex-M4 Processor Features

- Supports sleep modes
  - Up to 240 wake-up interrupts
  - Integrated wait for interrupt (WFI) and wait for event (WFE) instructions and sleep on exit capability
  - Sleep and deep sleep signals
  - Optional retention mode with arm power management kit
- Enhanced instructions
  - Hardware divide (2-12 cycles)
  - Single-cycle 16, 32-bit MAC, single-cycle dual 16-bit MAC
  - 8, 16-bit SIMD arithmetic

arm

Cortex-M4 Processor Features

- Debug
  - Optional JTAG & serial-wire debug (SWD) ports
  - Up to eight breakpoints and four watchpoints
- Memory protection unit (MPU)
  - Optional eight-region MPU with sub regions and background regions

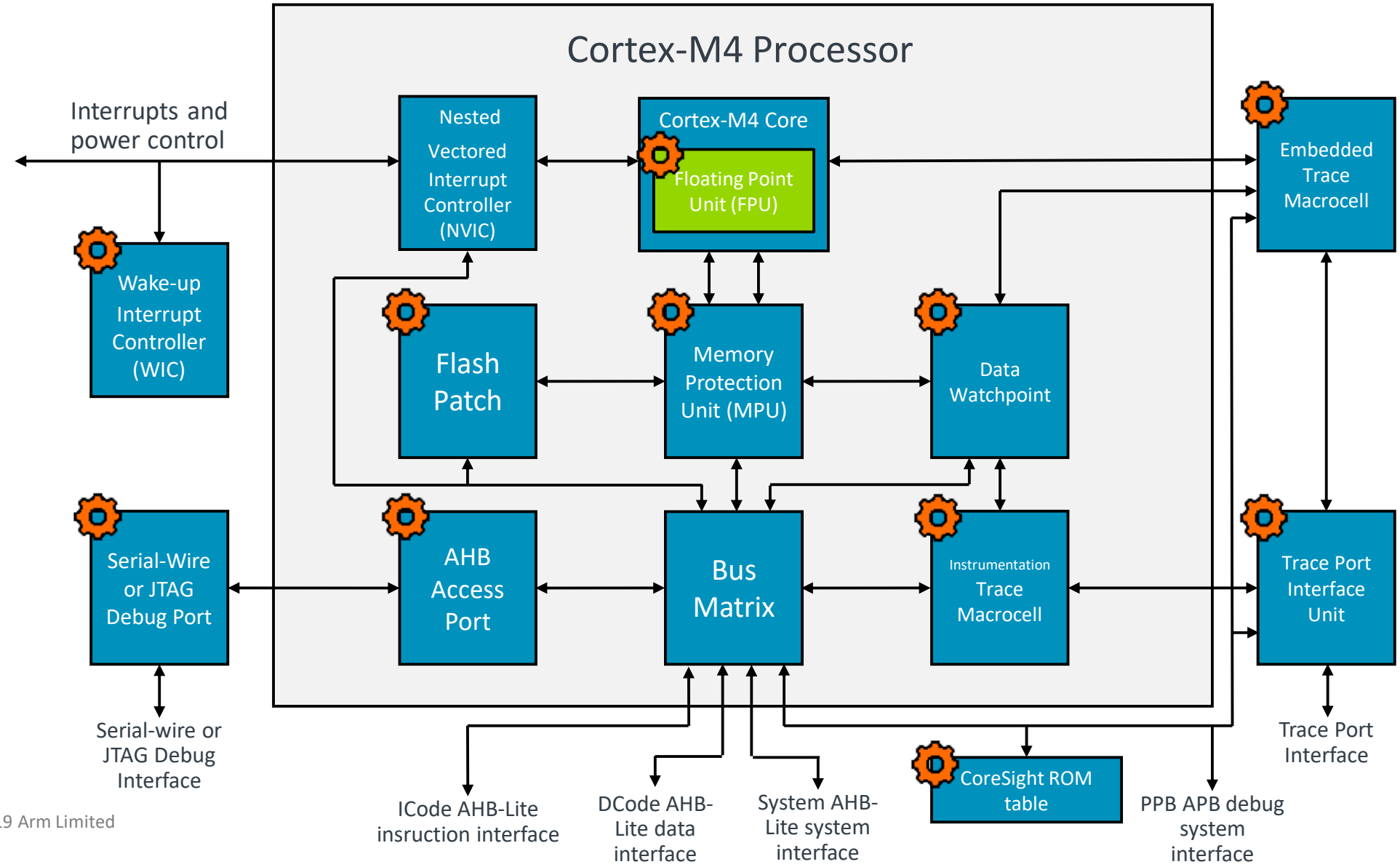
Cortex-M4 Processor Features

- The Cortex-M4 processor is designed to meet the challenges of low dynamic power constraints while retaining a light footprint
  - 180ULL ultra low power process: 151 µW/MHz
  - 90LP low power process: 32.82 µW/MHz
  - 40LP low power process: 12.26 µW/MHz

Arm Cortex-M4 Implementation Data			
Process	180ULL (7-track, typical 1.8v, 25C)	90LP (7-track, typical 1.2v, 25C)	40G (9-track, typical 0.9v, 25C)
Dynamic power	151 µW/MHz	32.82 µW/MHz	12.26 µW/MHz
Floor planned area	0.44 mm^2	0.119 mm^2	0.028 mm^2

Cortex-M4 Block Diagram

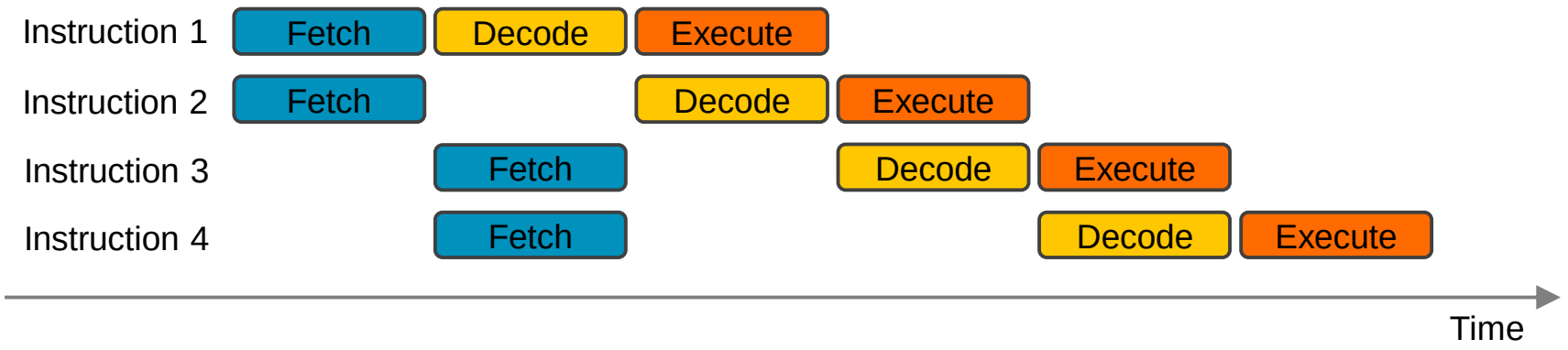
Optional component



Cortex-M4 Block Diagram

- Processor core
  - Contains internal registers, the ALU, data path, and some control logic
  - Registers include 16x 32-bit registers for both general and special use

- Processor pipeline stages
  - Three-stage pipeline: fetch, decode, and execution
  - Some instructions may take multiple cycles to execute, in which case the pipeline will be stalled
  - Speculatively prefetches instructions from branch target addresses
  - Up to two instructions can be fetched in one transfer (16-bit instructions)



Cortex-M4 Block Diagram

- Nested vectored interrupt controller (NVIC)
  - Up to 240 interrupt request signals and an NMI
  - Automatically handles nested interrupts, such as comparing priorities between interrupt requests and the current priority level
- Wake-up interrupt controller (WIC)
  - For low-power applications, the microcontroller can enter sleep mode by shutting down most of the components
  - When an interrupt request is detected, the WIC can inform the power management unit to power up the system
- Memory protection unit (MPU)
  - Used to protect memory content, e.g., make some memory regions read-only or preventing user applications from accessing privileged application data

Cortex-M4 Block Diagram

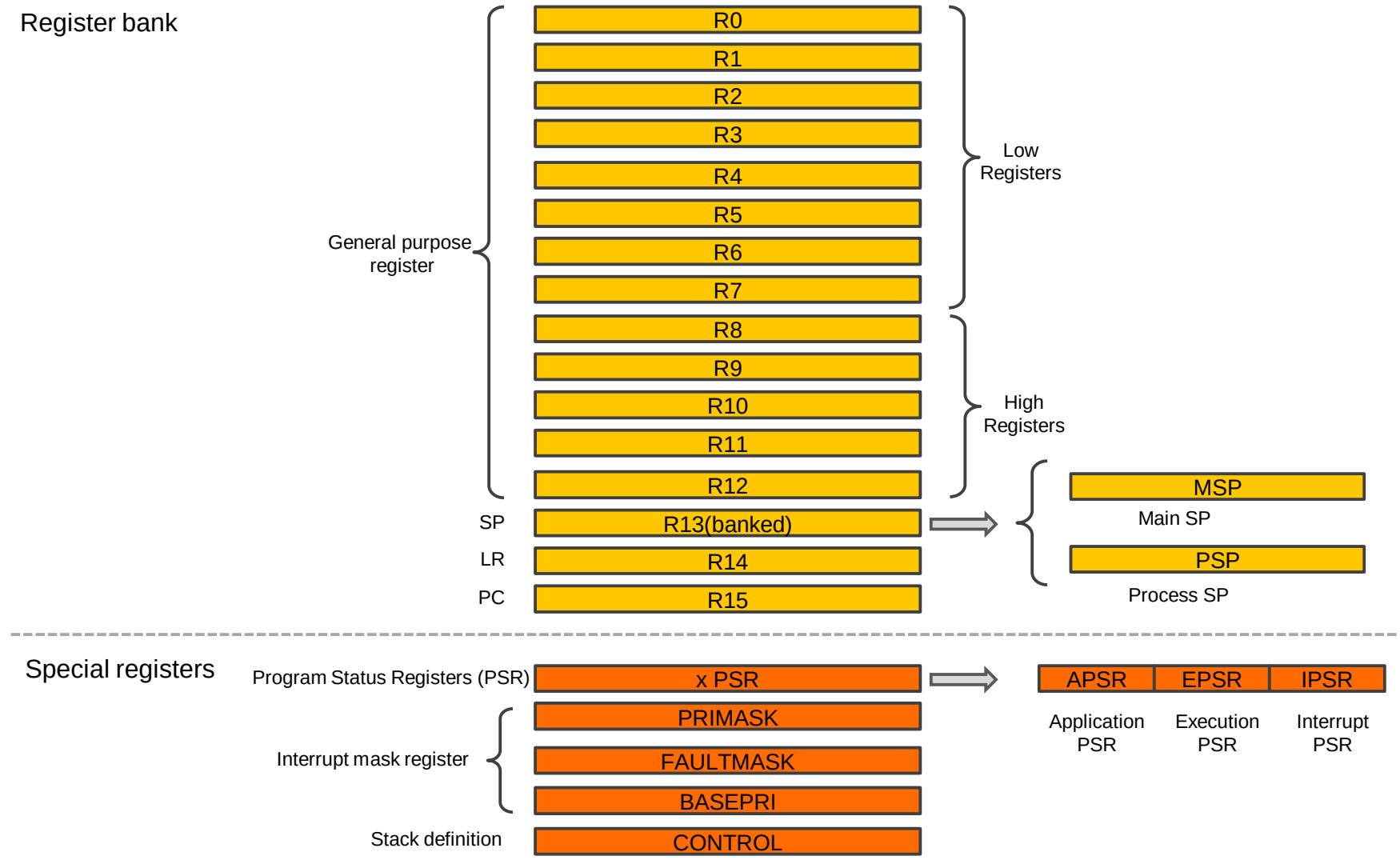
- Bus interconnect
  - Allows data transfer to take place on different buses simultaneously
  - Provides data transfer management, e.g. write buffer, bit-oriented operations (bit-band)
  - May include bus bridges (e.g. AHB-to-APB bus bridge) to connect different buses into a network using a single global memory space
  - Includes the internal bus system, the data path in the processor core, and the AHB LITE interface unit

- Debug subsystem
  - Handles debug control, program breakpoints, and data watchpoints
  - When a debug event occurs, it can put the processor core in a halted state, so developers can analyse the status of the processor, such as register values and flags, at that point

Arm Cortex-M4 Processor Registers

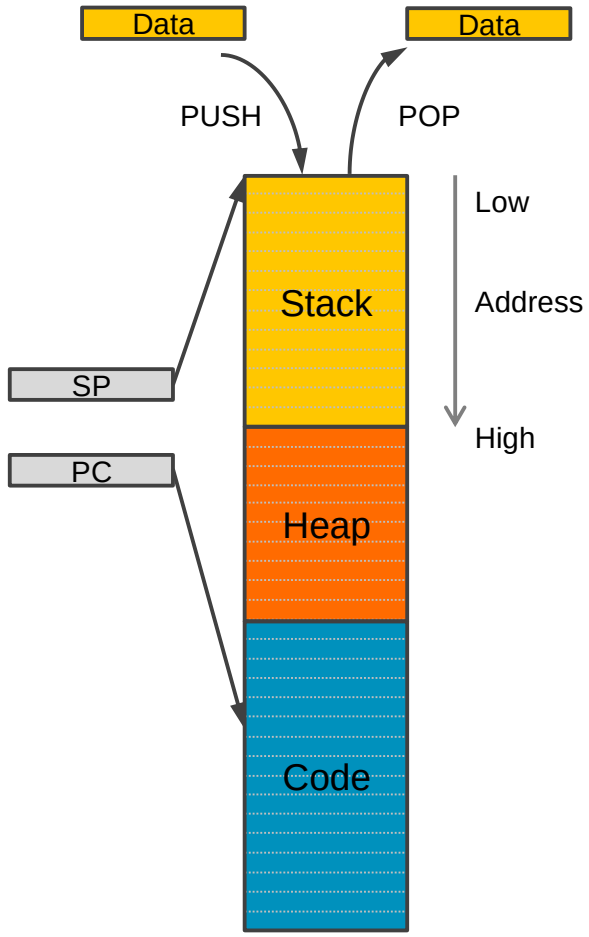
- Processor registers
  - The internal registers are used to store and process temporary data within the processor core
  - All registers are inside the processor core, so they can be accessed quickly
  - Load-store architecture
    - To process memory data, they have to first be loaded from memory to registers, processed inside the processor core using register data only, and then written back to memory if needed
  - Cortex-M4 registers
    - Register bank
      - 16x 32-bit registers (thirteen are used for general-purpose)
    - Special registers

Cortex-M4 Registers



Cortex-M4 Registers

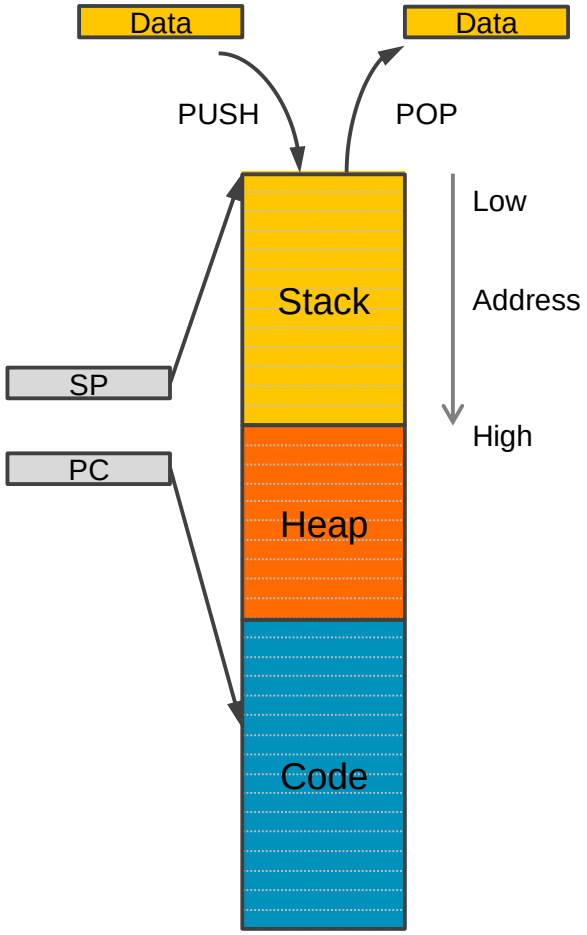
- R0-R12: general purpose registers
  - Low registers (R0-R7) can be accessed by any instruction
  - High registers (R8-R12) sometimes cannot be accessed e.g. by some Thumb (16-bit) instructions
- R13: Stack Pointer (SP)
  - Records the current address of the stack
  - Used for saving the context of a program while switching between tasks
  - Cortex-M4 has two SPs: Main SP, used in applications that require privileged access e.g. OS kernel and process SP, used in base-level application code (when not running an exception handler)



arm

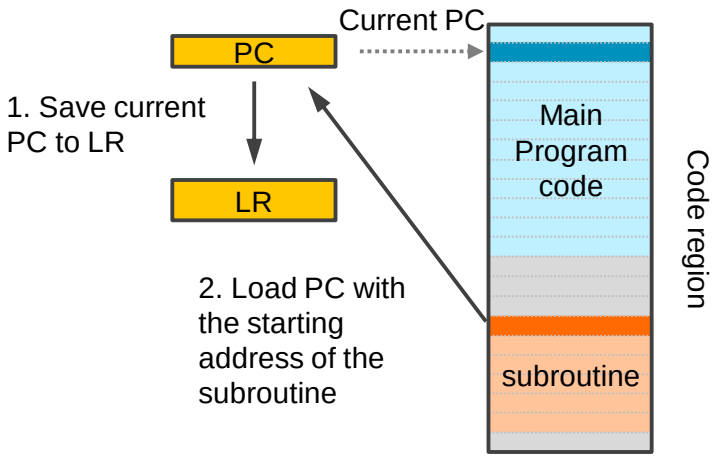
Cortex-M4 Registers

- Program Counter (PC)
  - Records the address of the current instruction code
  - Automatically incremented by four at each operation (for 32-bit instruction code), except branching operations
  - A branching operation, such as function calls, will change the PC to a specific address, while saving the current PC to the LR

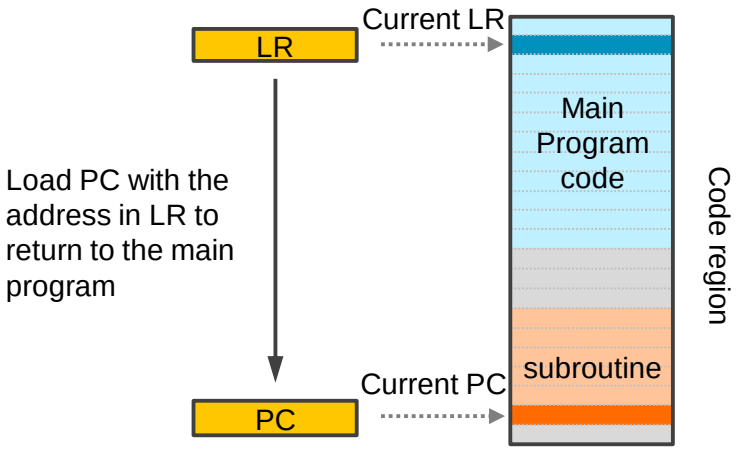


Cortex-M4 Registers

- R14: Link Register (LR)
  - The LR is used to store the return address of a subroutine or a function call
  - The PC will load the value from the LR after a function is finished



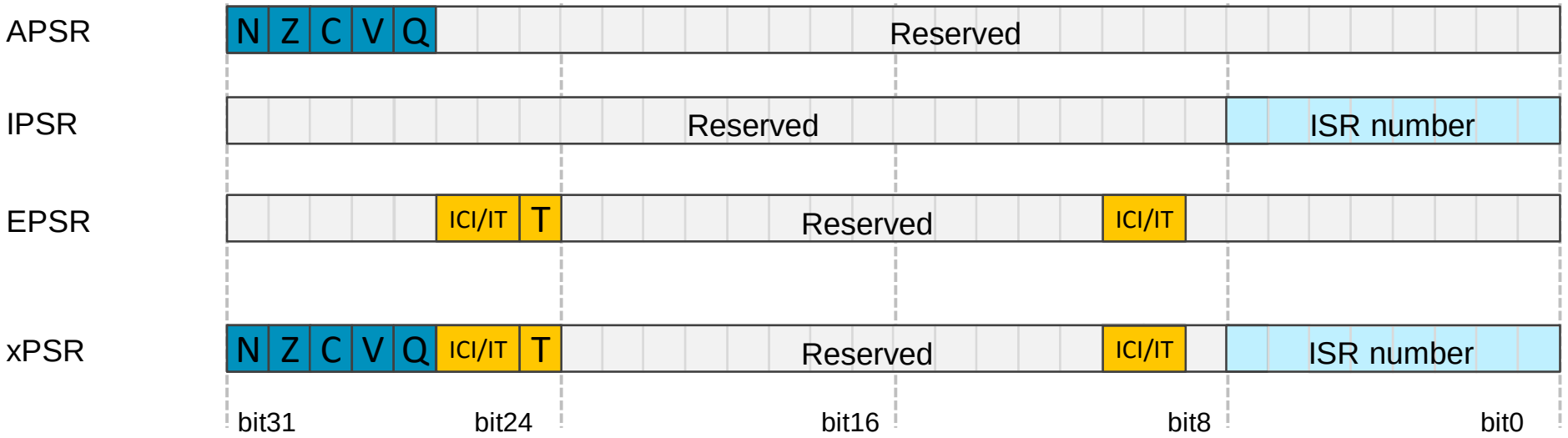
Call a subroutine



Return from subroutine to main program

Cortex-M4 Registers

- xPSR, combined program status register (PSR)
  - Provides information about program execution and ALU flags
  - Application PSR (APSR)
  - Interrupt PSR (IPSR)
  - Execution PSR (EPSR)



Cortex-M4 Registers

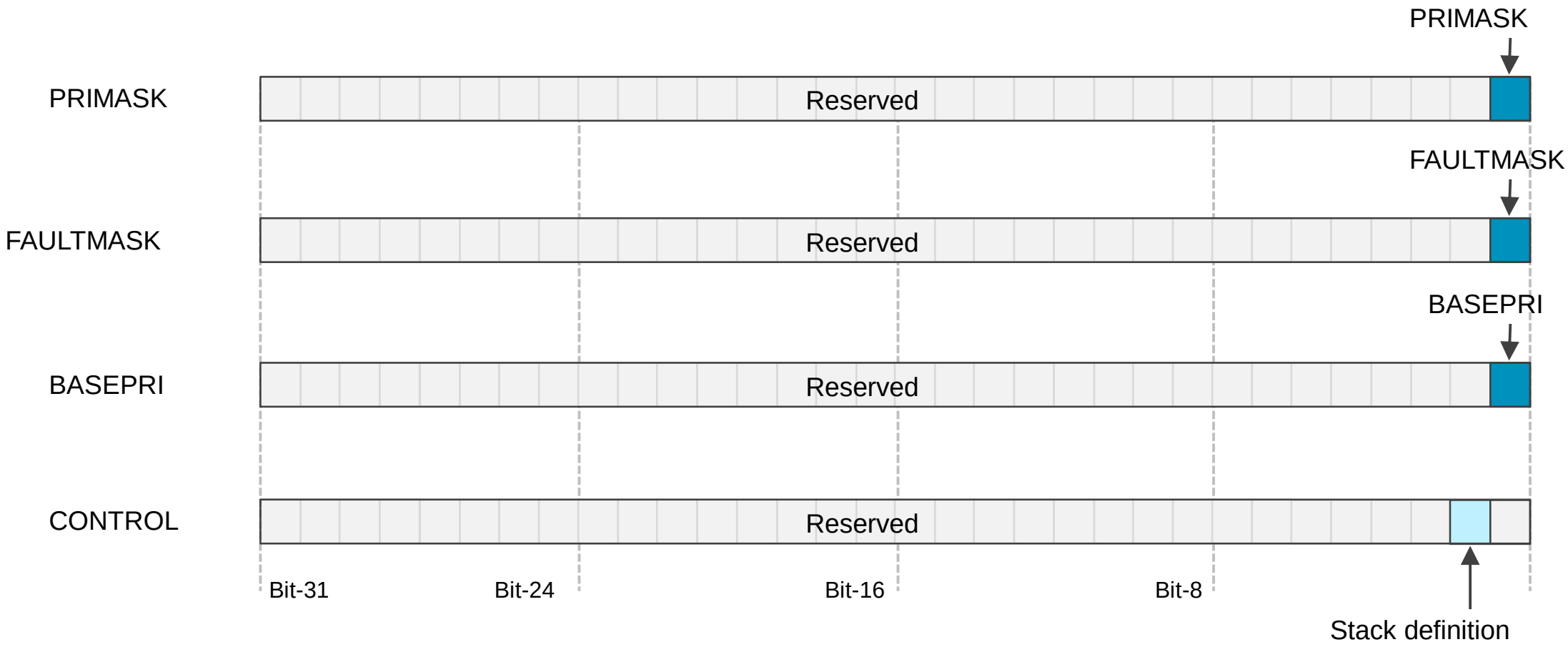
- APSR
  - N: negative flag: set to one if the result from the ALU is negative
  - Z: zero flag: set to one if the result from the ALU is zero
  - C: carry flag: set to one if an unsigned overflow occurs
  - V: overflow flag: set to one if a signed overflow occurs
  - Q: stick saturation flag: set to one if saturation has occurred in saturating arithmetic instructions, or overflow has occurred in certain multiply instructions
- IPSR
  - Interrupt service routine (ISR) number: current executing ISR number
- EPSR
  - T: Thumb state: always one since Cortex-M4 only supports the Thumb state
  - IC/IT: Interrupt-continuable instruction (ICI) bit, IF-THEN instruction status bit

Cortex-M4 Registers

- Exception mask registers
  - 1-bit PRIMASK
    - If set to one, will block all interrupts apart from NMI and the hard fault exception
  - 1-bit FAULTMASK
    - If set to one, will block all the interrupts apart from NMI
  - 1-bit BASEPRI
    - If set to one, will block all interrupts of the same or lower level (only allowing for interrupts with higher priorities)

- CONTROL: special register
  - 1-bit stack definition
    - Set to one to use the PSP
    - Clear to zero to use the MSP

Cortex-M4 Registers



Resources

- Cortex-M4 Technical Reference Manual: https://developer.arm.com/docs/100166/0001

- Cortex-M4 Devices Generic User Guide: https://developer.arm.com/docs/dui0553/b

arm

Introduction to Cortex-M4 Programming

Module Syllabus

- Program code
  - C language vs. assembly language
  - Program-generation flow
  - Cortex-M4 program image
- Program data
  - How is data stored in RAM?
  - Data types
  - Accessing data using C and assembly
- Mixed assembly and C programming
  - Calling a C function from assembly
  - Calling an assembly function from C
  - Embedded assembly

Program Code Overview

- All microcontrollers need a program code to perform an intended task.

- The program of embedded systems is usually developed at a relatively low level.
  - Less support from operating systems, as operating systems are partly unwanted overhead
  - Lack of standard programing interface
  - It is often hardware specific, depending on memory usage constraints, available instructions, etc.

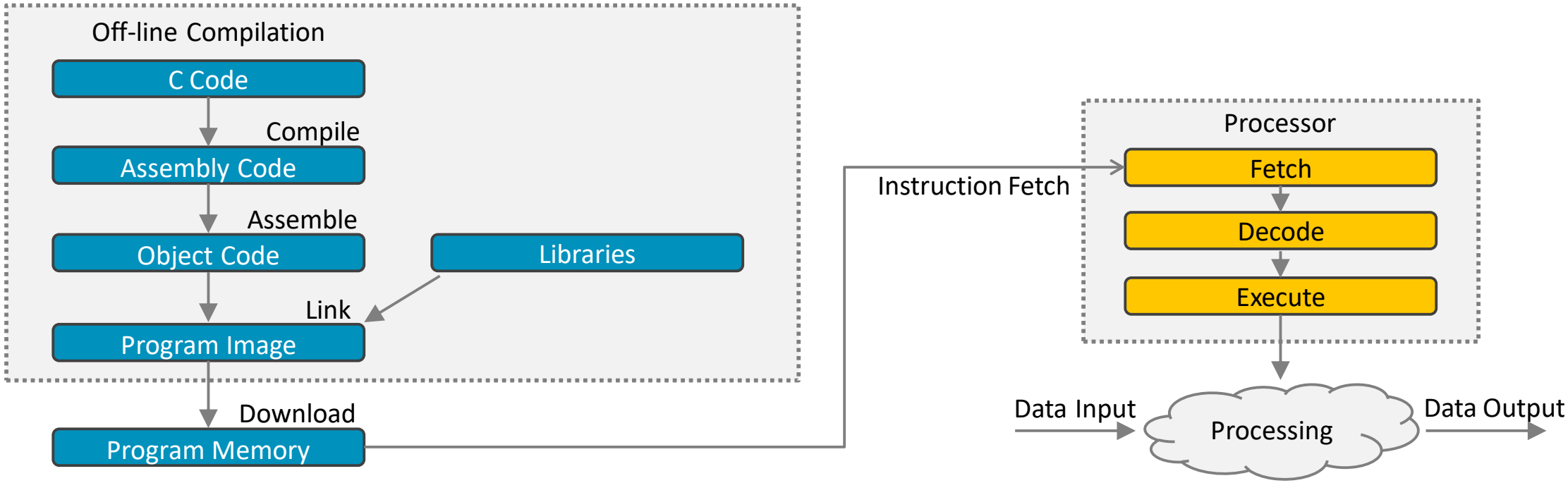
- Programming language
  - Embedded systems are usually programed using C (or C++) language and assembly language.
  - Arm-based tools can compile the C code or the assembly code to the executable file, which can be executed by Arm-based processors.
  - A completely integrated program code can also be referred as a program image or an executable file.

C Language vs. Assembly Language

Language	Advantages	Disadvantages
C	Easy to learn	Limited or no direct access to core registers and stack
	Portable	No direct control over instruction sequence generation
	Easy handling of complex data structures	No direct control over stack usage
Assembly	Allows direct control to each instruction step and all memory	Takes a longer time to learn
	Allows direct access to instructions that cannot be generated with C	Difficult to manage data structure
		Less portable

Typical Program-generation Flow

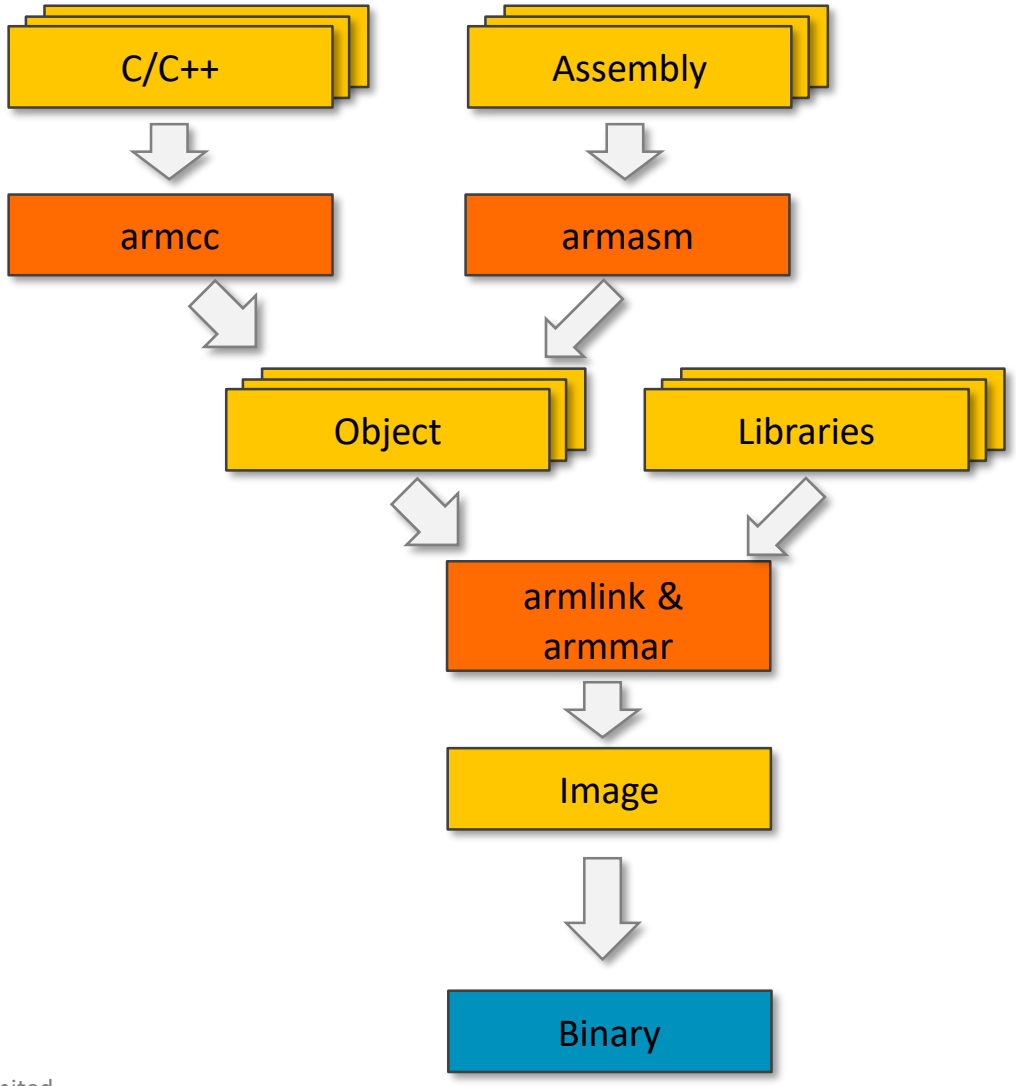
- The generation of program follows a typical development flow:
  - Compile -> Assemble -> Link -> Download
  - The generated executable file (or program image) is stored in the program memory (normally an on-chip flash memory), to be fetched by the processor



Typical program-generation flow

arm

Compilation using Arm-based Tools



C, C++
ASM files

Compile/assemble

.O Files
.S Files

Link

.AXF File
.LIB file

Executable

.BIN File
.HEX File
Disassembly File

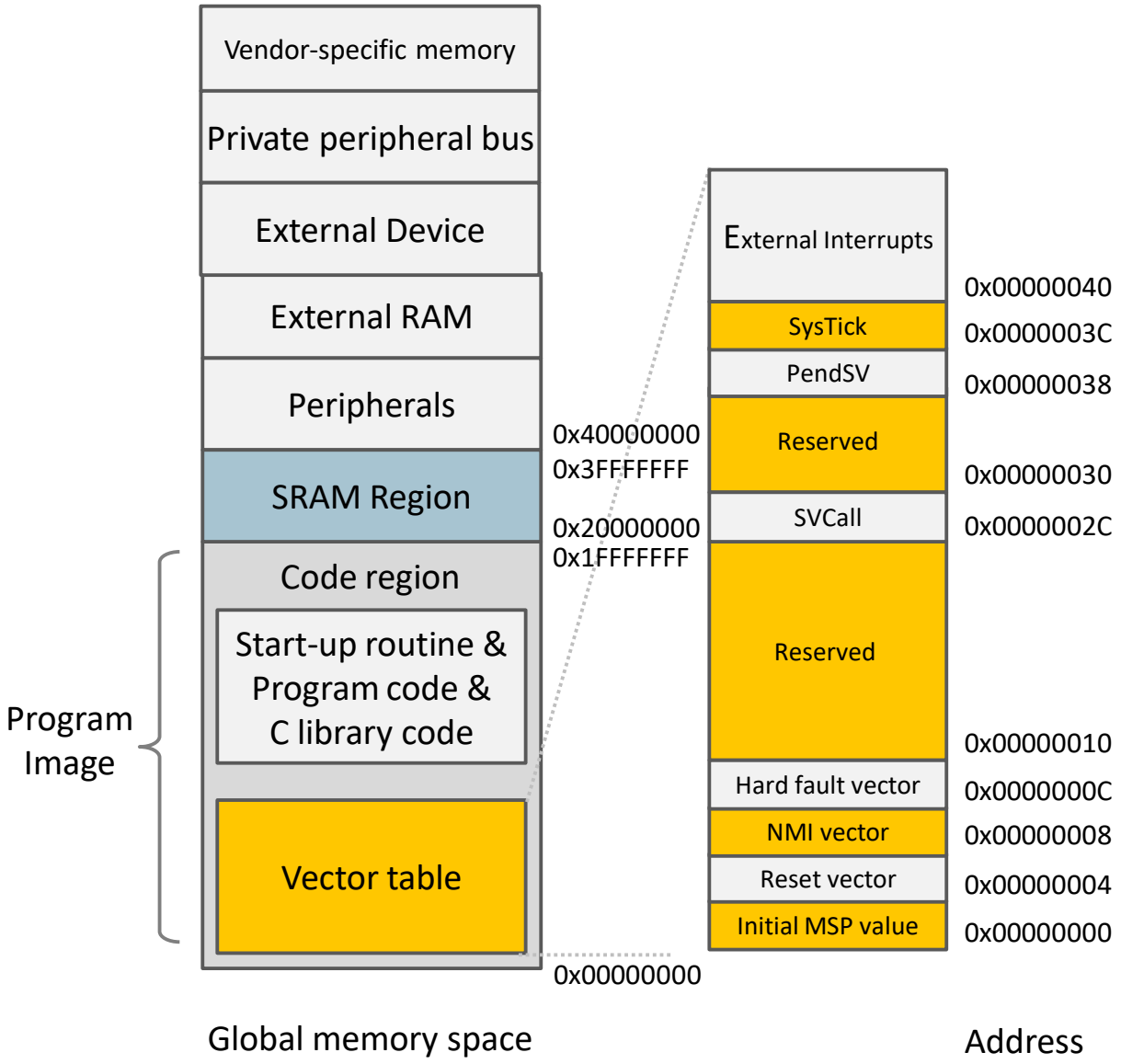
Compiler Stages

- Pre-processing
  - Replaces macros, defined by an initial hash-tag (#) in the code
  - Merges all subfiles (.c, .h) to one complete file
- Parser
  - Reads in C code
  - Checks for syntax errors
  - Forms intermediate code (tree representation)
- High-Level Optimizer: Modifies intermediate code (processor-independent)
- Code Generator
  - Creates assembly code step-by-step from each node of the intermediate code
  - Allocates variable uses to registers
- Low-Level Optimizer: Modifies assembly code (parts are processor-specific)
- Assembler: Creates object code (machine code)
- Linker/Loader: Creates executable image from object file

Cortex-M4 Program Image

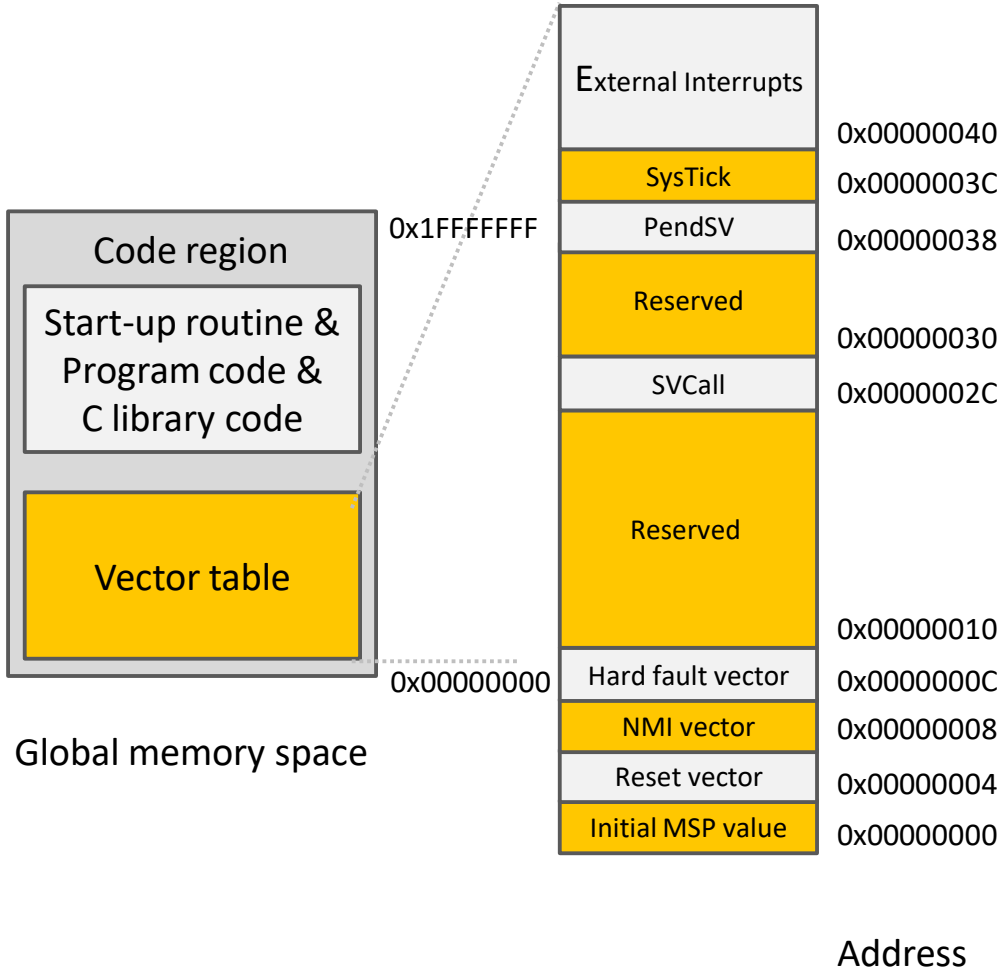
- What is a program image?
  - The program image (sometimes also called the executable file) refers to a piece of fully integrated code that is ready to execute.

- In the Cortex-M4, the program image includes:
  - Vector table: includes the starting addresses of exceptions (vectors) and the value of the main stack point (MSP)
  - C start-up routine
  - Program code: application code and data
  - C library code: program codes for C library functions



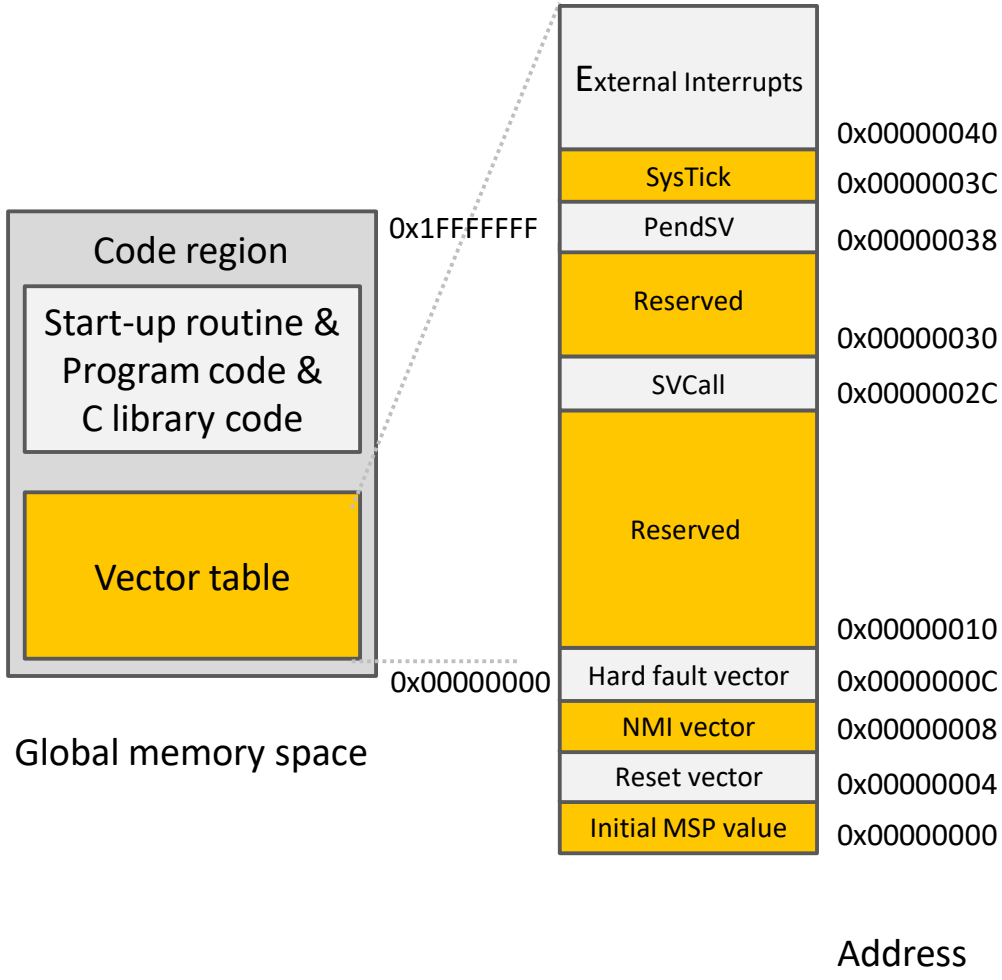
Cortex-M4 Program Image

- Vector table
  - Contains the starting addresses of exceptions (vectors) and the value of the main stack point (MSP)
- C Start-up code
  - Used to set up data memory and the initialization of values for global data variables
  - Is inserted by the compiler/linker automatically, labeled as '__main' by the Arm compiler, or '__start' by the GNU C compiler



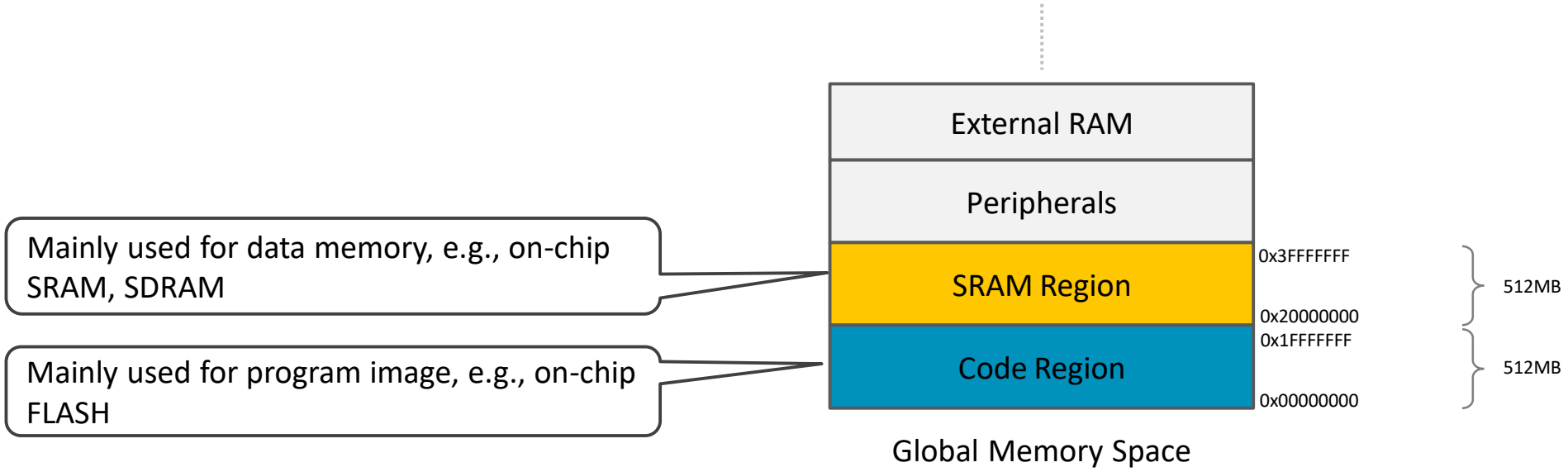
Cortex-M4 Program Image

- Program code
  - Program code refers to the instructions generated from the application program. Data types include:
    - Initial values of variables: the local variables that are initialized in functions or subroutines during program execution time
    - Constants: used in data values, address of peripherals, character strings, etc.
      - Sometimes stored together in data blocks called literal pools
      - Constant data such as lookup tables, graphics image data (e.g., bit map) can be merged into the program images
- C library code
  - Object codes inserted into the program image by linkers



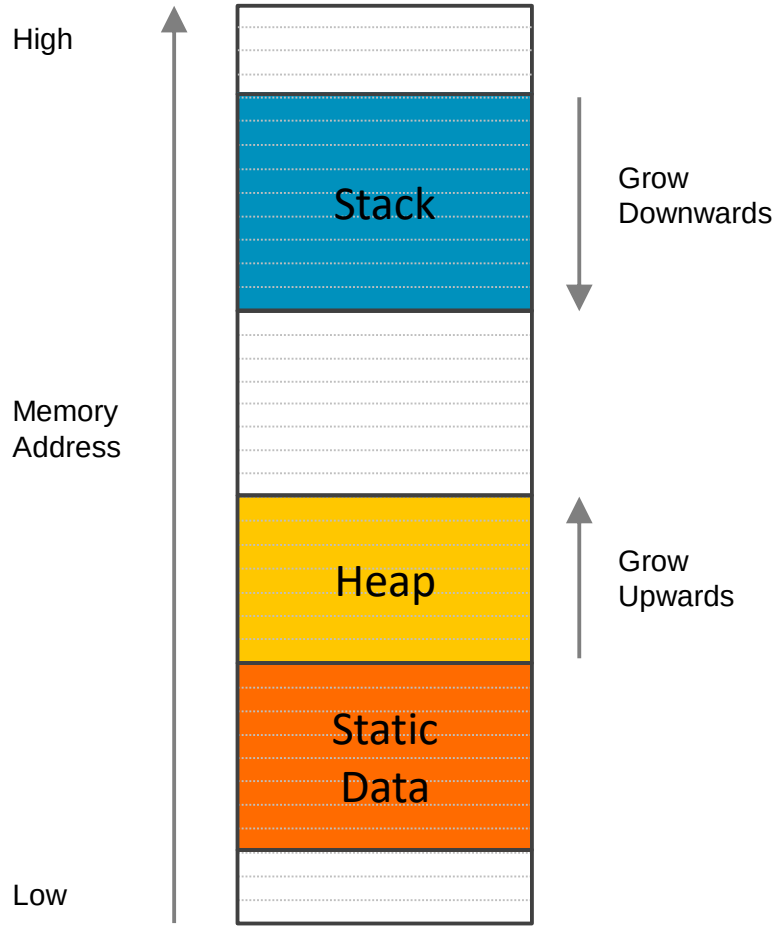
Program Image in Global Memory

- The program image is stored in the code region in global memory
  - Up to 512 MB memory space range from 0x00000000 to 0x1FFFFFFF
  - Usually implemented on non-volatile memory, such as on-chip FLASH memory
  - Normally separated from program data, which is allocated in the SRAM region (or data region)



How is Data Stored in RAM?

- Typically, the data can be divided into three sections: static data, stack, and heap
  - Static data: contains global variables and static variables
  - Stack: contains the temporary data for local variables, parameter passing in function calls, registers saving during exceptions, etc.
  - Heap: contains the pieces of memory spaces that are dynamically reserved by function calls, such as ‘alloc()’ and ‘malloc ()’



Defining the Stack and Heap

- The stack and heap can be defined in either C language (with linker file) or assembly language, for example, in C:

```
/* Set stack and heap parameters */

#define  STACK_BASE     0x10020000          // stack start address
#define  STACK_SIZE     0x5000              // length of the stack
#define  HEAP_BASE      0x10001000          // heap starts address
#define  HEAP_SIZE      0x10000 - 0x6000    // heap length

/* inker generated stack base addresses */

extern unsigned int Image$$Arm_LIB_STACK$$ZI$$Limit
extern unsigned int Image$$Arm_LIB_STACKHEAP$$ZI$$Limit

…
```

Defining the Stack and Heap

- The stack and heap can be defined in assembly language as follows:

```
Stack_Size      EQU     0x00000400              ; 256KB of STACK

                AREA    STACK, NOINIT, READWRITE, ALIGN=4
Stack_Mem       SPACE   Stack_Size
__initial_sp

Heap_Size       EQU     0x00000400              ; 1MB of HEAP

                AREA    HEAP, NOINIT, READWRITE, ALIGN=4
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit
```

Data Types

- A number of standard data types are supported by the C language
- However, their implementation depends on the processor architecture and C compiler
- In Arm programming, the data size is referred to as byte, half word, word, and double word:
  - Byte: 8-bit
  - Half word: 16-bit
  - Word: 32-bit
  - Double word: 64-bit

- The table in the next slide shows the implementation of different data types

Data Types

Data type	Size	Signed Range	Unsigned Range
char, int8_t, uint8_t	Byte	-128 to 127	0 to 255
short, int16_t, uint16_t	Half word	-32768 to 32767	0 to 65535
int, int32_t, uint32_t, long	Word	-2147483648 to 2147483647	0 to 4294967295
long long, int64_t, uint64_t	Double word	-2^{63} to $2^{63}-1$	0 to $2^{64}-1$
float	Word	$-3.4028234 \times 10^{38}$ to 3.4028234×10^{38}	
double, long double	Double word	$-1.7976931348623157 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
pointers	Word	0x00 to 0xFFFFFFFF	
enum	Byte/ half word/ word	Smallest possible data type	
bool (C++), _bool(C)	Byte	True or false	
wchar_t	Half word	0 to 65535	

Data Type and Class Qualifiers

Const

- Never written by program, can be put in ROM to save RAM

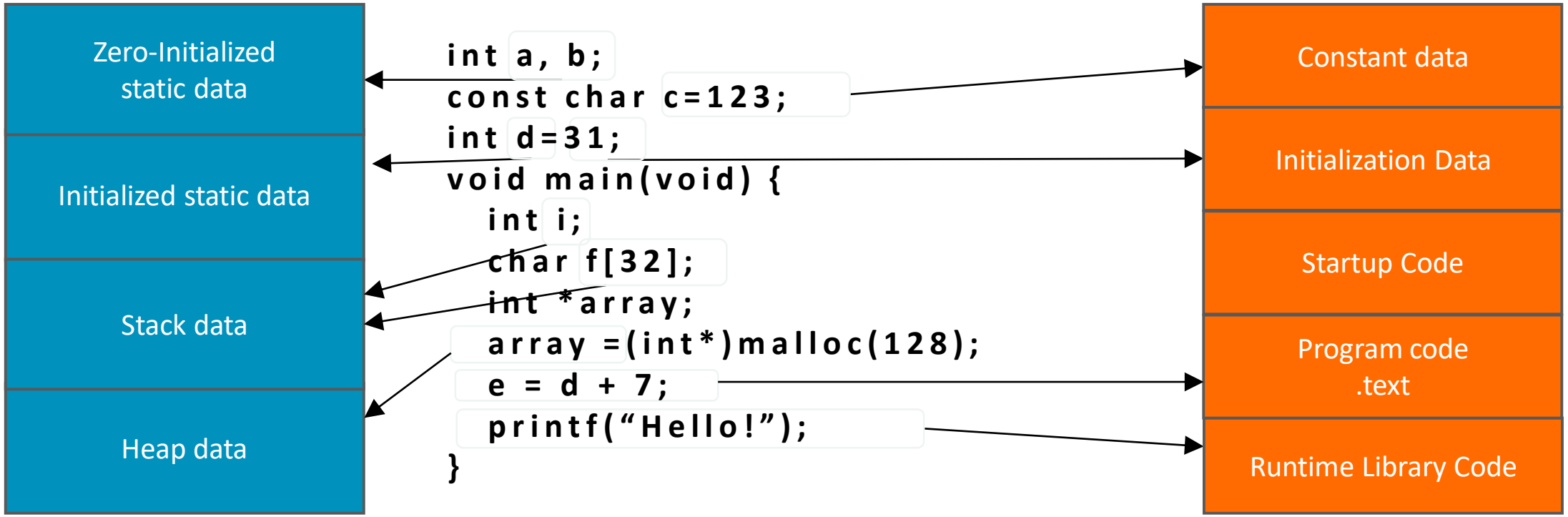
Volatile

- Can be changed outside of normal program flow: ISR, hardware register
- Compiler must be careful with optimizations

Static

- Declared within function, retains value between function invocations
- Scope is limited to function

Example of Data Storage



Usually stored in volatile memories, e.g. SRAM

Usually stored in non-volatile memories e.g. FLASH

arm

Defining the Interrupt Vector in C

- The interrupt vector can be defined in either C language or assembly language, for example, in C:

```
typedef void(* const ExecFuncPtr)(void) __irq;

#pragma arm section rodata="exceptions_area”

ExecFuncPtr exception_table[] = {
          (ExecFuncPtr)&Image$$Arm_LIB_STACK$$ZI$$Limit, /* Initial SP */
          (ExecFuncPtr)__main,                                  /* Initial PC */
          NMIException,
          HardFaultException,
          MemManageException,
          BusFaultException,
          UsageFaultException,
          0, 0, 0, 0,                                           /* Reserved */
          SVCHandler,
          DebugMonitor,
          0,                                                    /* Reserved */
          PendSVC,
          SysTickHandler
          /* Configurable interrupts start here...*/
};
#pragma arm section
```

Accessing Peripherals in C

- Define base addresses for peripherals:

```
#define FLASH_BASE      *((volatile unsigned long *) (0x00000000UL)
#define RAM_BASE        *((volatile unsigned long *) (0x10000000UL)
#define GPIO_BASE       *((volatile unsigned long *) (0x2009C000UL)
#define APB0_BASE       *((volatile unsigned long *) (0x40000000UL)
#define AHB_BASE        *((volatile unsigned long *) (0x50000000UL)
#define CM3_BASE        *((volatile unsigned long *) (0xE0000000UL)
```

- Write a value to a peripheral register:

```
//store a value to the memory
RAM_BASE = 0x3FFFF
```

- Read a value from a peripheral register:

```
//read a value from the memory
i= RAM_BASE;
```

Using Bit-band Operation in Assembly

- For example, in order to set bit[3] in word data in address 0x20000000
  - Read-Modify-Write operation
    - Read the real data address (0x20000000)
    - Modify the desired bit (retain other bits unchanged)
    - Write the modified data back
  - Bit-band operation
    - Directly set the bit by writing ‘ 1’ to address 0x2200000C, which is the alias address of fourth bit of the 32-bit data at 0x20000000

```
;Read-Modify-Write Operation

LDR     R1, =0x20000000  ;Setup address
LDR     R0, [R1]         ;Read
ORR.W   R0, #0x8         ;Modify bit
STR     R0, [R1]         ;Write back
```

```
;Bit-band Operation

LDR     R1, =0x2200000C  ;Setup address
MOV     R0, #1           ;Load data
STR     R0, [R1]         ;Write
```


arm

Use Bit-band Operation in C

- The bit-band operation is not natively supported in most C compilers
- For ease of use of the bit-band feature, the address and the bit-band address can be separately declared, for example:

```
#define RAM_Data1              *((volatile unsigned long *)(0x20000000))
#define RAM_Data1_Bit0         *((volatile unsigned long *)(0x22000000))
#define RAM_Data1_Bit1         *((volatile unsigned long *)(0x22000004))

RAM_Data1= RAM_Data1 | 0x01              ;Setup bit0 without using bit-band
RAM_Data1_Bit0= 0x00                     ;clear bit0 using bit-band
RAM_Data1_Bit1= 0x01                     ;Setup bit1 using bit-band
```

- Alternatively, C macros can be used to easily access bit-band alias, for example:

```
#define RAM_Data1                        *((volatile unsigned long *)(0x20000000))
#define bit-band(data_addr, bit)         ((data_addr & 0x0F0000000) + 0x2000000 +
((data_addr & 0xFFFFF)<<5)+(bit<<2))
#define Data_Addr (data_addr)            *((volatile unsigned long*)(data_addr))

Data_Addr(bit-band(RAM_Data1,0)) = 0x01           ;set bit0
```

Calling a C Function from Assembly

- When a C function is called from an assembly file, be aware of the following areas:
  - Register R0, R1, R2, R3, R12, and LR could be changed; hence, it is better to save them to the stack.
  - The value of SP should be aligned to a double-word address boundary.
  - Input parameters must be stored in the correct registers, for example, registers R0 to R3 can be used for passing four parameters.
  - The return value is usually stored in R0.

Calling a C Function from Assembly

- For example, writing an adding function in C:

```
int my_add(int x1, int x2, int x3, int x4) {
        return (x1+x2+x3+x4);
}
```

- Call the C function from the assembly code, for example:

```
MOVS    R0, #0X1          ; parameter x1
MOVS    R1, #0X3          ; parameter x2
MOVS    R2, #0X5          ; parameter x3
MOVS    R3, #0X7          ; parameter x4
IMPORT  my_add
BL      my_add            ; call “my_add” function in C
```

Calling an Assembly Function from C

- When calling an assembly function from C code, you should be aware of the following:
  - If registers R4 to R11 need to be changed, they must be stacked and restored in the assembly function
  - If another function is called inside the assembly function, the LR needs to be saved on the stack and used for return
  - The function return value is normally stored in R0

Calling an Assembly Function from C

- Writing a function in assembly, for example:

```
                EXPORT add_asm
add_asm         FUNCTION
                ADDS    R0, R0, R1
                ADDS    R0, R0, R2
                ADDS    R0, R0, R3
                BX      LR                  ; result is returned in R0
                ENDFUNC
```

- Calling an assembly function in C, for example:

```
external int  add_asm( int k1, int k2, int k3, int k4);
void main {
        int x;
        x = add_asm (11,22,33,44);          // call assembly function
        …
}
```


arm

Embedded Assembly

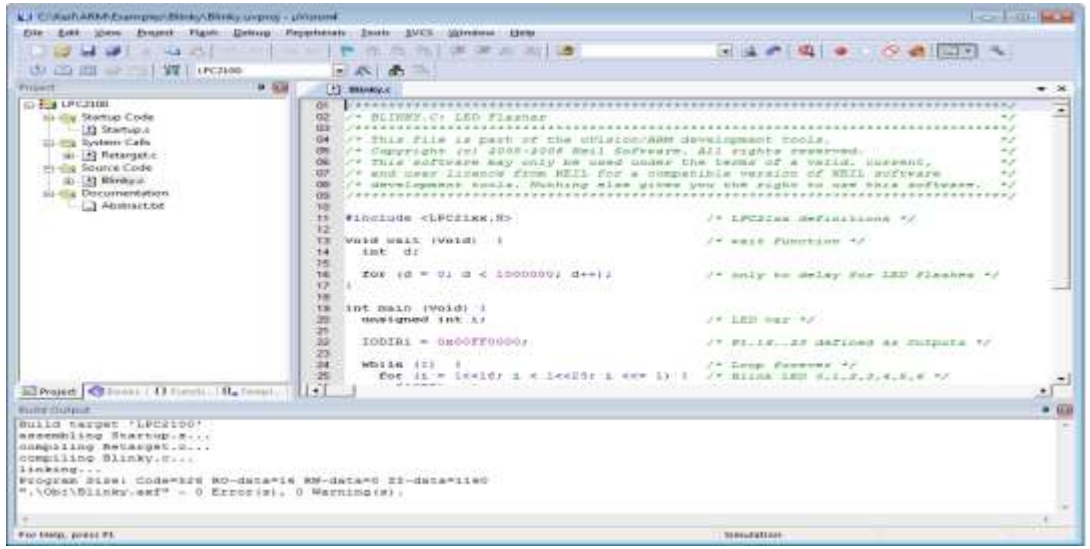
- The embedded assembler allows developers to write assembly functions inside C files, for example, in C:

```
_asm int  add_asm(  int k1, int k2, int k3, int k4) {

                ADDS     R0, R0, R1
                ADDS     R0, R0, R2
                ADDS     R0, R0, R3
                BX       LR
}

void main {
        int x;
        x = add_asm (11,22,33,44);        // call assembly function
        …
}
```

Arm Keil® MDK

- Keil µVision has an Integrated Development Environment (IDE), which allows the user to build a project easily and quickly. The IDE includes:
  - Project management
  - Build facilities
  - Source code editing
  - Program debugging
  - Complete simulation

- A series of Arm-based tools are integrated in Keil µVision, including:
  - Compiler
  - Assembler
  - Linker
  - Format converter
  - Libraries



Resources

- Cortex-M4 Technical Reference Manual: https://developer.arm.com/docs/100166/0001

- Cortex-M4 Devices Generic User Guide: https://developer.arm.com/docs/dui0553/b

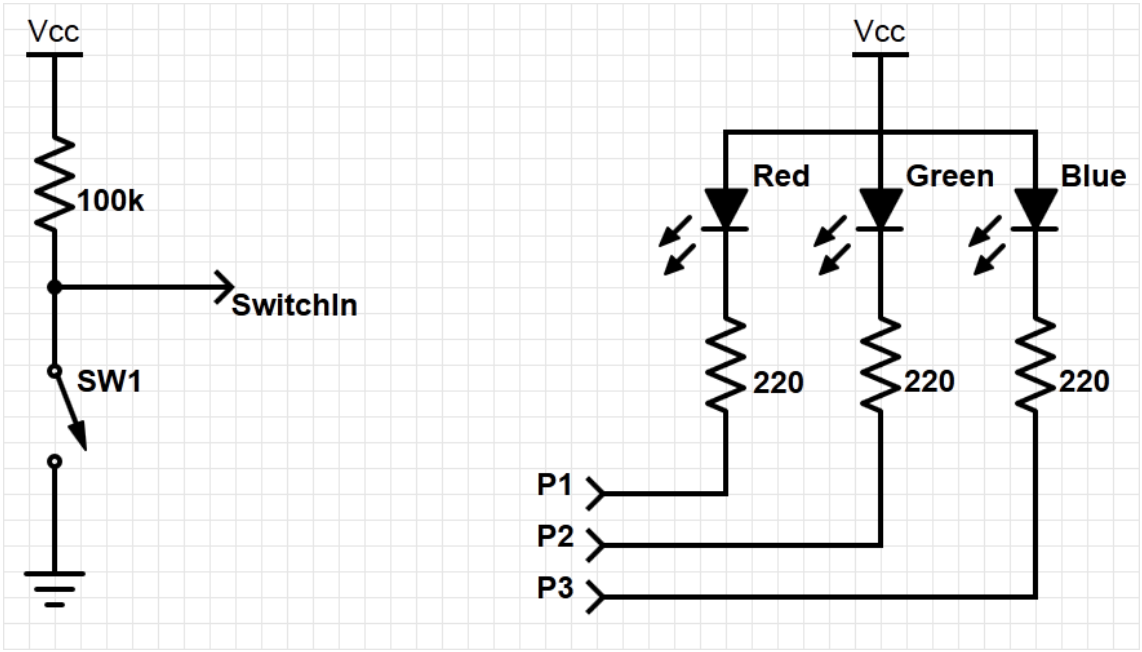
arm

Interrupts and Low Power Features

Module Syllabus

- Interrupts
  - What are interrupts?
  - Why use interrupts?
- Exceptions
  - Entering an exception handler
  - Exiting an exception handler
- Microcontroller interrupts
- Timing analysis
- Program design with interrupts
- Sharing data safely between ISRs and other threads

Example System with Interrupt



- Goal: to change the colour of the RGB LED when the switch is pressed
  - Need to add an external switch
    - Resistor is internal – a pull-up resistor

arm

Detecting Switch Presses

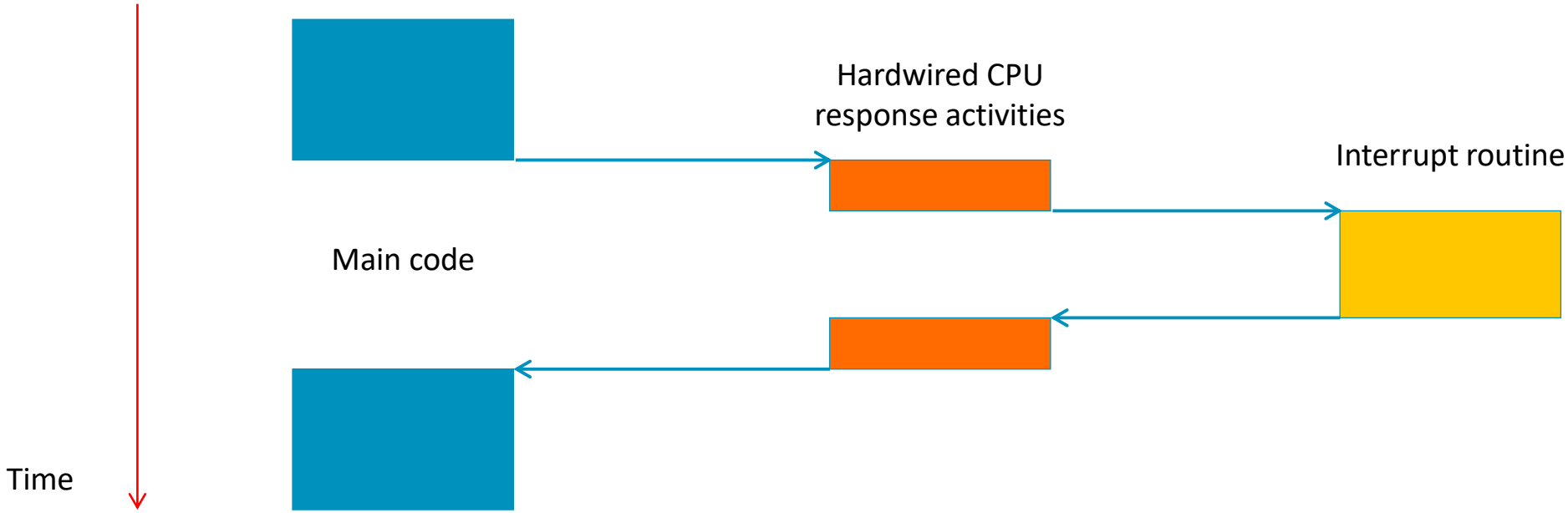
- One option is polling, i.e., using software to check it regularly. However, polling is:
  - Slow: user needs to check to see if the switch is pressed
  - Wasteful of CPU time: the faster the response needed, the more often the user needs to check
  - Not scalable: it's difficult to build a multi-activity system that can respond quickly. The system's response time depends on all other processing it has to do

- A better option is an interrupt, i.e., using special hardware in the MCU to detect and run ISR in response. An interrupt is:
  - Efficient: the code runs only when necessary
  - Fast: it's a hardware mechanism
  - Scalable:
    - ISR response time doesn't depend on most other processing
    - Code modules can be developed independently

Interrupt or Exception Processing Sequence

- The main code is running. When the interrupt trigger occurs:
  - The processor does some hard-wired processing
  - The processor executes the ISR, including return-from-interrupt instruction at the end

- Then the processor resumes running the main code



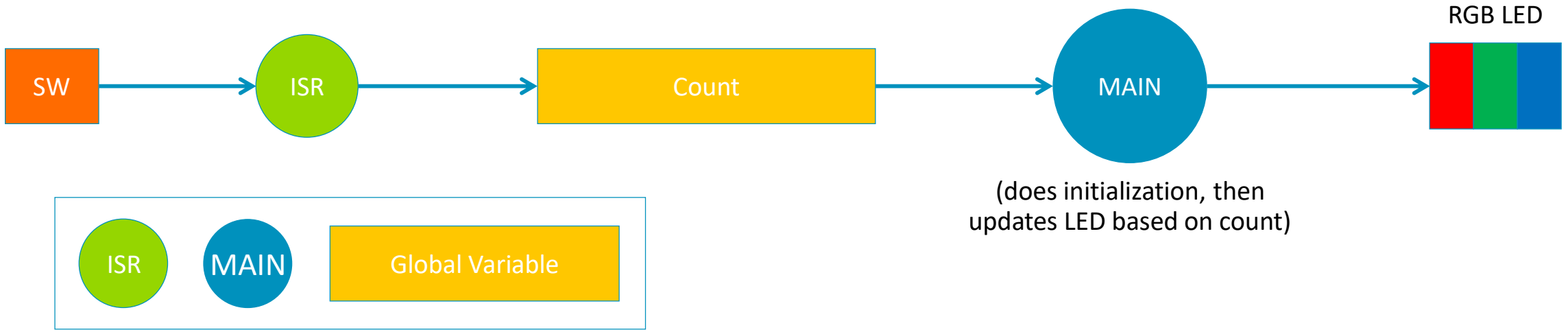
arm

Interrupts

- Hardware-triggered asynchronous routine
  - Triggered by hardware signal from peripheral or external device
  - Asynchronous: can happen anywhere in the program (unless interrupt is disabled)
  - Software routine: interrupt service routine runs in response to interrupt

- Fundamental mechanism of microcontrollers
  - Provides efficient event-based processing rather than polling
  - Provides quick response to events, regardless of program state, complexity, and location
  - Allows many multithreaded embedded systems to be responsive without an operating system (specifically task scheduler)

Example Program Requirements and Design



- When switch SW is pressed, ISR will increment count variable
- Main code will light LEDs according to count value in binary sequence (Blue: 4, Green: 2, Red: 1)
- Main code will toggle its debug line each time it executes
- ISR will raise its debug line (and lower main's debug line) whenever it is executing

arm

Example Exception Handler

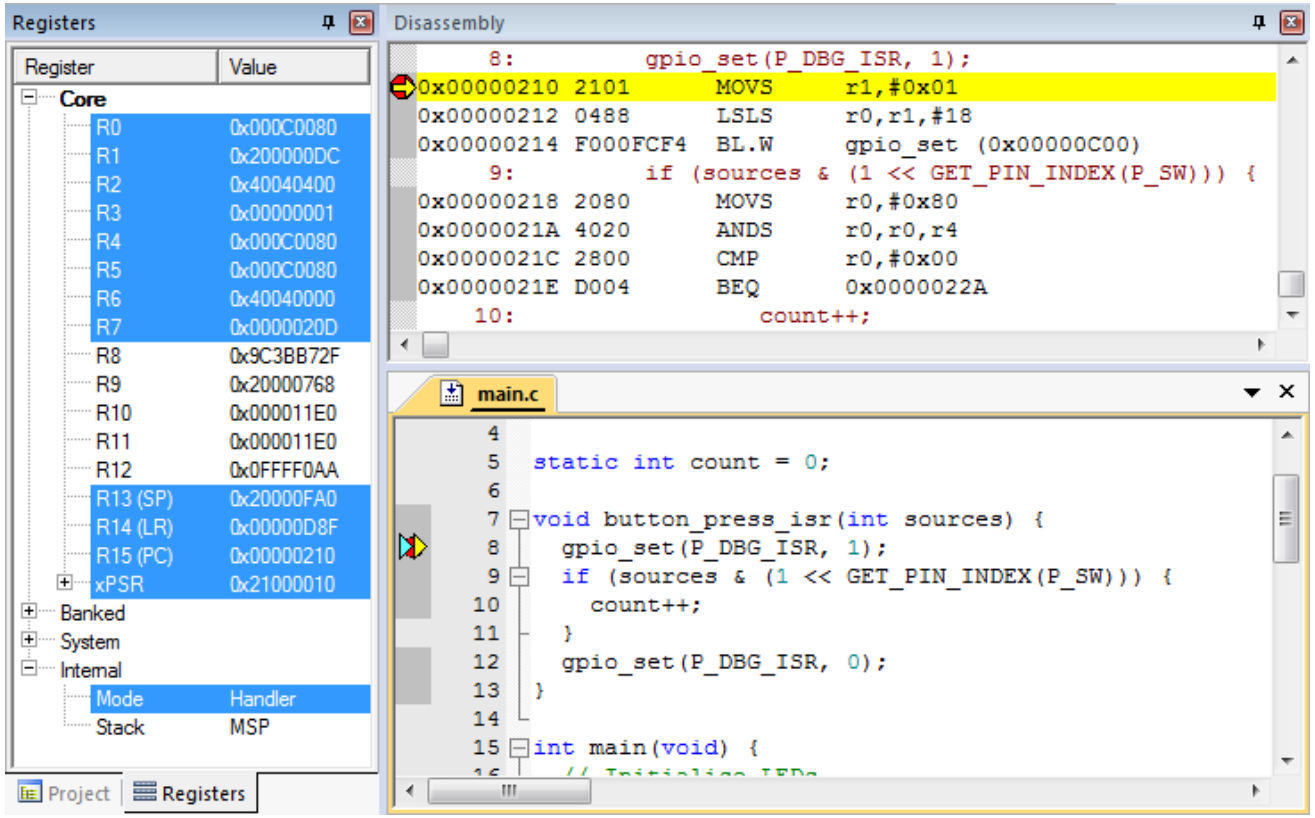
- Now, we will examine the processor's response to an exception in detail:

main.c

```
4
5   static int count = 0;
6
7   void button_press_isr(int sources) {
8     gpio_set(P_DBG_ISR, 1);
9     if (sources & (1 << GET_PIN_INDEX(P_SW))) {
10      count++;
11    }
12    gpio_set(P_DBG_ISR, 0);
13  }
14
15  int main(void) {
16    // Initialise LEDs.
17    leds_init();
18    leds_set(0, 0, 0);
19
20    // Set up debug signals.
21    gpio_set_mode(P_DBG_ISR, Output);
```

Using the Debugger for a Detailed Processor View

- Can see registers, stack, source code, disassembly (object code)
- Note: compiler may generate code for function entry
- Place breakpoint on the handler function declaration line in source code, not at the first line of code



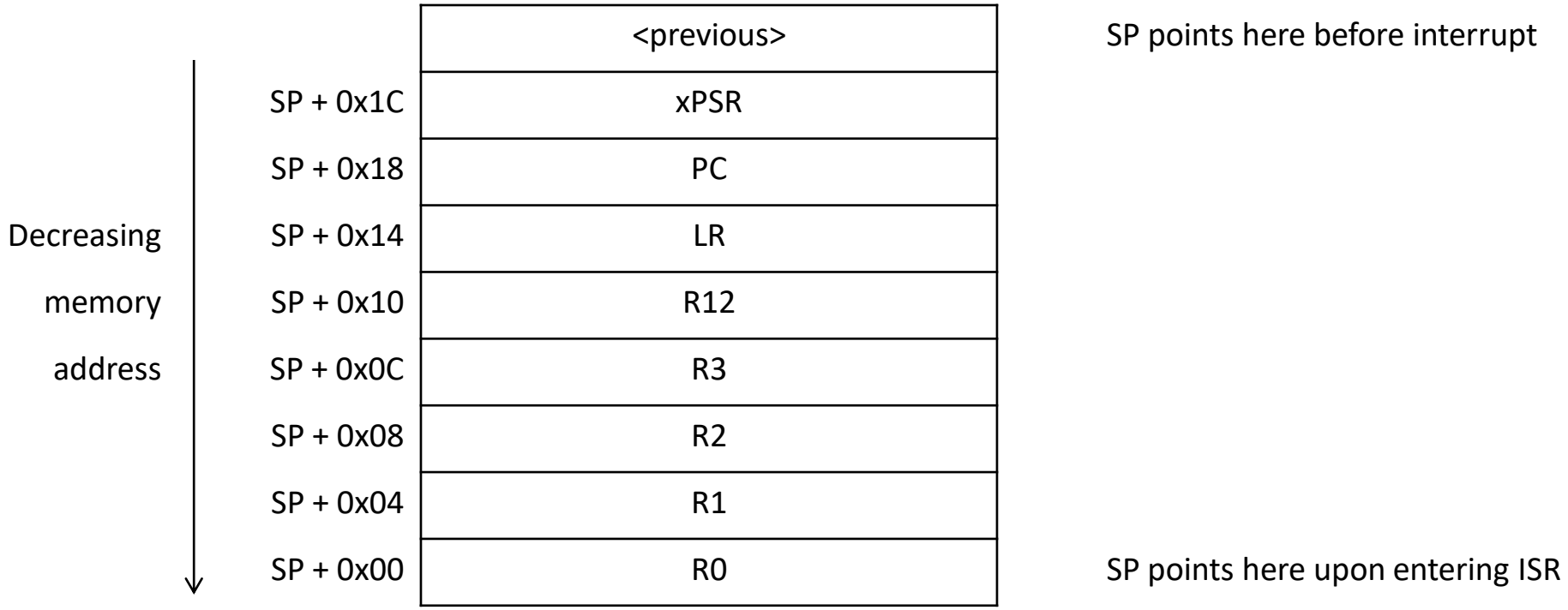
Entering Exception Handler: CPU Hardwired Exception Processing

- Finish current instruction
  - Except for lengthy instructions
- Push context (registers) onto current stack (MSP or PSP)
  - xPSR, return address, LR (R14, R12, R3, R2, R1, R0)
- Switch to handler/privileged mode, use MSP
- Load PC with address of interrupt handler
- Load LR with EXC_RETURN code
- Load IPSR with exception number
- Start executing code of interrupt handler
- Usually 16 cycles from exception request to execution of first instruction in handler

1. Finish Current Instruction

- Most instructions are short and finish quickly.
- Some instructions may take many cycles to execute.
  - Load Multiple (LDM), Store Multiple (STM), Push, Pop, MULS (32 cycles for some CPU core implementations)
- This will delay interrupt response significantly.
- If one of these is executing when the interrupt is requested, the processor:
  - abandons the instruction
  - responds to the interrupt
  - executes the ISR
  - returns from interrupt
  - restarts the abandoned instruction

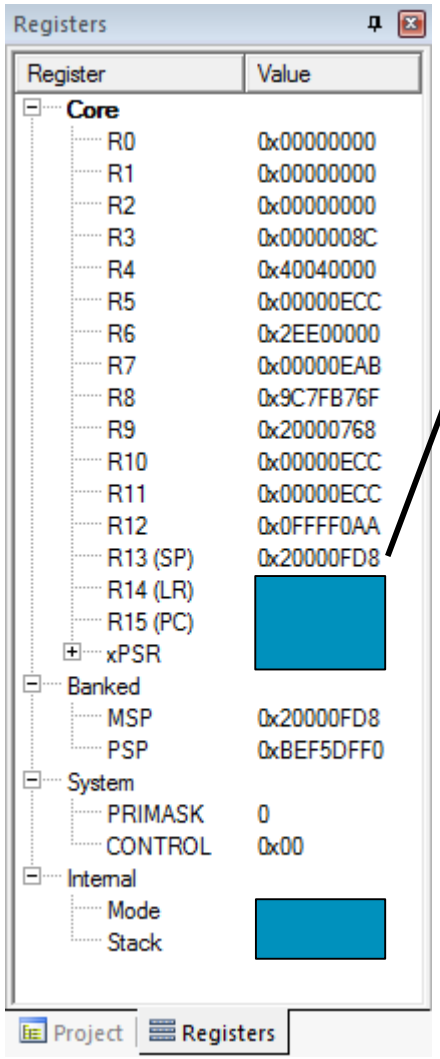
2. Push Context onto Current Stack



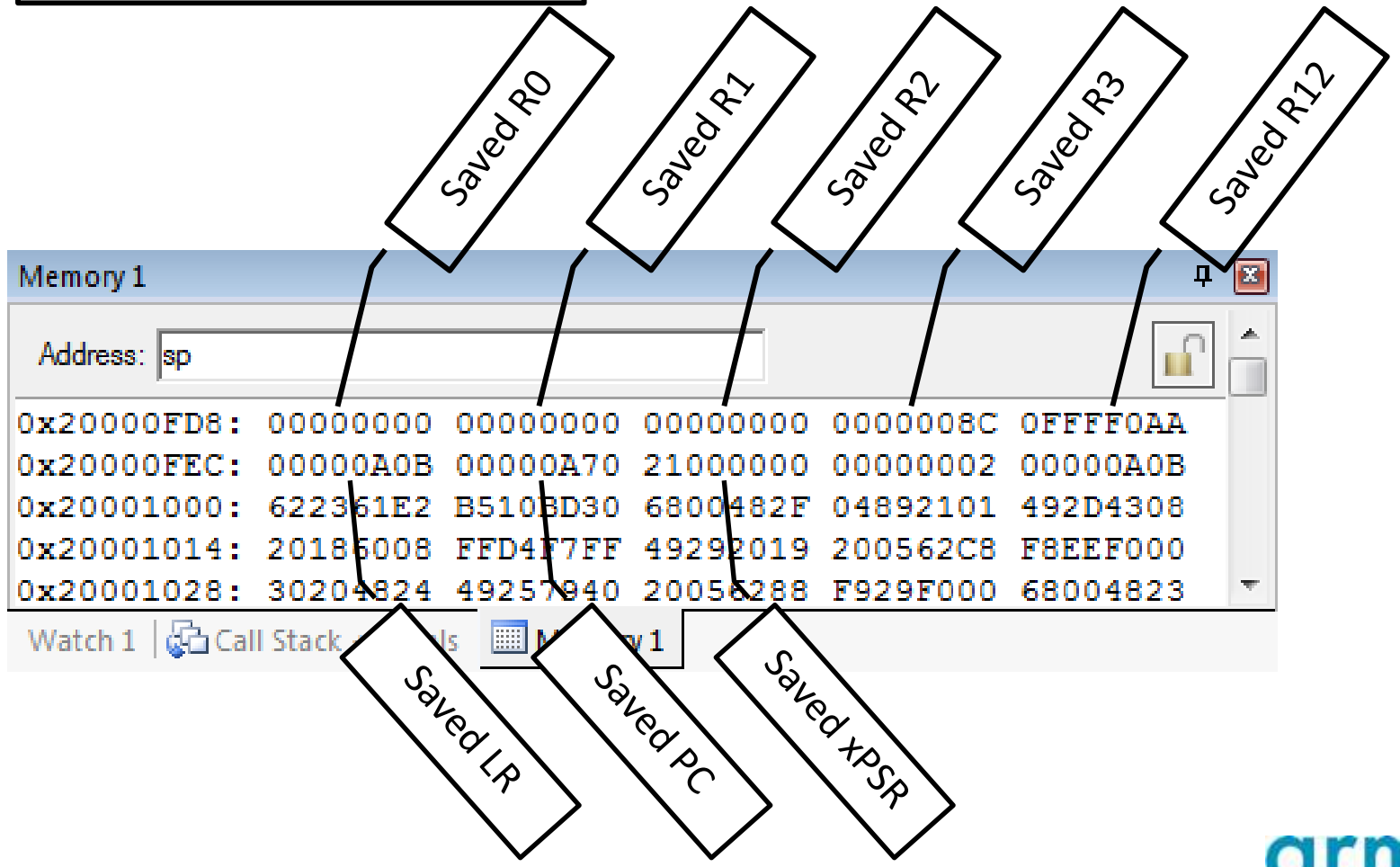
- Two SPs: main (MSP), process (PSP)
- Which is active depends on operating mode, CONTROL register bit 1.
- Stack grows toward smaller addresses.

arm

Context Saved on Stack

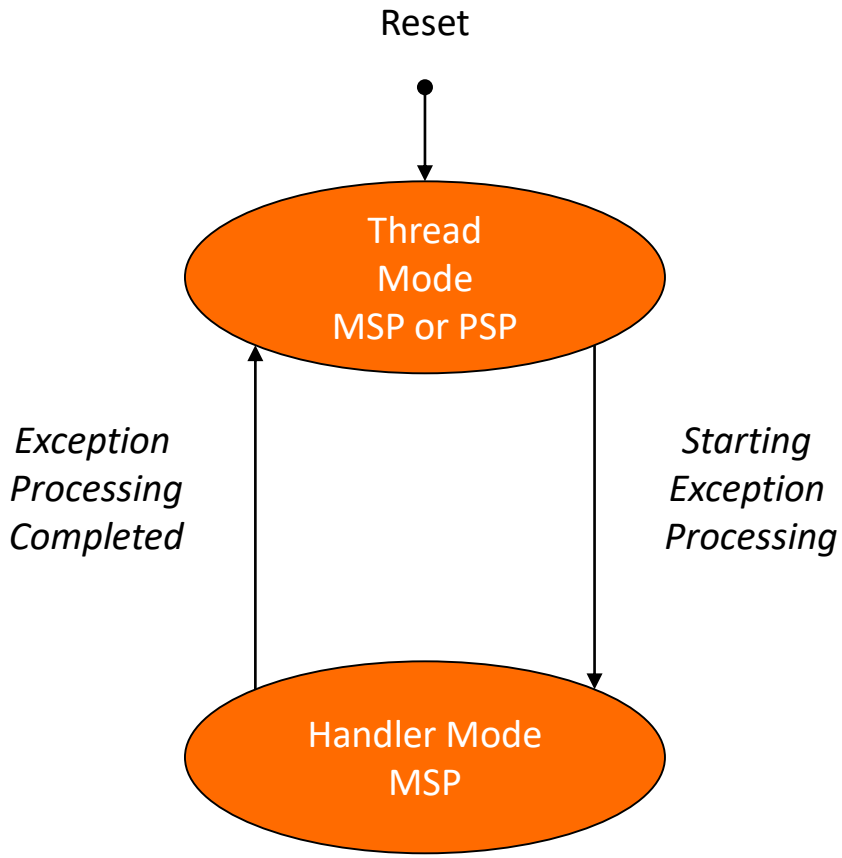


SP value is reduced since registers have been pushed onto stack.

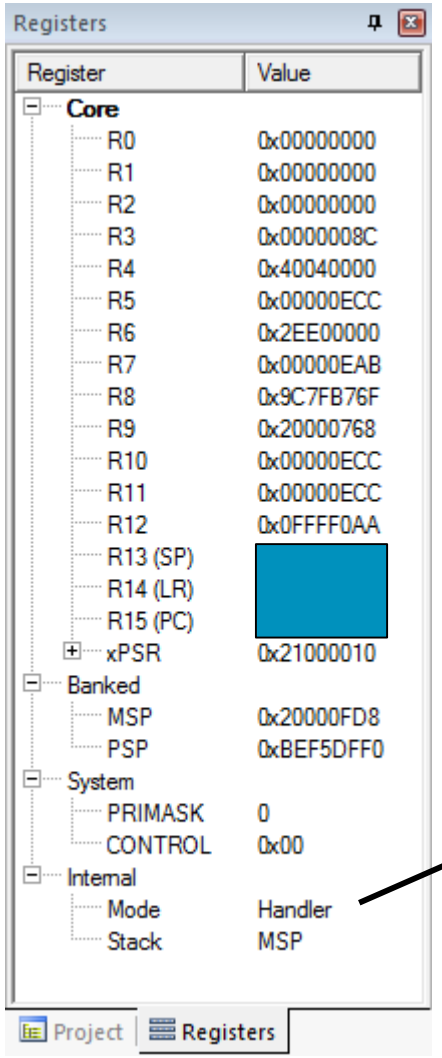


3. Switch to Handler/Privileged Mode

- Handler mode always uses main SP

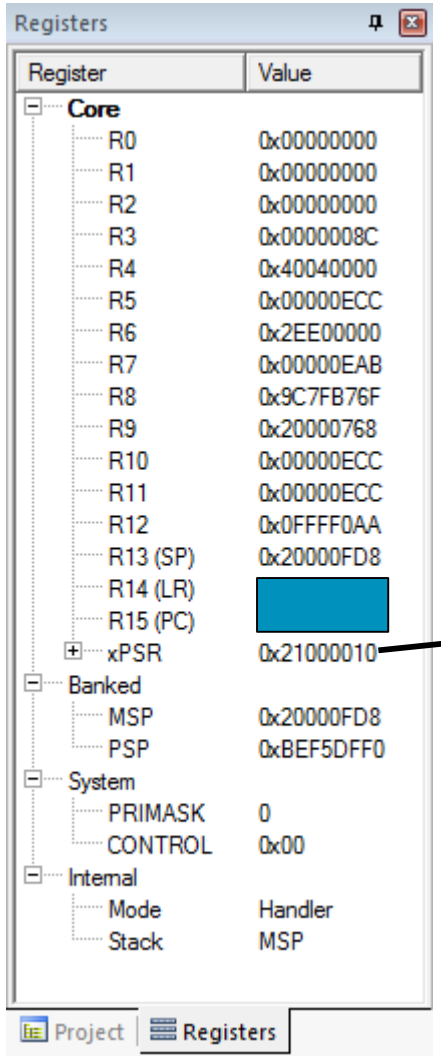


Handler and Privileged Mode



Mode changed to Handler.
Was already using MSP.

Update IPSR with Exception Number



Exception number 0x10
(interrupt number + 0x10)

arm

4. Load PC with Address of Exception Handler

Memory Address	Value
0x0000_0000	Initial Stack Pointer
0x0000_0004	Reset
0x0000_0008	NMI_IRQHandler
…	
	IRQ0_Handler
	IRQ1_Handler
…	
Reset:	
…	
NMI_IRQHandler:	
…	
IRQ0_Handler:	
…	
IRQ1_Handler:	

- Which program counter is selected from the vector table depends on which exception is used.

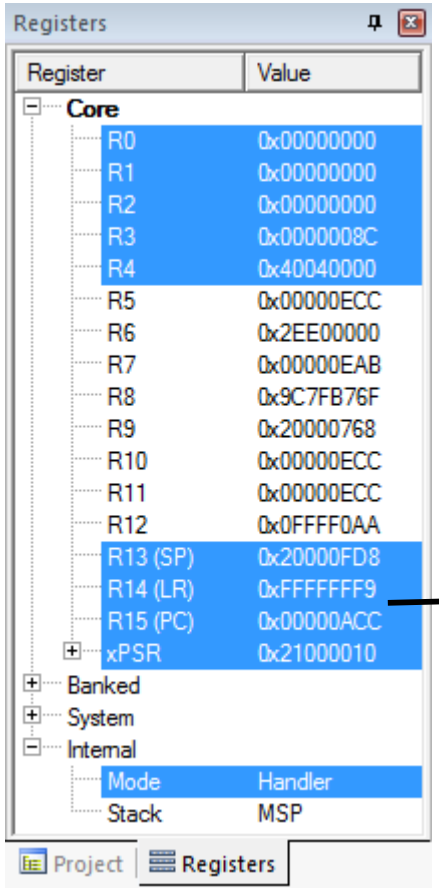
Examine Vector Table with Debugger

Exception Number	IRQ Number	Vector	Offset
		Initial SP	0x00
1		Reset	0x04
2	-14	NMI	0x08
3	-13	HardFault	0x0C
4		Reserved	0x10
5			
6			
7			
8			
9			
10			
11	-5	SVCall	0x2C
12		Reserved	
13			
14	-2	PendSV	0x38
15	-1	SysTick	0x3C
16	0	IRQ0	0x40
17	1	IRQ1	0x44
18	2	IRQ2	0x48
.		.	
16+n	n	IRQn	0x40+4n

- Why is the vector odd?
- LSB of address indicates that handler uses Thumb code.

arm

Upon Entry to Handler




```
    42: void switch_isr(void) {
0x00000ACC B510      PUSH     {r4,lr}
```

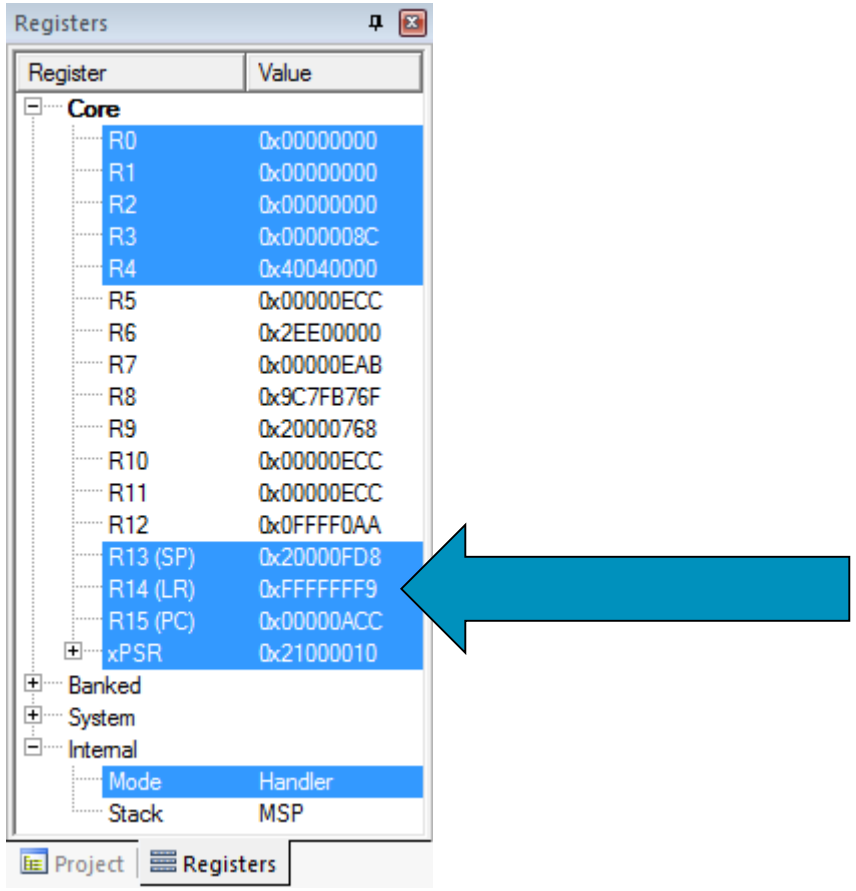
PC has been loaded with start address of handler.

5. Load LR with EXC_RETURN Code

EXC_RETURN	Return Mode	Return Stack	Description
0xFFFF_FFF1	0 (Handler)	0 (MSP)	Return to exception handler
0xFFFF_FFF9	1 (Thread)	0 (MSP)	Return to thread with MSP
0xFFFF_FFFD	1 (Thread)	1 (PSP)	Return to thread with PSP

- EXC_RETURN value generated by CPU to provide information on how to return
  - Which SP to restore registers from: MSP (0) or PSP (1)?
    - Previous value of SPSEL
  - Which mode to return to: Handler (0) or Thread (1)?
    - Another exception handler may have been running when this exception was requested

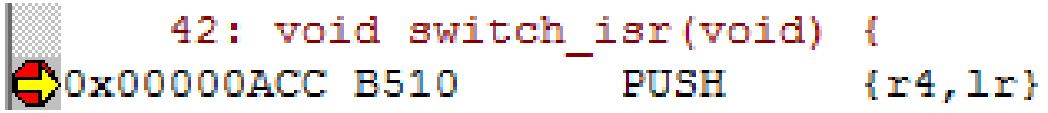
Updated LR with EXC_RETURN Code



6. Start Executing Exception Handler

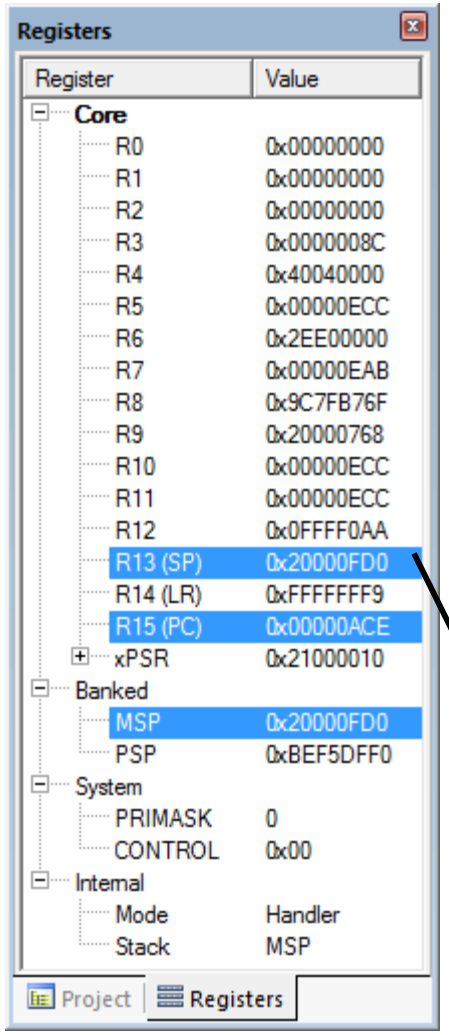
- Exception handler starts running, unless preempted by a higher-priority exception.

- Exception handler may save additional registers on stack.
  - e.g., if handler may call a subroutine, LR and R4 must be saved.

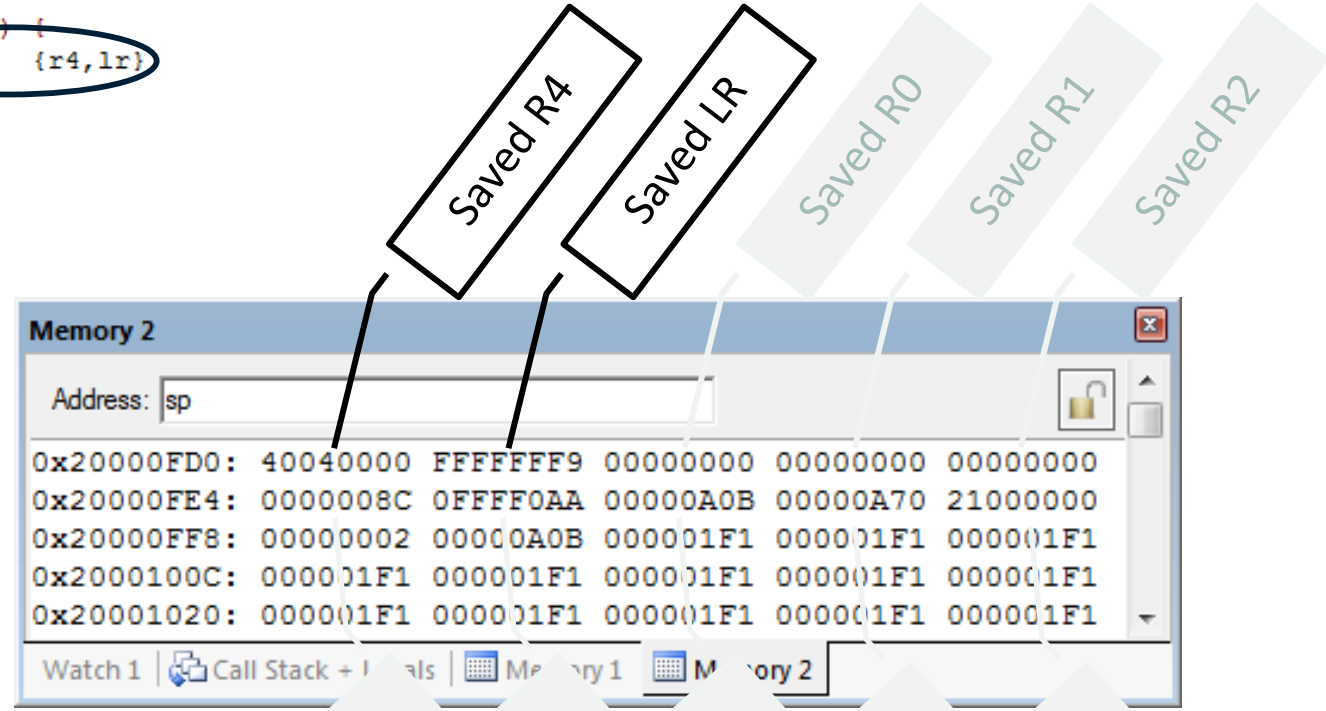


arm

After Handler Has Saved More Context




```
    42: void switch_isr(void) {
0x00000ACC B510      PUSH     {r4,lr}
```



SP value reduced since registers have been pushed onto stack.

arm

Exiting an Exception Handler

1. Execute instruction triggering exception return processing

2. Select return stack, restore context from that stack

3. Resume execution of code at restored address

1. Execute Instruction for Exception Return

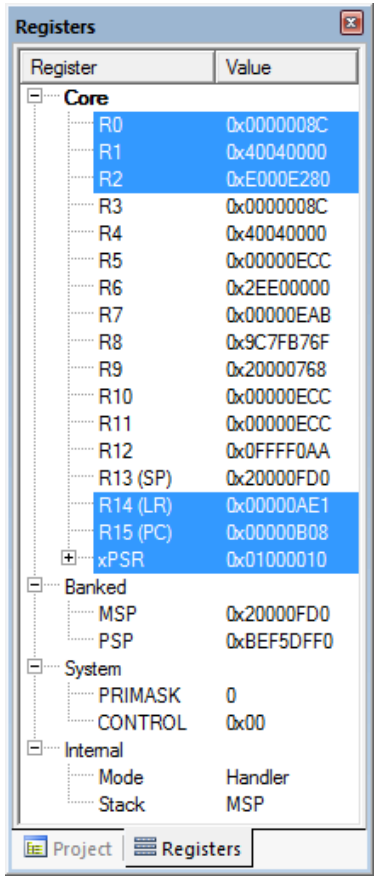
- No ‘return from interrupt’ instruction
- Use regular instruction instead
  - BX LR - Branch to address in LR by loading PC with LR contents
  - POP …, PC - Pop address from stack into PC
- … with a special value EXC_RETURN loaded into the PC to trigger exception handling processing
  - BX LR used if EXC_RETURN is still in LR
  - If EXC_RETURN has been saved on stack, then use POP

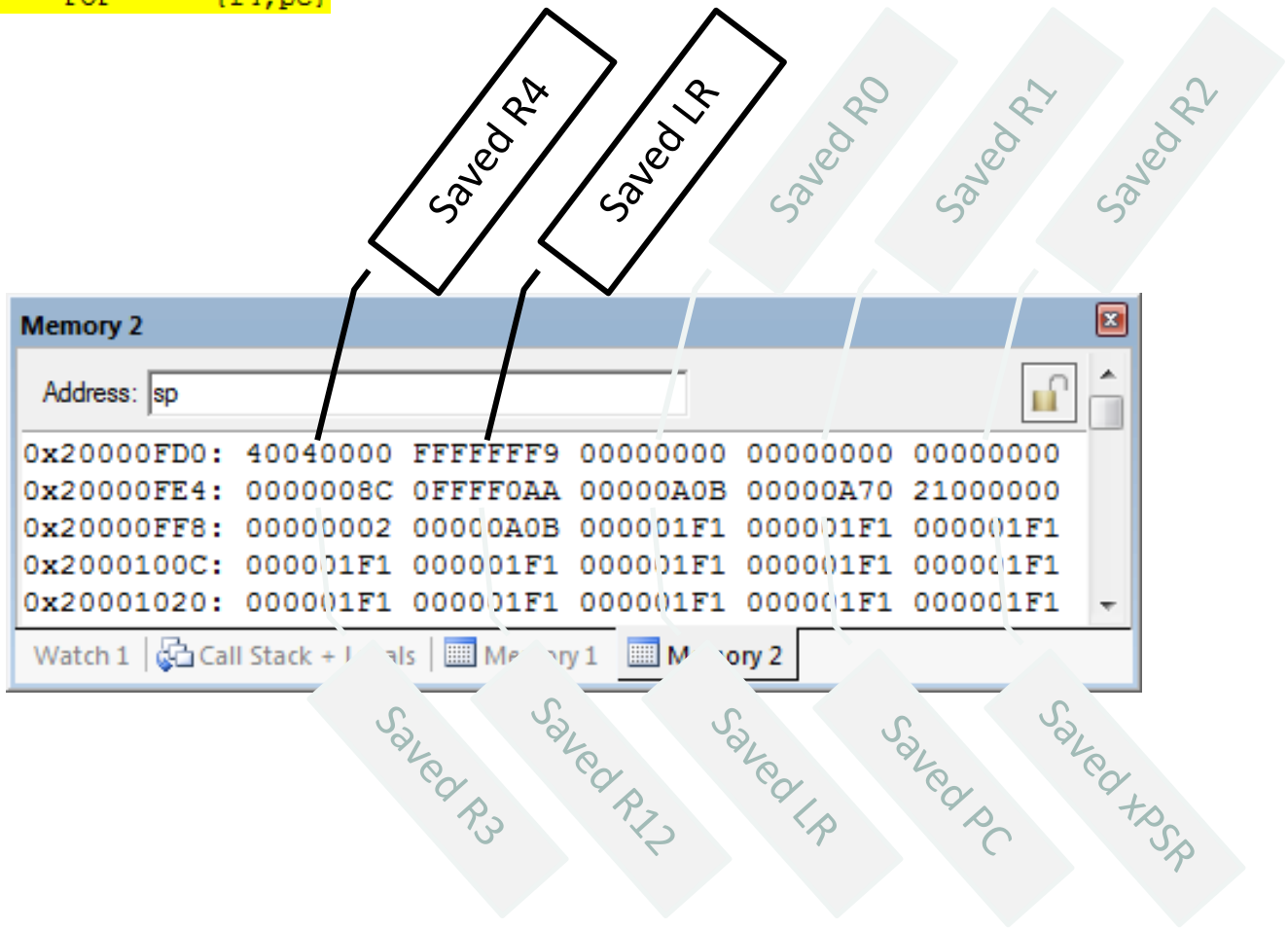
arm

What Will Be Popped from Stack?

- R4: 0x4040_0000
- PC: 0xFFFF_FFF9

```
    51: }
0x00000B08 BD10      POP      {r4,pc}
```



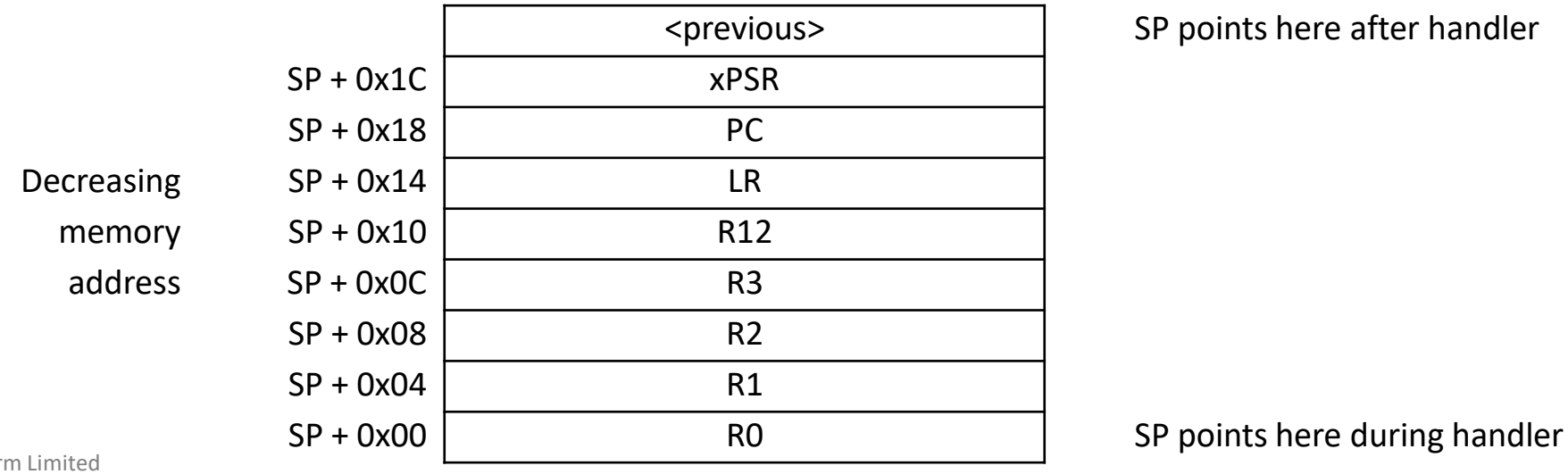


2. Select Stack, Restore Context

- Check EXC_RETURN (bit 2) to determine from which SP to pop the context

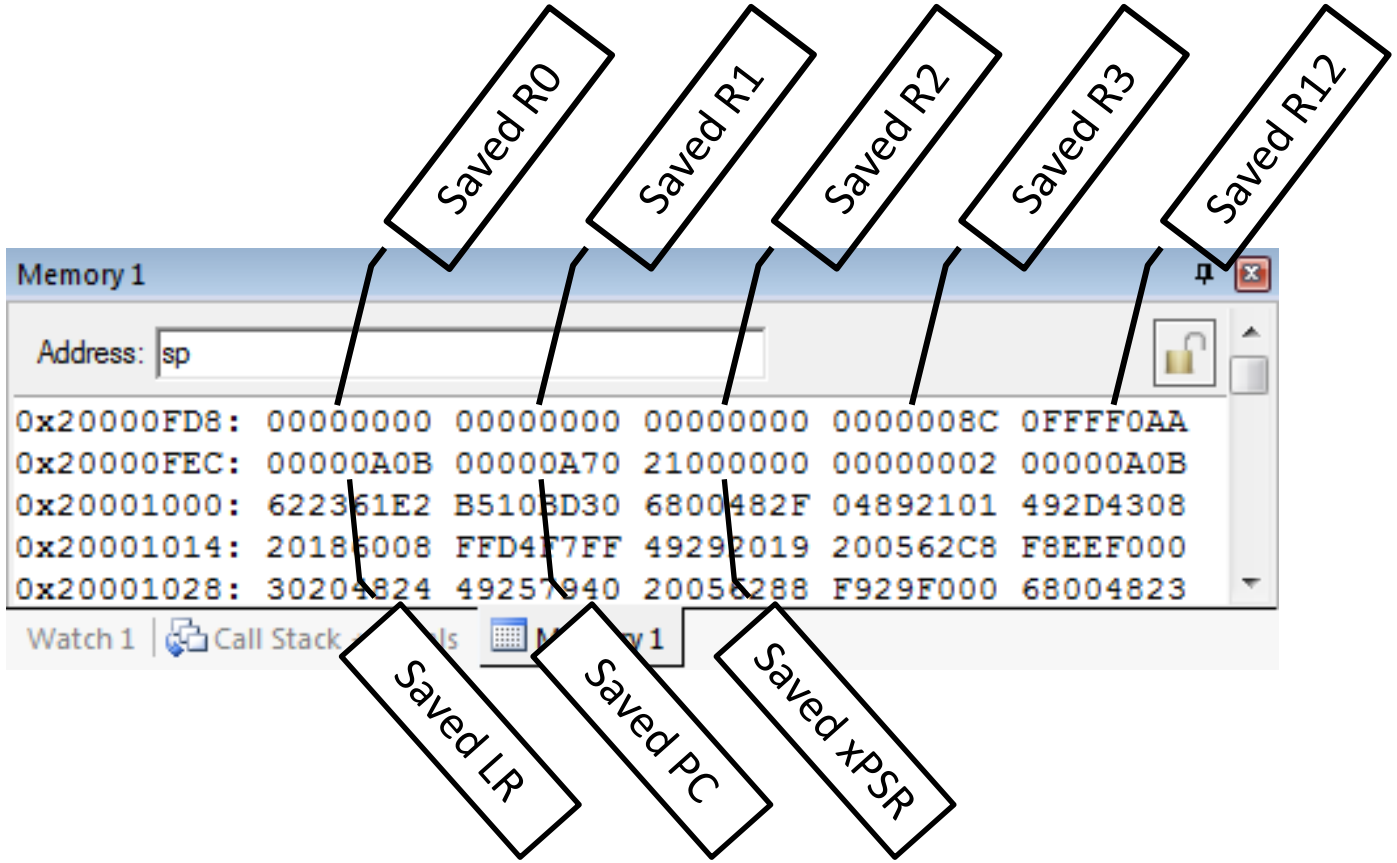
EXC_RETURN	Return Stack	Description
0xFFFF_FFF1	0 (MSP)	Return to exception handler with MSP
0xFFFF_FFF9	0 (MSP)	Return to thread with MSP
0xFFFF_FFFD	1 (PSP)	Return to thread with PSP

- Pop the registers from that stack



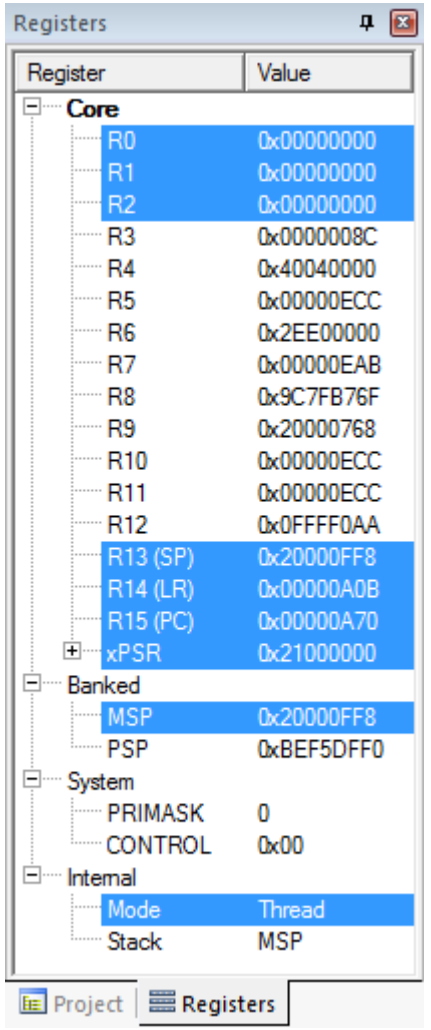
Example

- PC=0xFFFF_FFF9, so return to thread mode with main stack pointer
- Pop exception stack frame from stack back into registers



Resume Executing Previous Main Thread Code

- Exception handling registers have been restored: R0, R1, R2, R3, R12, LR, PC, xPSR
- SP is back to previous value
- Back in thread mode
- Next instruction to execute is at 0x0000_0A70



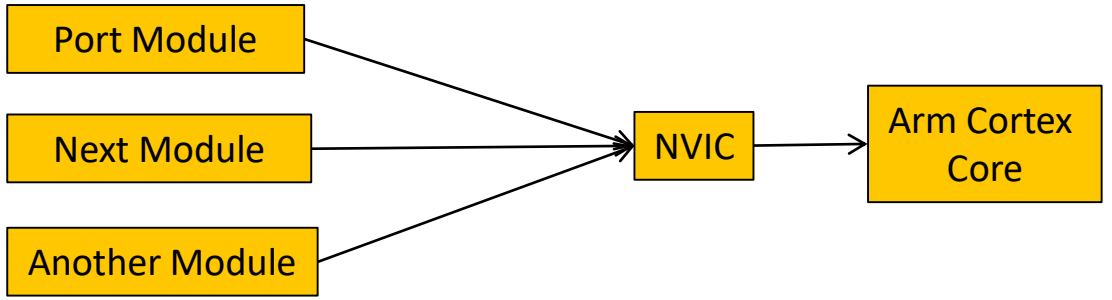
Microcontroller Interrupts

- Types of interrupts
  - Hardware interrupts
    - *Asynchronous*: not related to what code the processor is currently executing
    - Examples: interrupt is asserted, character is received on serial port, or ADC converter finishes conversion
  - Exceptions, faults, software interrupts
    - *Synchronous*: the result of specific instructions executing
    - Examples: undefined instructions, overflow occurs for a given instruction
  - We can enable and disable (*mask*) most interrupts as needed (*maskable*), others are *non-maskable*

- Interrupt service routine (ISR)
  - Subroutine which processor is *forced to execute* to respond to a *specific event*
  - After ISR completes, MCU goes back to previously executing code

Nested Vectored Interrupt Controller

- NVIC manages and prioritizes external interrupts.
- Interrupts are types of exceptions.
  - Exceptions 16 through 16+N
- Modes
  - Thread Mode: entered on reset
  - Handler Mode: entered on executing an exception
- Privilege level
- Stack pointers
  - Main Stack Pointer, MSP
  - Process Stack Pointer, PSP
- Exception states: Inactive, Pending, Active, A&P



NVIC Registers and State

- Enable: allows interrupt to be recognized
  - Accessed through two registers (set bits for interrupts)
    - Set enable with NVIC_ISER, clear enable with NVIC_ICER
  - CMSIS Interface: NVIC_EnableIRQ(IRQnum), NVIC_DisableIRQ(IRQnum)

- Pending: interrupt has been requested but is not yet serviced
  - CMSIS: NVIC_SetPendingIRQ(IRQnum), NVIC_ClearPendingIRQ(IRQnum)

Core Exception Mask Register

- Similar to ‘Global interrupt disable’ bit in other MCUs

- PRIMASK: Exception mask register (CPU core)
  - Bit 0: PM flag
    - Set to 1 to prevent activation of all exceptions with configurable priority
    - Clear to 0 to allow activation of all exception
  - Access using CPS, MSR and MRS instructions
  - Use to prevent data race conditions with code needing atomicity

- CMSIS-CORE API
  - void __enable_irq() - clears PM flag
  - void __disable_irq() - sets PM flag
  - uint32_t __get_PRIMASK() - returns value of PRIMASK
  - void __set_PRIMASK(uint32_t x) - sets PRIMASK to x

Prioritization

- Exceptions are prioritized to order the response simultaneous requests (smaller number = higher priority)

- Priorities of some exceptions are fixed
  - Reset: -3, highest priority
  - NMI: -2
  - Hard Fault: -1

- Priorities of other (peripheral) exceptions are adjustable.
  - Value is stored in the interrupt priority register (IPR0-7).
  - 0x00
  - 0x40
  - 0x80
  - 0xC0

Special Cases of Prioritization

- Simultaneous exception requests?
  - Lowest exception type number is serviced first

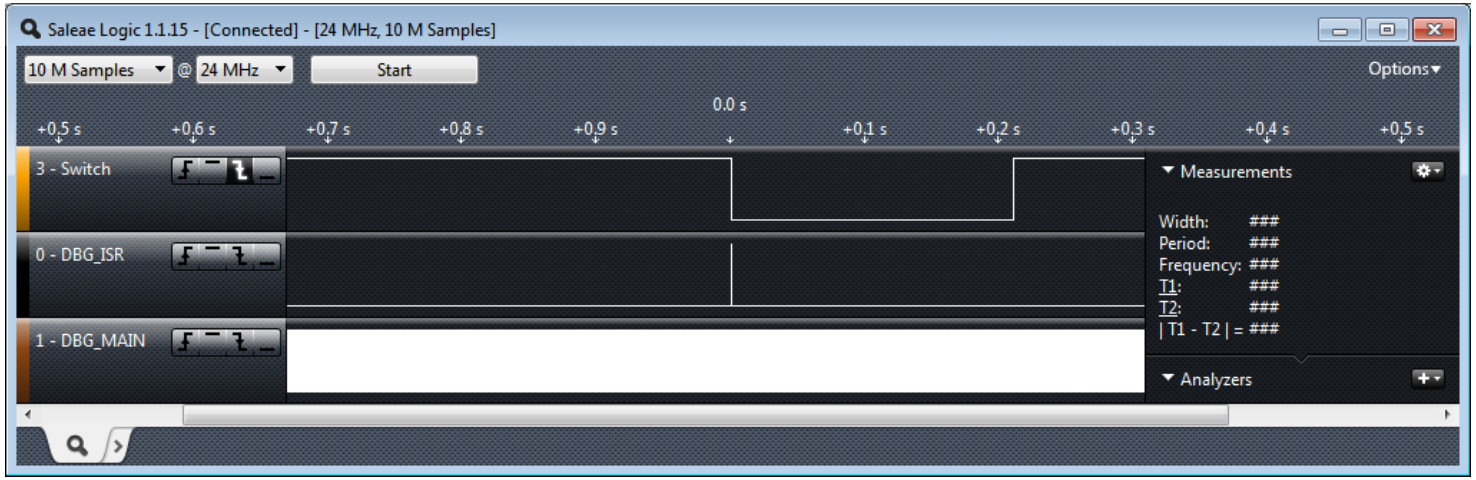
- New exception requested while a handler is executing?

  - New priority higher than current priority?
    - New exception handler preempts current exception handler.

  - New priority lower than or equal to current priority?
    - New exception held in pending state .
    - Current handler continues and completes execution.
    - Previous priority level restored.
    - New exception handled if priority level allows.

arm

Timing Analysis: Big Picture Timing Behaviour



- Switch was pressed for about 0.21s.
- ISR runs in response to switch signal's falling edge.
- Main seems to be running continuously (signal toggles between 1 and 0).

arm

Interrupt Response Latency

- Latency = time delay

- Why do we care?
  - This is overhead, which wastes time and increases as the interrupt rate rises
  - This delays our response to external events, which may or may not be acceptable for the application, such as sampling an analog waveform

- How long does it take?
  - Finish executing the current instruction or abandon it
  - Push various registers on to the stack, fetch vector
    - $C_{IntResponseOvhd}$: Overhead for responding to each interrupt
  - If we have external memory with wait states, this takes longer

Maximum Interrupt Rate

- We can only handle so many interrupts per second.
  - F_{Max_Int}: Maximum interrupt frequency
  - F_{CPU}: CPU clock frequency
  - C_{ISR}: Number of cycles ISR takes to execute
  - $C_{Overhead}$: Number of cycles of overhead for saving state, vectoring, restoring state, etc.
  - $F_{Max\_Int} = F_{CPU}/(C_{ISR}+ C_{Overhead})$
  - Note that model applies only when there is one interrupt in the system.

- When processor is responding to interrupts, it isn't executing our other code.
  - U_{Int}: Utilization (fraction of processor time) consumed by interrupt processing
  - $U_{Int} = 100\%*F_{Int}* (C_{ISR}+C_{Overhead})/ F_{CPU}$
  - CPU looks like it's running the other code with CPU clock speed of $(1-U_{Int})*F_{CPU.}$

Program Design with Interrupts

- How much work to do in ISR?
  - Trade-off: faster response for ISR code will delay completion of other code
  - In system with multiple ISRs with short deadlines, perform critical work in ISR and buffer partial results for later processing
- Should ISRs re-enable interrupts?
- How to communicate between ISR and other threads?
  - Data buffering
  - Data integrity and race conditions
    - Volatile data: can be updated outside of the program's immediate control
    - Non-atomic shared data: can be interrupted partway through read or write, is vulnerable to race conditions

Volatile Data

- Compilers assume that variables in memory don't change spontaneously, and optimize based on that belief.
  - *Don't reload a variable from memory if current function hasn't changed it.*
  - Read variable from memory into register (faster access)
  - Write back to memory at the end of the procedure, or before a procedure call, or when compiler runs out of free registers
- This optimization can fail.
  - Example: reading from input port, polling for key press
    - While (SW_0) ; will read from SW_0 once and reuse that value
    - Will generate an infinite loop triggered by SW_0 being true
- Variables for which it fails
  - Memory-mapped peripheral register: register changes on its own
  - Global variables modified by an ISR: ISR changes the variable
  - Global variables in a multithreaded application: another thread or ISR changes the variable

The Volatile Directive

- We need to tell compiler which variables may change outside of its control.
  - Use volatile keyword to force compiler to reload these variables from memory for each use

```
volatile unsigned int num_ints;
```

  - Pointer to a volatile int

```
volatile int * var; // or
int volatile * var;
```

  - Now each C source read of a variable (e.g. status register) will result in an assembly language LDR instruction.

Non-Atomic Shared Data

- We want to keep track of current time and date.
- Use 1 Hz interrupt from timer
- System:
  - current_time structure tracks time and days since some reference event
  - current_time's fields are updated by periodic 1 Hz timer ISR

```
void GetDateTime(DateTimeType * DT){
 DT->day = current_time.day;
 DT->hour = current_time.hour;
 DT->minute = current_time.minute;
 DT->second = current_time.second;
}
```

```
void DateTimeISR(void){
 current_time.second++;
 if (current_time.second > 59){
   current_time.second = 0;
   current_time.minute++;
   if (current_time.minute > 59) {
     current_time.minute = 0;
     current_time.hour++;
     if (current_time.hour > 23) {
        current_time.hour = 0;
       current_time.day++;
       … etc.
     }
```

Example: Checking the Time

- Problem: An interrupt at the wrong time will lead to half-updated data in DT.
- Failure Case
  - current_time is {10, 23, 59, 59} (10th day, 23:59:59)
  - Task code calls GetDateTime(), which copies the current_time fields to DT: day = 10, hour = 23
  - A timer interrupt occurs, which updates current_time to {11, 0, 0, 0}.
  - GetDateTime() resumes executing, copying the remaining current_time fields to DT: minute = 0, second = 0
  - DT now has a time stamp of {10, 23, 0, 0}.
  - *The system thinks time just jumped backwards one hour!*
- Fundamental problem: 'race condition'
  - Preemption enables ISR to interrupt other code and possibly overwrite data
  - Must ensure *atomic (indivisible)* access to the object
    - Native atomic object size depends on processor's instruction set and word size
    - 32 bits for Arm

Examining the Problem More Closely

- Protect any data object which both:
  - Requires multiple instructions to read or write (non-atomic access), and
  - Is potentially written by an ISR
- How many tasks/ISRs can write to the data object?
  - If one, then we have one-way communication
    - Must *ensure the data isn't overwritten* partway through being *read.*
    - Writer and reader don't interrupt each other.
  - If more than one, we
    - Must *ensure the data isn't overwritten* partway through being *reader*
      - Writer and reader don't interrupt each other.
    - Must *ensure the data isn't overwritten* partway through being *written*
      - Writers don't interrupt each other.

Definitions

- **Race condition**

‘Anomalous behavior due to unexpected critical dependence on the relative timing of events. Result of example code depends on the relative timing of the read and write operations.’

- **Critical section**

‘A section of code which creates a possible race condition. The code section can only be executed by one process at a time. Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use.’

Solution: Briefly Disable Preemption

- Prevent preemption within critical section
- If an ISR can *write* to the shared data object, need to *disable interrupts*
  - save current interrupt masking state in m
  - disable interrupts
- Restore *previous state* afterwards (interrupts may have already been disabled for another reason)
- Use CMSIS-CORE to save, control and restore interrupt masking state
- Avoid disabling interrupts. Disabling delays response to all other processing requests
- Use as few instructions as possible, to make the time as short as possible

```
void GetDateTime(DateTimeType * DT){
 uint32_t m;

 m = __get_PRIMASK();
 __disable_irq();

 DT->day = current_time.day;
 DT->hour = current_time.hour;
 DT->minute = current_time.minute;
 DT->second = current_time.second;
 __set_PRIMASK(m);
}
```

arm

Digital Input and Output (IO)

Module Syllabus

- Digital input and output (IO)
  - Voltages and logic values
- General-purpose IO (GPIO) controller
  - Using pointers to access GPIO
  - Define data structure for peripherals
- Mbed API Digital I/O
  - DigitalIn/DigitalOut
  - BusIn/BusOut
- Digital IO examples
  - Using LED
  - Using 7-segment display
  - Using infrared emitter/detector

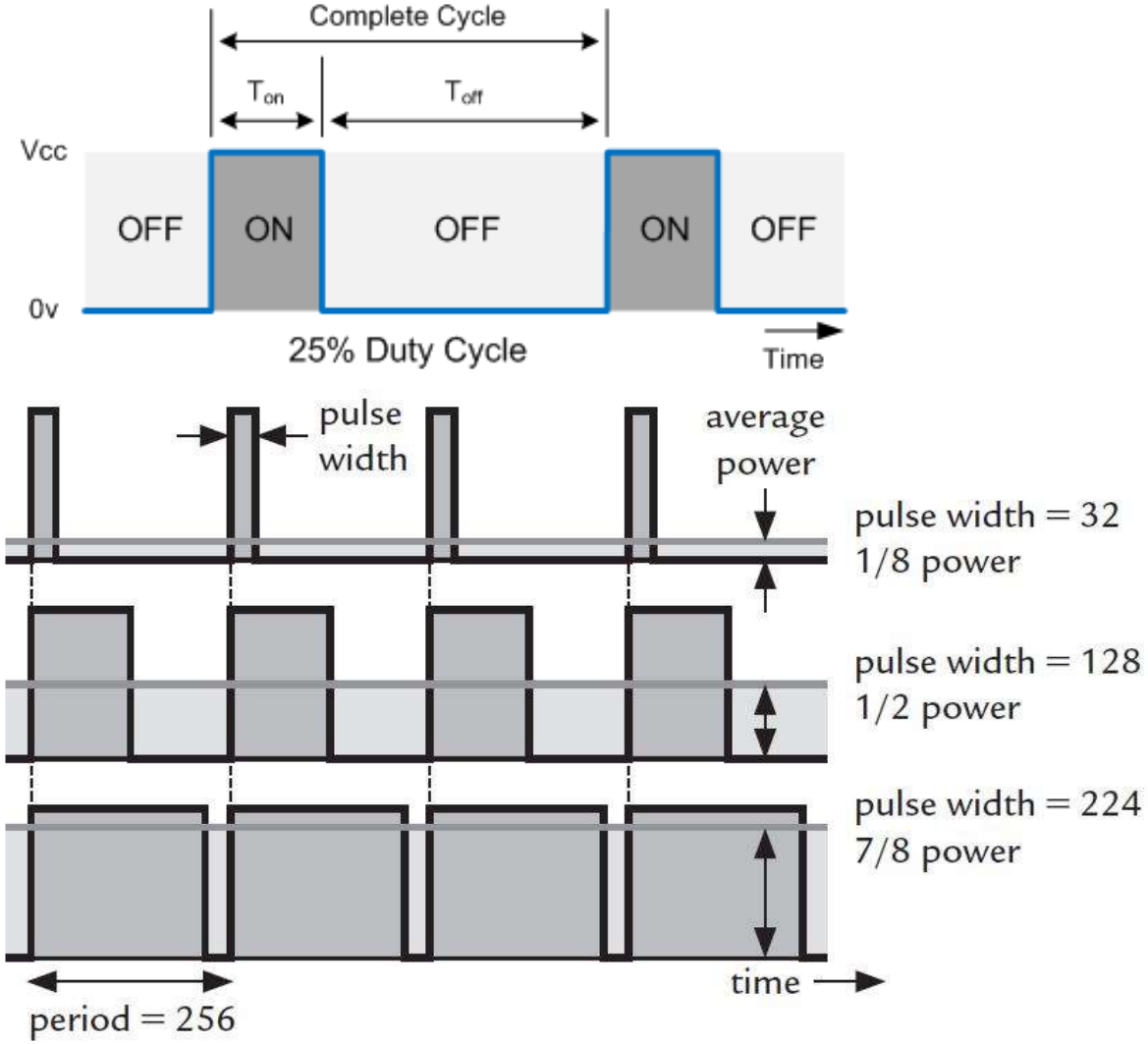
Voltages and Logic Values

- In digital devices, logical values : ‘ 1’ or ‘0’, are represented as electrical voltages
- Different devices may use different voltages to represent a logic value
  - For example, the external pins of the Nucleo F401RE platform use 3.3 volts for logic ‘1’, and 0 volts for logic ‘0 ’
- Different logic can, however, have different meanings in different contexts depending on the interpretation we give to different voltage levels

Logic	Voltage	Boolean	Circuit	Switch
‘1’	3.3V	True	Closed	On
‘0’	0V	False	Open	Off

Example: Pulse Width Modulation (PWM)

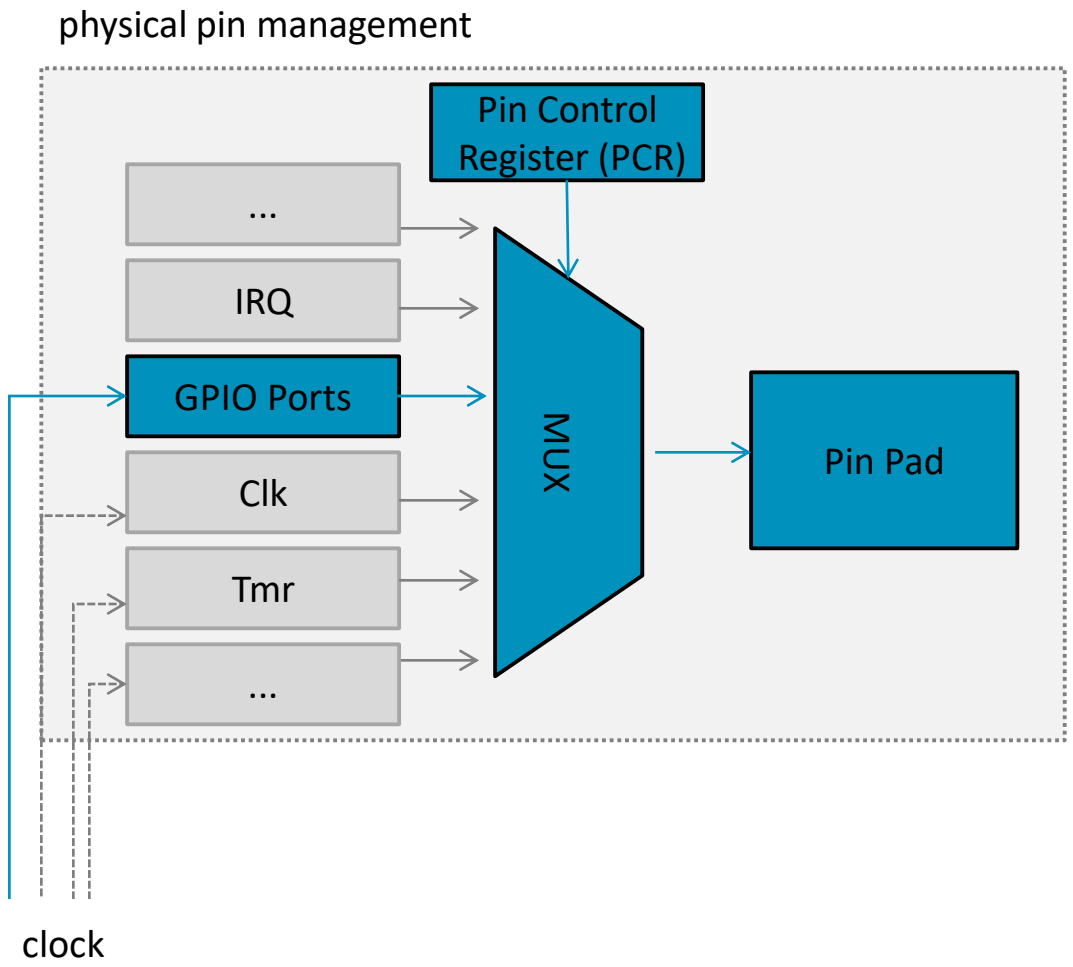
- Generating analog-like signal digitally by modulation of power:
  - Digital signals can only turn signals on and off -> generate 2 values
  - Using pulse width modulation, analog-like signals can be generated, varying the average electrical power over time
  - Change ratio between on and off times within one periodic cycle
- Duty-cycle: $t_{on} / (t_{on} + t_{off})$ $= t_{on} / t_{cycle}$
- Application
  - Control of voltage-sensitive components such as LEDs or servos



https://students.cs.byu.edu/~clement/cs224/labs/L06a-morse/images/PWM_duty_cycle.jpg

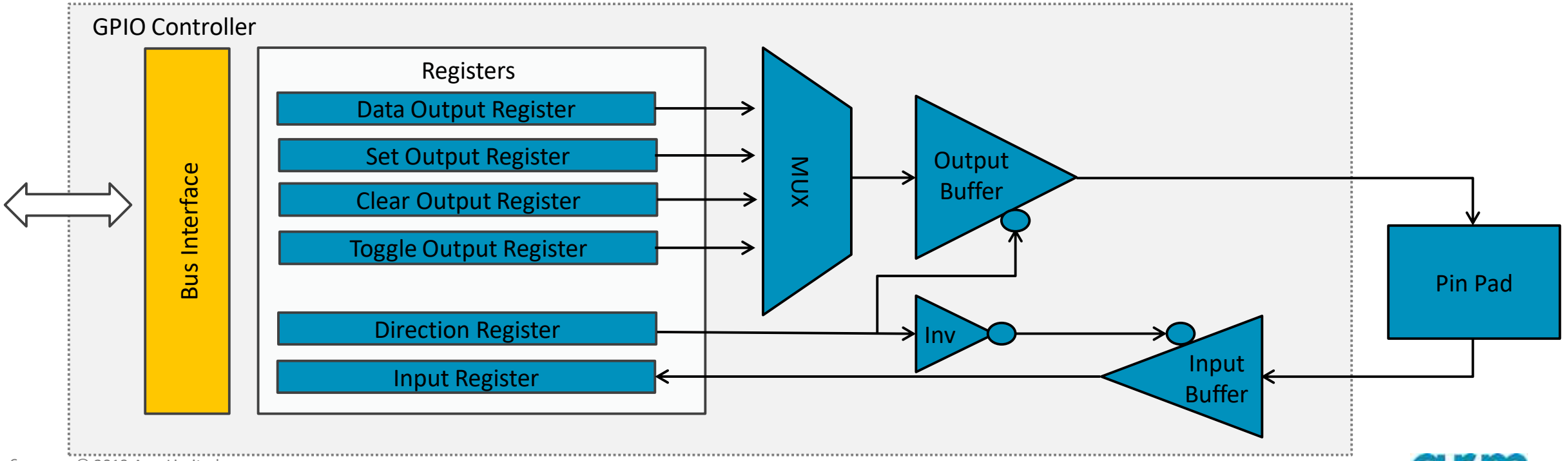
GPIO Design

- Each physical pin is connected to a multiplexer
  - Allows each pin to perform several functions
  - Optimizes functionality for small packages
  - Each port is connected to microcontroller via peripheral bridge
    - Additional settings are required to make virtual pins on physical pins (pin pad) available
  - Signal multiplexer and other pin options can be configured in the pin control register (PCR).
- GPIO pins split into groups, called ports
  - Each port has 32 virtual pins [31:0] assigned
- To reduce power consumption, clocks are turned off for particular areas
  - Default setting: clock is disabled
  - To use a peripheral or a resource, clock has to be enabled for each item individually



GPIO Design

- Normally, the external pins are not directly accessible, but accessed via a peripheral called general-purpose input/output (GPIO).
  - Used for general purpose, no special usage defined
  - Widely used for most applications
  - The direction of input/output is controlled by the direction register



arm

Methods for Programming Digital GPIO

- We will look at two ways of programming digital GPIO:

- Low-Level: GPIO Registers
  - Slower to write
  - More understanding of underlying functionality

- High-Level: Mbed API
  - Quicker to write
  - Less understanding of underlying functionality

GPIO Registers

- For example, in the Nucleo F401RE MCU, there are eight GPIO peripherals named from PORTA-PORTH, and each of them has 10 registers, which include:
  - **Port Mode Register (MODER)** - configure the I/O direction mode (input/output/alternate/analog)
  - **Output Type Register (TYPER)** – configure the output type of the I/O port (push-pull/open-drain)
  - **Output Speed Register (OSPEEDR)** – configure the I/O output speed (2/25/50/100 MHz)
  - **Pull-Up/Pull-Down Register (PUPDR)** – configure the I/O pull-up or pull-down (no pull-up, pull-down/pull-up/pull-down)
  - **Input Data Register (IDR)** – contain the input value of the corresponding I/O port
  - **Output Data Register (ODR)** – can be read and written by software (ODR bits can be individually set and reset by writing to the BSRR)
  - **Bit Set/Reset Register (BSRR)** – can be used for atomic bit set/reset
  - **Configuration Lock Register (LCKR)** – used to lock the configuration of the port bits when a correct write sequence is applied to bit 16
  - **Alternate Function Low Register (AFRL)** – configure alternate function I/Os
  - **Alternate Function High Register (AFRH)** – configure alternate function I/Os

Using Pointer to Access GPIO

- Normally, you can use a pointer to directly access a GPIO peripheral in either C or assembly. For example in C, to light an LED on PB_10:

```
#define RCC_AHB1ENR (*((volatile unsigned long *) 0x40023830))
#define GPIOB_MODER (*((volatile unsigned long *) 0x40020400))
#define GPIOB_PUPDR (*((volatile unsigned long *) 0x40020408))
#define GPIOB_OSPEEDR         (*((volatile unsigned long *) 0x4002040C))
#define GPIOB_ODR             (*((volatile unsigned long *) 0x40020414))

RCC_AHB1ENR |= 0x02;
GPIOB_MODER |= 0x00100000;
GPIOB_PUPDR |= 0x00200000;
GPIOB_OSPEEDR |= 0x00200000;
GPIOB_ODR |= 0x0400;
```

- This solution is fine for simple applications. However, if multiple instantiations of the same type of peripheral are available at the same time, we will need to define registers for each peripheral, which makes code maintenance difficult

- On the other hand, since each register is defined as a separate pointer, each register access requires a 32-bit address constant. As a result, the program image will consume a larger memory space.

Define a Data Structure for Peripherals

- To further simplify the code and reduce its length, we can:
  - Define the peripheral register set as a data structure,
  - Define the peripheral as a memory pointer to this data structure:

- For example:

```
typedef struct {
     volatile unsigned int  MODER;
     volatile unsigned int  OTYPER;
     volatile unsigned int  OSPEEDR;
     volatile unsigned int  PUPDR;
     volatile unsigned int  IDR;
     volatile unsigned int  ODR;
     volatile unsigned int  BSRRL;
     volatile unsigned int  BSRRH;
     volatile unsigned int  LCKR;
     volatile unsigned int  AFR[2];
 } GPIO_TypeDef;
```

Define a Data Structure for Peripherals

- Then, to turn on an LED on PB_10:

```
#define GPIOB_BASE     0x40020400

#define GPIOB((GPIO_TypeDef *) GPIOB_BASE )

RCC -> AHB1ENR |= 0x02;

GPIOB -> MODER |= 0x00100000;
GPIOB -> PUPDR |= 0x00200000;
GPIOB -> OSPEED |= 0x00200000;
GPIOB -> ODR |= 0x0400;
```

Define a Data Structure for Peripherals

- With such arrangement:
  - The same register data structure for the peripheral can be shared between multiple instantiations
  - Hence code maintenance is easier
  - The requirement for immediate data storage is reduced
  - Compiled code is smaller, giving better code density

- With further modification, the functions developed for one peripheral can be shared between multiple instantiations by passing the base pointer to the function, for example:

```
#define GPIOA                    ((GPIO_TypeDef *) GPIOA_BASE )
#define GPIOB                    ((GPIO_TypeDef *) GPIOB_BASE )
#define GPIOC                    ((GPIO_TypeDef *) GPIOC_BASE )

void GPIO_init (GPIO_TypeDef *GPIO_pointer) {
    GPIOB -> MODER |= 0x00100000;
    GPIOA -> MODER |= 0x00010000;
}
```

Mbed API – Digital I/O

- The Mbed API provides a number of drivers that provide access to general purpose microcontroller hardware

- For digital I/O the key APIs are:
  - DigitalIn
  - DigitalOut
  - BusIn
  - BusOut

- We will take a look at some examples and see how they compare to the low-level equivalent

DigitalIn/DigitalOut

- The DigitalIn interface is used to read the value of a digital input pin
  - The logic level is either 1 or 0

- The DigitalOut interface is used to configure and control a digital output pin by setting the pin to a logic level of 0 or 1.

- Any number of Arm Mbed pins can be used as a DigitalIn or DigitalOut, for example:

```
DigitalIn mybutton (Input Pin);
DigitalOut Led_out(Output Pin);

int main() {
        if (mybutton)
                Led_out = 1;
}
```

- There is also the DigitalInOut interface, which is a bidirectional digital pin, we can use this interface to read the value of a digital pin when set as an input(), as well as write the value when set as an output().

arm

DigitalIn Class Reference

Function Name	Description
DigitalIn(PinName pin)	Create a DigitalIn connected to the specified pin.
DigitalIn(PinName pin, PinMode mode)	Create a DigitalIn connected to the specified pin.
int read ()	Read the input, represented as 0 or 1 (int).
void mode (PinMode pull)	Set the input mode.
int is_connected ()	Return the output setting, represented as 0 or 1 (int).
operator int ()	An operator shorthand for read().

DigitalOut Class Reference

Function Name	Description
DigitalOut(PinName pin)	Create a DigitalOut connected to the specified pin.
DigitalOut(PinName pin, int value)	Create a DigitalOut connected to the specified pin.
void write ()	Set the output, specified as 0 or 1 (int).
int read ()	Return the output setting, represented as 0 or 1 (int).
int is_connected ()	Return the output setting, represented as 0 or 1 (int).
DigitalOut & operator= (int value)	A shorthand for write().
DigitalOut & operator= (DigitalOut &rhs)	A shorthand for write() using the assignment operator which copies the state from the DigitalOut argument.
operator int ()	An operator shorthand for read().

BusIn/BusOut

- The BusIn API allows you to combine a number of DigitalIn pins to read them at once
  - Useful for checking multiple inputs together as a single interface instead of individual pins
- The BusOut API allows you to combine a number of DigitalOut pins to write to them at once
  - Useful for writing to multiple pins together as a single interface instead of individual pins.
- Any number of Arm Mbed pins can be used as a BusIn or BusOut, for example:

```
#include "mbed.h"

BusOut myleds(LED1, LED2, LED3, LED4);

int main() {
    while(1) {
        for(int i=0; i<16; i++) {
            myleds = i;
            wait(0.25);
        }
    }
}
```

https://os.mbed.com/docs/mbed-os/v5.15/apis/busout.html

BusIn Class Reference

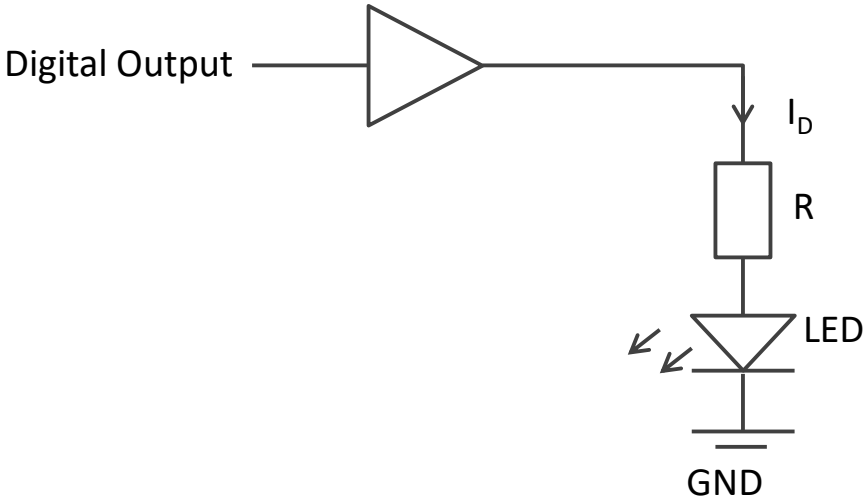
Function Name	Description
BusIn (PinName p0, PinName p1=NC … PinName p15=NC)	Create a BusIn, connected to the specified pins.
BusIn (PinName pins[16])	Create a BusIn, connected to the specified pins.
int read ()	Read the value of the input bus.
void mode (PinMode pull)	Set the input pin mode.
int mask ()	Binary mask of bus pins connected to actual pins (not NC pins). If bus pin is in NC state make corresponding bit will be cleared (set to 0), else bit will be set to 1.
operator int ()	A shorthand for read().
DigitalIn& operator[] (int index)	Access to a particular bit in random-iterator fashion.

BusOut Class Reference

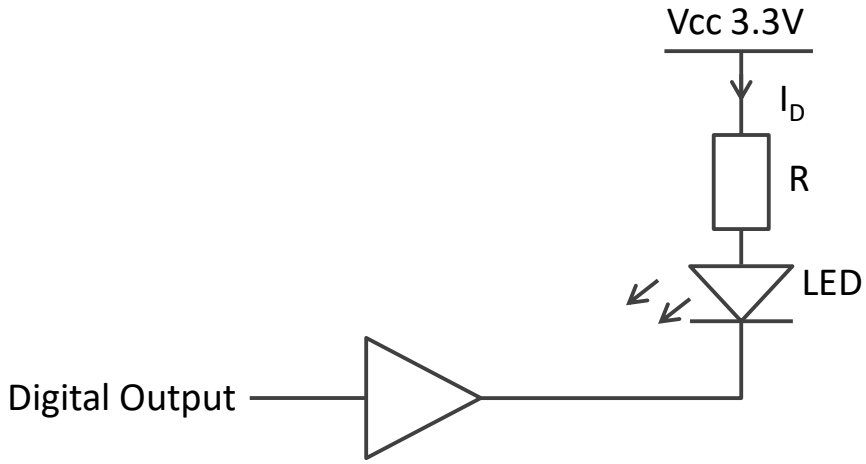
Function Name	Description
BusOut (PinName p0, PinName p1=NC … PinName p15=NC)	Create a BusOut, connected to the specified pins.
BusOut (PinName pins[16])	Create a BusOut, connected to the specified pins.
void write (int value)	Write the value to the output bus.
int read ()	Read the value currently output on the bus.
void mode (PinMode pull)	Set the input pin mode.
int mask ()	Binary mask of bus pins connected to actual pins (not NC pins). If bus pin is in NC state make corresponding bit will be cleared (set to 0), else bit will be set to 1.
BusOut & operator= (int v)	A shorthand for write().
operator int ()	A shorthand for read().
DigitalIn& operator[] (int index)	Access to a particular bit in random-iterator fashion.

Digital IO Example: LEDs

- Light-emitting Diode (LED)
  - Emits light when switched on
  - Is the simplest way to indicate the status of logic
  - Is also widely used in applications such as automotive lighting, general lighting, traffic signals etc



Digital output sources current to LED



Digital output sinks current from LED

arm

Digital IO Example: 7-Segment Display

- Use 7 segments and a dot to display numerals or letters
- Widely used in digital electronic devices, such as digital clocks and electronic meters
- Simple to controller, easy to debug
- Different values can be represented by different combinations of 8-bit segments (including dot). For example:

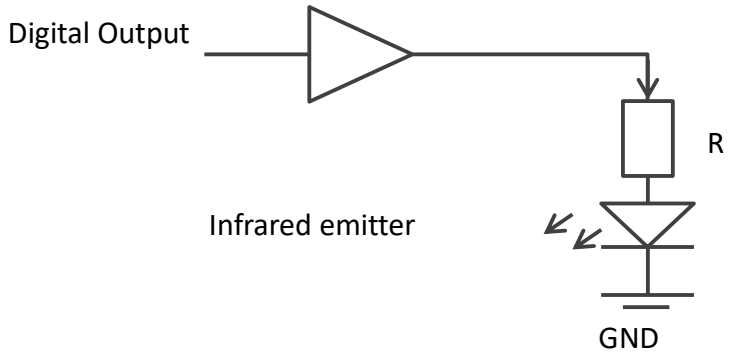
Display	0	1	2	3	4
8-bit value	00111111	00000110	01011011	01001111	01100110
Display	5	6	7	8	9
8-bit value	01101101	01111101	00000111	01111111	01101111

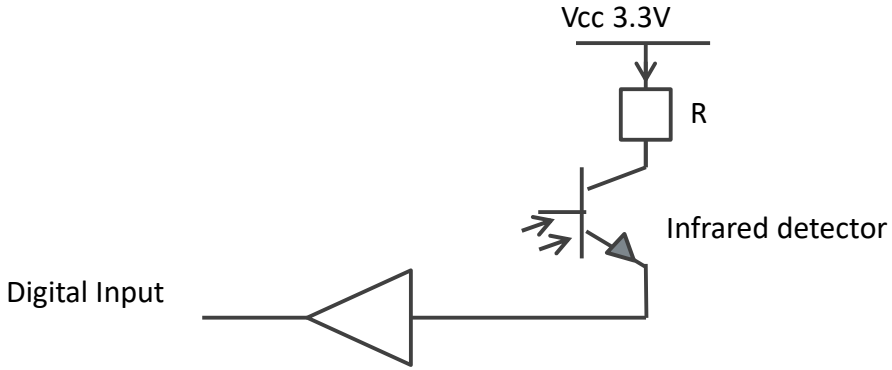
arm

Digital IO Example: Infrared Emitter/Detector

- Infrared emitter (LED)
  - A light-emitting diode that emits invisible infrared (IR) when conducting e.g. when connected to digital output:
    - '1': IR emitted (or brightness above a threshold)
    - '0': no IR emitted (or brightness below a threshold)

- Infrared detector (photodiode)
  - Converts light into either current or voltage e.g. when connected to digital input:
    - '1': IR received (or brightness above a threshold)
    - '0': no IR received (or brightness below a threshold)





Resources

- Cortex-M4 Technical Reference Manual: https://developer.arm.com/docs/100166/0001
- Cortex-M4 Devices Generic User Guide: https://developer.arm.com/docs/dui0553/b
- STM32 Nucleo-F401RE: https://www.st.com/en/evaluation-tools/nucleo-f401re.html

arm

Introduction to the Mbed Platform and CMSIS

Module Syllabus

- Mbed Overview
- Mbed and the Internet of Things
- Overview of Software Libraries
- Mbed OS
- Mbed Software Development Kit
- Mbed Hardware Development Kit (HDK)
- Cortex Microcontroller Software Interface Standard (CMSIS)

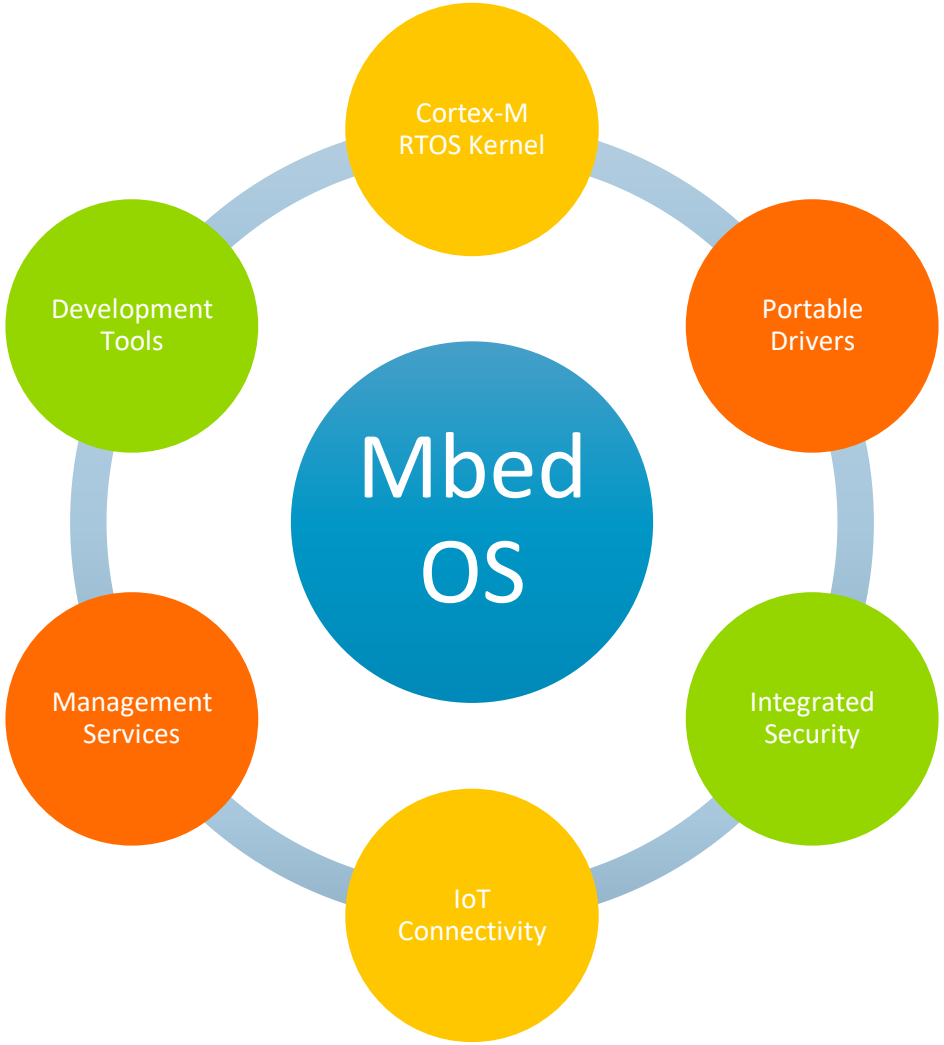
Introduction to Mbed

- What is Mbed?
  - A platform used for the easy prototyping and development of applications and systems based on Arm Cortex-M-based microcontrollers, typically for use in the world of the Internet of Things (IoT)

- The Mbed platform provides:
  - Open software libraries
  - Open hardware designs
  - Open online tools for professional rapid prototyping of products based on Arm-based microcontrollers

- The Mbed platform includes:
  - Mbed Operating System (Mbed OS)
    - Libraries, RTOS core, HAL, API, and more
  - A microcontroller Hardware Development Kit (HDK) and supported development boards
  - Integrated Development Environment (IDE), including an online compiler and online developer collaboration tools

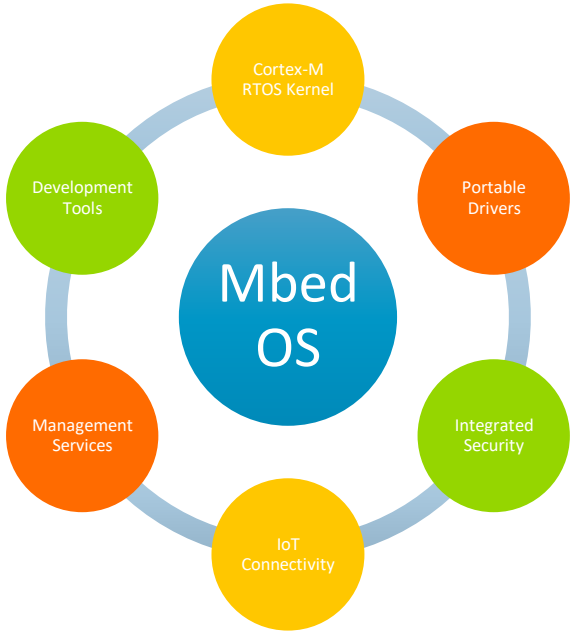
Mbed OS



arm

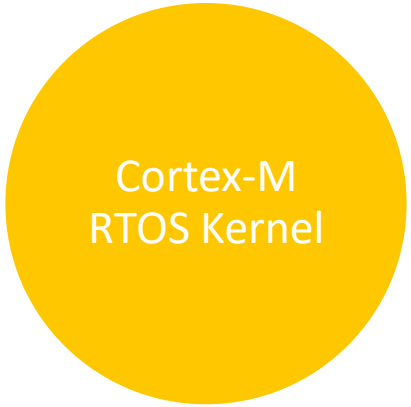
Mbed OS - Overview

- An open-source operating system for platforms using Arm microcontrollers, specifically designed for devices involved in the Internet of Things (IoT)
- Offers a variety of features to enable the development of IoT connected systems
- Provides a layer of abstraction that interprets the application code in way the hardware can understand
- This enables the developer to focus on programming applications that can run on a range of devices
  - An Mbed OS application can be run on any Mbed-compatible platform



Mbed OS – Cortex-M RTOS Kernel

- Mbed has an RTOS core
  - Supports deterministic, multithreaded, real-time software execution
- RTOS primitives are available to allow drivers and applications to rely on threads, semaphores, mutexes and other RTOS features



arm

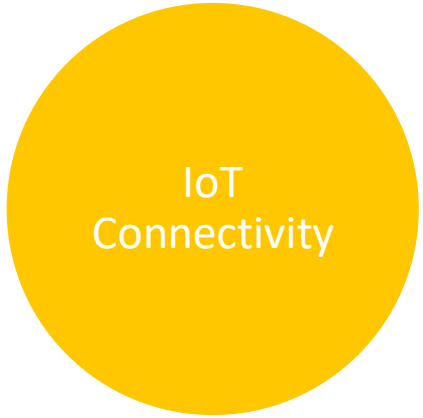
Mbed OS - Security

- Mbed provides two security-focused embedded building blocks:
  - Arm Mbed TLS
  - Secure Partition Manager (SPM)
- Mbed TLS is a protocol for securing communication channels between devices and servers or gateways
- The secure partition manager is responsible for:
  - Isolating software within partitions
  - Managing the execution of software within partitions
  - Providing Inter-Process Communication (IPC) between partitions

arm

Mbed OS - Connectivity

- Mbed OS supports a number of connectivity protocols
  - Paired with Pelion Device Management to provide full support for a range of communication options
- Such technologies include:
  - NarrowBand-IoT (NB-IoT):
  - Bluetooth Low Energy (BLE)
  - 6LoWPAN
  - Thread



arm

Mbed Software Development Kit

- Mbed Software Development Kit (SDK) includes:
  - Software libraries
    - Official C/C++ software libraries
      - Start-up code, peripheral drivers, networking, RTOS and runtime environment
    - Community-developed libraries and codes
      - Cookbook of hundreds of reusable peripheral and module libraries have been built on top of the SDK, which can be used to build your projects faster
  - Software tools, such as build tools, test and debug scripts

- Enables rapid prototyping of embedded applications

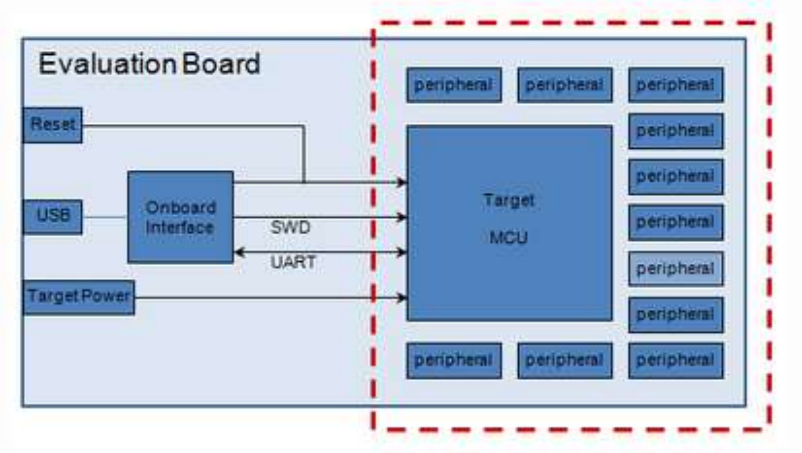
Mbed HDK

- The Mbed Hardware Development Kit (HDK) consists of different hardware design resource that assist in the development of custom hardware
  - Allows utilisation of the Arm Mbed ecosystem and associated technologies such as Mbed OS and DAPLink
- Working with boards based on the Mbed HDK allows for an efficient start to working with the Mbed platform
- The HDK offers:
  - Eagle schematics and board files
  - PDF schematics and board copies
  - CAM Job GERBERS for manufacture (including pick/place and drill)
  - Bill of Materials (BOM)
  - A repository of content and projects

Source: https://os.mbed.com/docs/mbed-os/v5.9/reference/arm-mbed-hdk.html

Benefits of the Mbed HDK

- Quick design short-cut
  - Using ready-made schematics
- Easy-to-use USB and debugging support
- Compatible with Mbed OS



An example of how a microcontroller sub-system might be used to build an evaluation board

DAPLink

- Open Source project that implements the embedded firmware required for a Cortex debug probe
  - Complemented by numerous reference designs that enable creation of DAPLink debug probe hardware
- The DAPLink debug probe is connected to the host computer via USB and connects to the target system through a standard Cortex debug connecter
- Features – provided via USB connection:
  - **HID interface**
    - Provides a channel over which CMSIS-DAP protocols can run
      - Enables tools such as Keil MDK, IAR Workbench, and pyOCD
  - **USB drag and drop programming**
    - The DAPLink debug probes appear as a USB disk on the host computer
    - Binary and hex files can be copied to the USB disk which are then programmed into the memory of the target system
  - **USB serial port**
    - DAPLink probe also provides a USB serial port
      - Port will appear on a Windows as a COM port or Linux as a /dev/tty interface

Development Tools

There are multiple ways for developers to program with Mbed. These consist of:

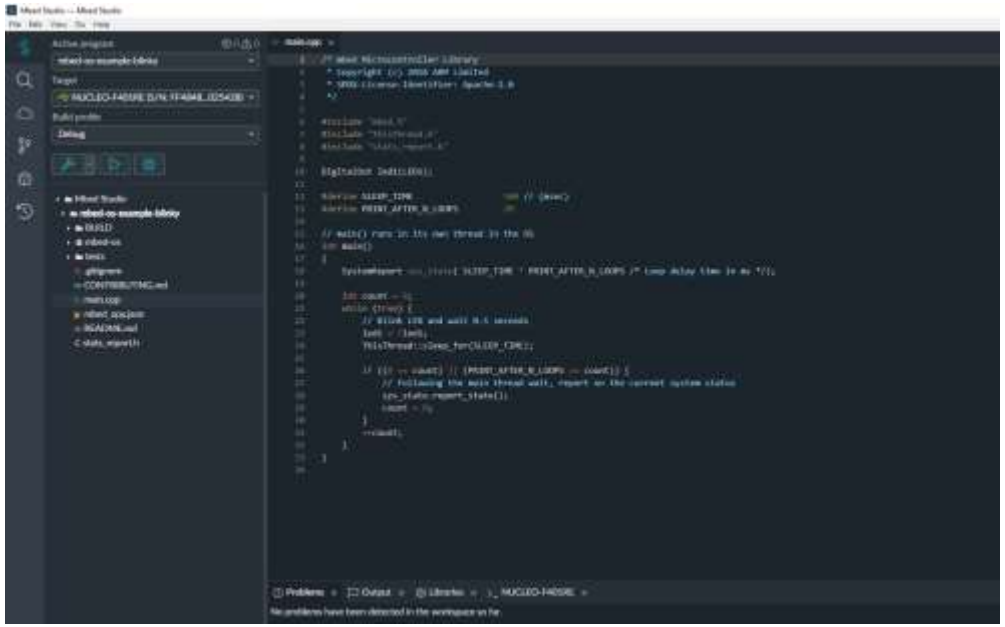
Mbed Studio

Mbed Online Compiler

Mbed CLI

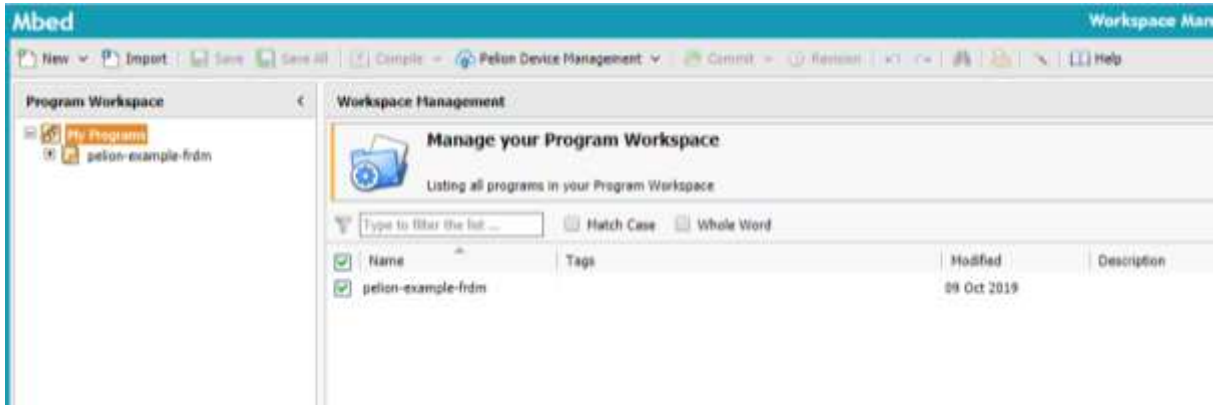
Mbed Studio

- Integrated development environment (IDE) for Mbed OS 5 applications
  - Includes everything required to create, compile and debug Mbed programs
  - Automatically detects connected Mbed enabled boards
    - Quick development for specific targets
  - Flashes code directly to connected platform
  - Provides debug session for debugging and profiling the target board



Mbed Online Compiler

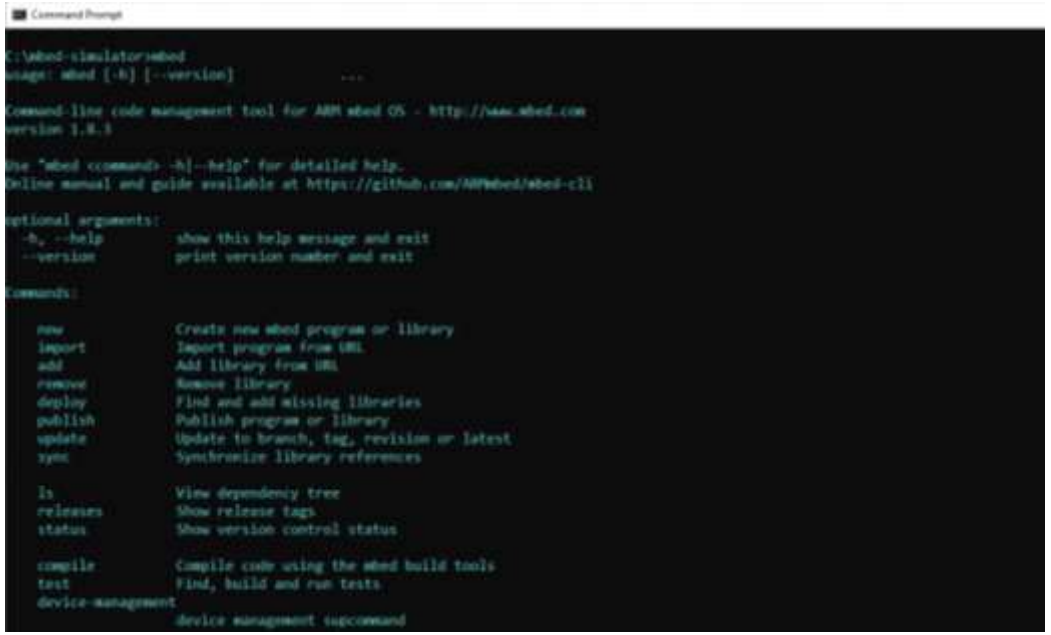
- A lightweight online C/C++ IDE that enables quick writing of programs, compilation, and running on a microcontroller
- No prior-installation or set-up required to work with Mbed
- Includes full code editor, version control, and library management



Mbed CLI

- Arm Mbed CLI is a command-line tool packaged as *'mbed-cli'* and based on Python.

- Enables Git and Mercurial-based version control, along with dependency management, code publishing, support for remotely hosted repositories, and use of the Arm Mbed OS build system.



Reference: https://os.mbed.com/docs/mbed-os/v5.14/tools/developing-mbed-cli.html

Testing with Mbed

- The Mbed platform offers a number of tools that support testing of your Mbed code

- Greentea
  - Automated testing tool for Arm Mbed OS development
  - Pair with 'UNITY' and 'utest' frameworks
  - https://os.mbed.com/docs/mbed-os/v5.15/tools/greentea-testing-applications.html

- Icetea
  - Automated testing tool for Arm Mbed OS development
  - Typically used for local development and automation in a continuous integration environment
  - https://os.mbed.com/docs/mbed-os/v5.15/tools/icetea-testing-applications.html

- Process of flashing boards, running the tests, and generating reports is automated by the test system

Mbed Enabled Platforms

- The Arm® Mbed Enabled™ program outlines a set of functionality and requirements that must be met in order to become “Mbed Enabled”. This can cover development boards, modules, components, and interfaces
  - This benefits developers as they are assured that the platforms they choose to work with can perform certain functions/provide certain performance
  - It is also beneficial to the vendors as it allows their products more exposure when certified, and enables their product to become more familiar with developers in the Mbed eco-system
  - Some example boards include:

NUCLEO-F401RE

DISCO-F413ZH

DISCO-F413ZH

Nordic nRF51-DK

High-level vs. Low-level Programming

High-level

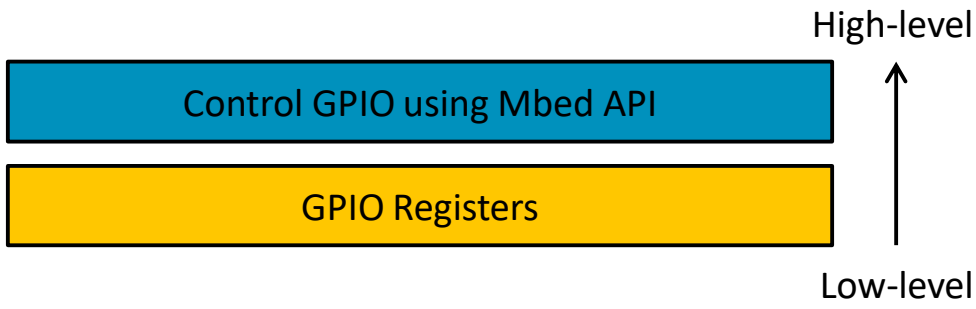
- **Advantages:**
  - Higher productivity (less development time)
  - Portability across devices
  - Resulting code that is easier code to read and maintain
  - Allows reuse of code
  - Rapid prototyping of applications
- **Disadvantages:**
  - Less optimized code
  - Additional translation time for source to machine code
  - Another level of abstraction to deal with

Low-level

- **Advantages:**
  - More optimized code and memory efficient
  - Less translation time for source to machine code
  - Directly talk to hardware
- **Disadvantages:**
  - Less portability from one device to another
  - Resulting code is more difficult for others to read, reuse, and maintain
  - Low productivity

Comparison: High-level vs Low-level

- MCU register layer
  - Blinky example by poking registers
- Mbed API
  - Blinky example using Mbed API functions



MCU Register Example

- As shown in previous modules, the GPIO peripheral can be directly accessed by writing/reading specific memory addresses:
  - Assign a pointer to the address of each register
  - Registers can be read/ written to using the pointer

```
1  #include "mbed.h"
2
3  // Reuse initialization code from the mbed library
4  DigitalOut led1(LED1); // P1_18
5
6  int main() {
7      unsigned int mask_pin18 = 1 << 18;
8
9      volatile unsigned int *port1_set = (unsigned int *)0x2009C038;
10     volatile unsigned int *port1_clr = (unsigned int *)0x2009C03C;
11
12     while (true) {
13         *port1_set |= mask_pin18;
14         wait(0.5);
15
16         *port1_clr |= mask_pin18;
17         wait(0.5);
18     }
19 }
```

Directly access MCU registers

Mbed API Example

- The Mbed API provides the actual user-friendly object-oriented API to the final user
  - More friendly functions/APIs
  - Object oriented API (using C++)
  - Top-level API used by the majority of the programs developed on the mbed platform
  - Defines basic operators to provide intuitive casting to primitive types and assignments
  - A digital IO class is defined as shown in the code clip.

```
class DigitalInOut {

public:
    DigitalInOut(PinName pin) {
        gpio_init(&gpio, pin, PIN_INPUT);
    }

    void write(int value) {
        gpio_write(&gpio, value);
    }

    int read() {
        return gpio_read(&gpio);
    }

    void output() {
        gpio_dir(&gpio, PIN_OUTPUT);
    }

    void input() {
        gpio_dir(&gpio, PIN_INPUT);
    }

    void mode(PinMode pull) {
        gpio_mode(&gpio, pull);
    }

    DigitalInOut& operator= (int value) {
        write(value);
        return *this;
    }

    DigitalInOut& operator= (DigitalInOut& rhs) {
        write(rhs.read());
        return *this;
    }
```

Mbed API Example

- With the support of the Mbed API, the same blinky example can be programmed in a much simpler and more intuitive way:

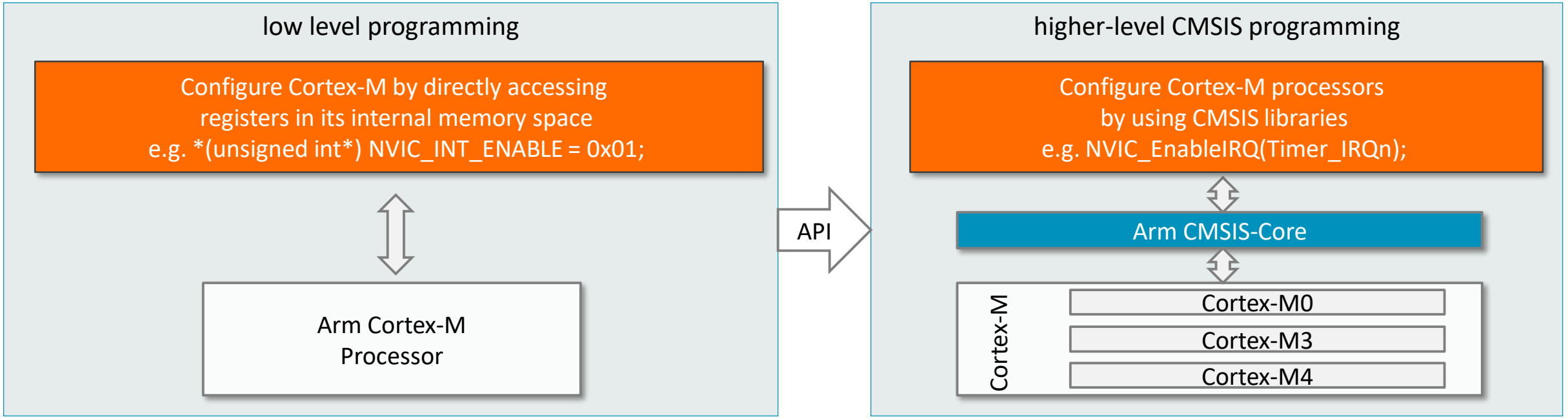
- Note that the Mbed API is programmed using the object-oriented language C++, which originated from C language with object-oriented features such as classes.

- Deep knowledge of C++ is not necessary to use the Mbed API. However, many tutorials and books can help you learn C++ if you wish.

arm

Cortex Microcontroller Software Interface Standard (CMSIS)

- CMSIS: Cortex Microcontroller Software Interface Standard
  - A vendor-independent hardware abstraction layer for the Cortex-M processor series
  - Provides a standardized software interface, such as library functions, which help control the processor more easily, e.g. configuring the Nested Vectored Interrupt Controller (NVIC)
  - Improves software portability across different Cortex-M processors and Cortex-M based microcontrollers



What is Standardized by CMSIS?

- Functions to access, e.g., NVIC, System Control Block (SCB) and System Tick timer
  - Enables an interrupt or exception: NVIC_EnableIRQ (IRQn_Type IRQn)
  - Sets pending status of interrupt: void NVIC_SetPendingIRQ (IRQn_Type IRQn)

- Access to special registers e.g.
  - Read PRIMASK register: uint32_t __get_PRIMASK (void)
  - Set CONTROL register: void __set_CONTROL (uint32_t value)

- Functions to access special instructions e.g.
  - REV: uint32_t __REV(uint32_t int value)
  - NOP: void __NOP(void)

- Names of system initialization functions e.g.
  - System initialization: void SystemInit (void)

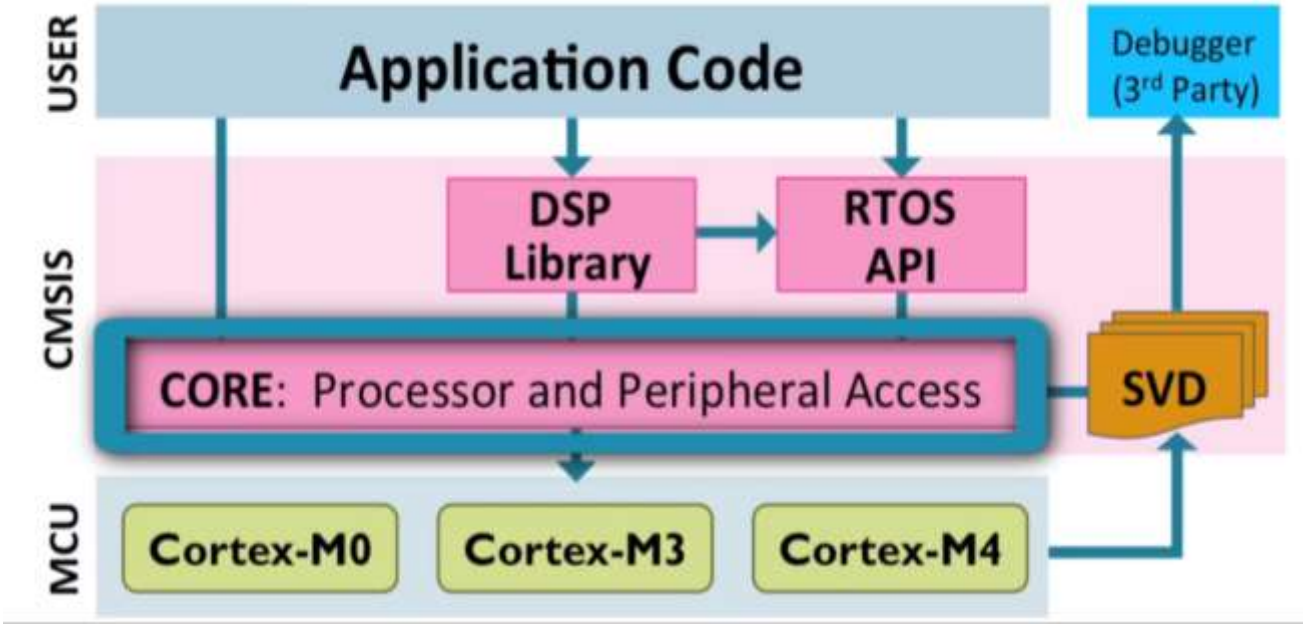
Benefits of CMSIS

- Easier to port application code from one Cortex-M based microcontroller to another Cortex-M based microcontroller
- Easier to reuse the same code between different Cortex-M based microcontrollers
- Better compatibility when integrating third-party software components, since all third-party components such as applications, embedded OS, middleware etc., can share the same standard CMSIS interface
- Better code density and smaller memory footprint, since the codes in CMSIS have been optimized and tested

arm

CMSIS Components

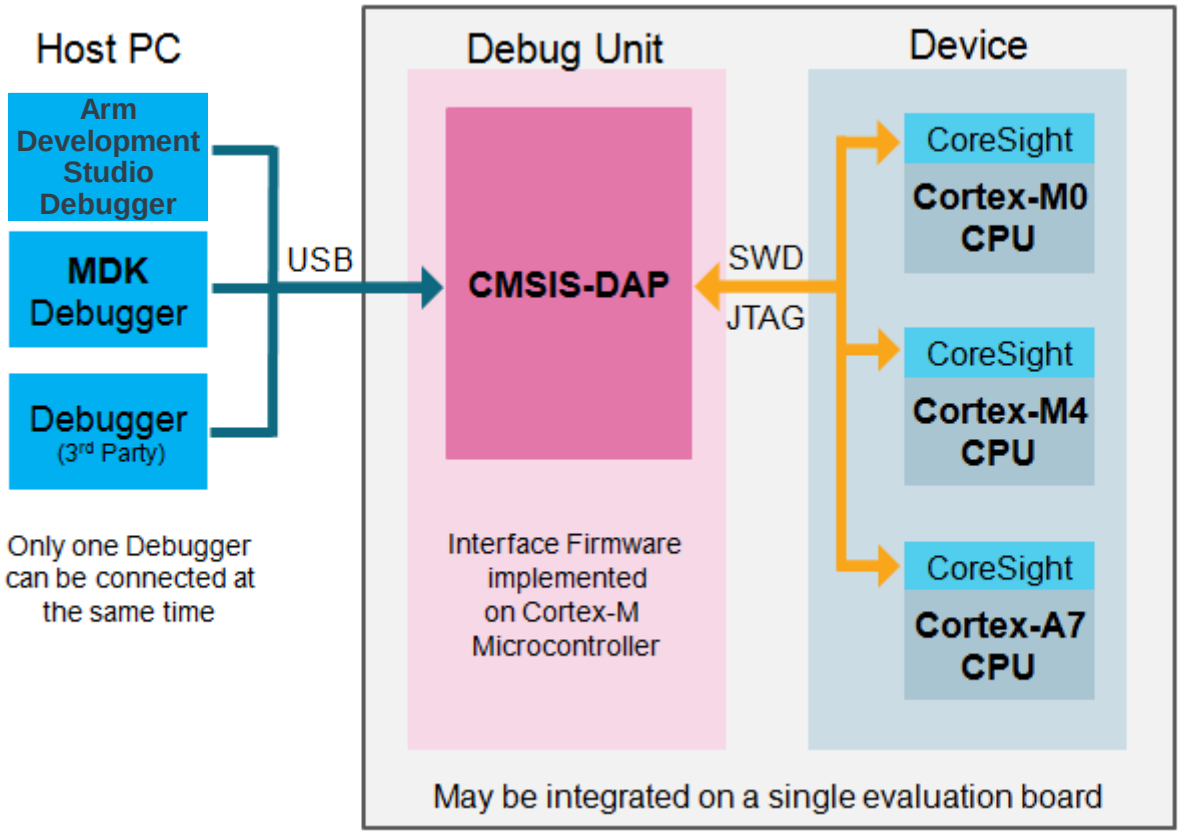
- The CMSIS consists of the following components:
  - CMSIS-CORE
  - CMSIS-DSP, CMSIS-RTOS API and CMSIS-SVD
  - In this module, we will focus on using CMSIS-CORE



arm

CMSIS-DAP

- CMSIS-DAP is an interface firmware for the debug unit that connects the debug port to USB
- Debuggers can connect via USB to the debug unit and to the device running the application software
- The debug unit is connected to the target device via JTAG or SW
- The Arm-Cortex processors then provide the CoreSight Debug and Trace Unit



CMSIS Functions: Access NVIC

CMSIS Function	Description
void NVIC_EnableIRQ (IRQn_Type IRQn)	Enables an interrupt or exception.
void NVIC_DisableIRQ (IRQn_Type IRQn)	Disables an interrupt or exception.
void NVIC_SetPendingIRQ (IRQn_Type IRQn)	Sets the pending status of interrupt or exception to 1.
void NVIC_ClearPendingIRQ (IRQn_Type IRQn)	Clears the pending status of interrupt or exception to 0.
uint32_t NVIC_GetPendingIRQ (IRQn_Type IRQn)	Reads the pending status of interrupt or exception. This function returns non-zero value if the pending status is set to 1.
void NVIC_SetPriority (IRQn_Type IRQn, uint32_t priority)	Sets the priority of an interrupt or exception with configurable priority level to 1.
uint32_t NVIC_GetPriority (IRQn_Type IRQn)	Reads the priority of an interrupt or exception with configurable priority level. This function return the current priority level.

CMSIS Functions: Access Special Registers

Special Register	Access	CMSIS Function
PRIMASK	Read	uint32_t __get_PRIMASK (void)
	Write	void __set_PRIMASK (uint32_t value)
CONTROL	Read	uint32_t __get_CONTROL (void)
	Write	void __set_CONTROL (uint32_t value)
MSP	Read	uint32_t __get_MSP (void)
	Write	void __set_MSP (uint32_t TopOfMainStack)
PSP	Read	uint32_t __get_PSP (void)
	Write	void __set_PSP (uint32_t TopOfProcStack)

CMSIS Functions: Execute Special Instructions

Instruction	CMSIS Intrinsic Function
CPSIE i	void __enable_irq(void)
CPSID i	void __disable_irq(void)
ISB	void __ISB(void)
DSB	void __DSB(void)
DMB	void __DMB(void)
NOP	void __NOP(void)
REV	uint32_t __REV(uint32_t int value)
REV16	uint32_t __REV16(uint32_t int value)
REVSH	uint32_t __REVSH(uint32_t int value)
SEV	void __SEV(void)
WFE	void __WFE(void)
WFI	void __WFI(void)

CMSIS Functions: Access System

CMSIS Function	Description
void NVIC_SystemReset(void)	Initiate a system reset request
uint32_t SysTick_Config(uint32_t ticks)	Initialize and start the SysTick counter and its interrupt
void SystemInit (void)	Initialize the system
void SystemCoreClockUpdate(void)	Update the SystemCoreClock variable

arm

Analog Input and Output

Module Syllabus

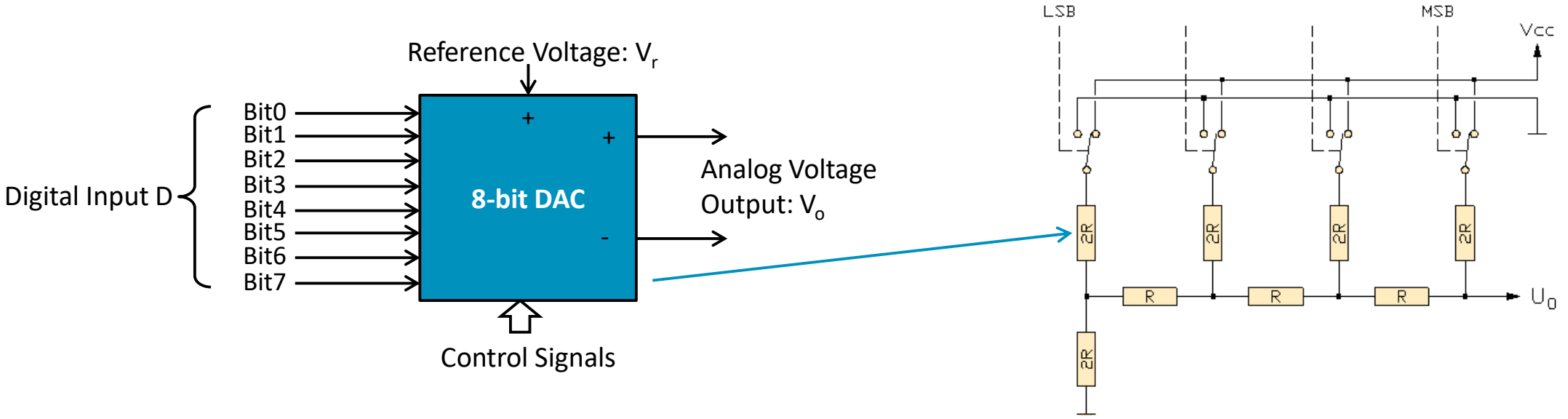
- Analog input and output overview
- Analog input
  - Digital-to-analog converter
- Analog output
  - Analog-to-digital converter
  - ADC range
  - Resolution and quantization
  - Sampling frequency
- Input/output analog signals using Mbed
  - Mbed analog input
  - Mbed analog output

Overview of Analog Input and Output

- Most signals in the real world are analog signals
- Processors are digital devices. To interface with analog signals, these have to be converted into digital signals and vice versa
- Analog-to-Digital Converter (ADC):
  - Converts the amplitude of physical analog signals to discrete digital numbers
  - Involves signal sampling and quantization
- Digital-to-Analog Converter (DAC):
  - An inverse operation of ADC
  - Converts digital data into analog signals, such as current, voltage, or electric charge
  - For example, a DAC could be used to drive an earphone or speaker amplifier so that sound (analog air pressure waves) can be produced.

Digital-to-Analog Converter

- Digital-to-Analog Converter (DAC):
  - Input: discrete finite-precision number (usually in binary)
  - Output: continually varying physical signals, such as voltage or pressure
  - The following picture gives an 8-bit DAC example:



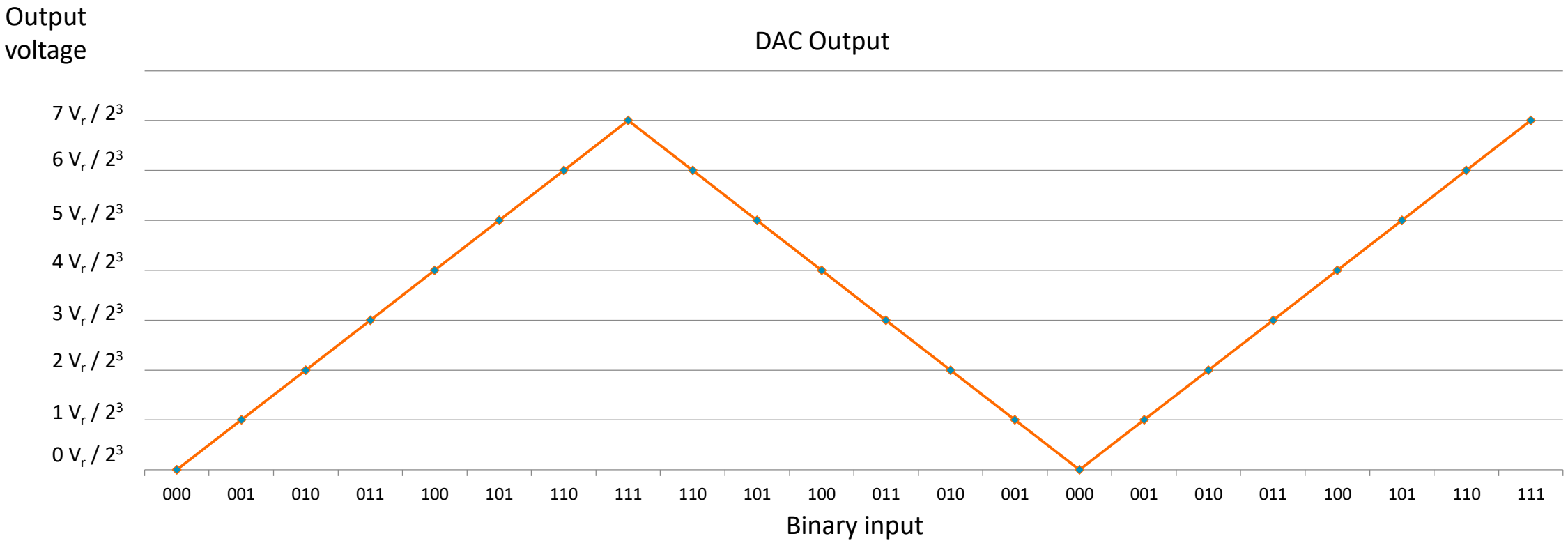
source: http://www.vias.org/mikroelektronik/da_converter.html

Digital-to-Analog Converter

- The relationship between a DAC's input and its output is: $V_0 = \frac{D}{2^n} V_r$
  - Vo: Output voltage
  - Vr: Value of the reference voltage
  - D: Value of the input binary word
  - n: the number of bits in the word
  - Maximum output: $V_{o,\max} = \frac{2^n - 1}{2^n} V_r = \left(1 - \frac{1}{2^n}\right) V_r$
    - Vr is never reached in this case since: $D_{\max} = 2^n - 1$

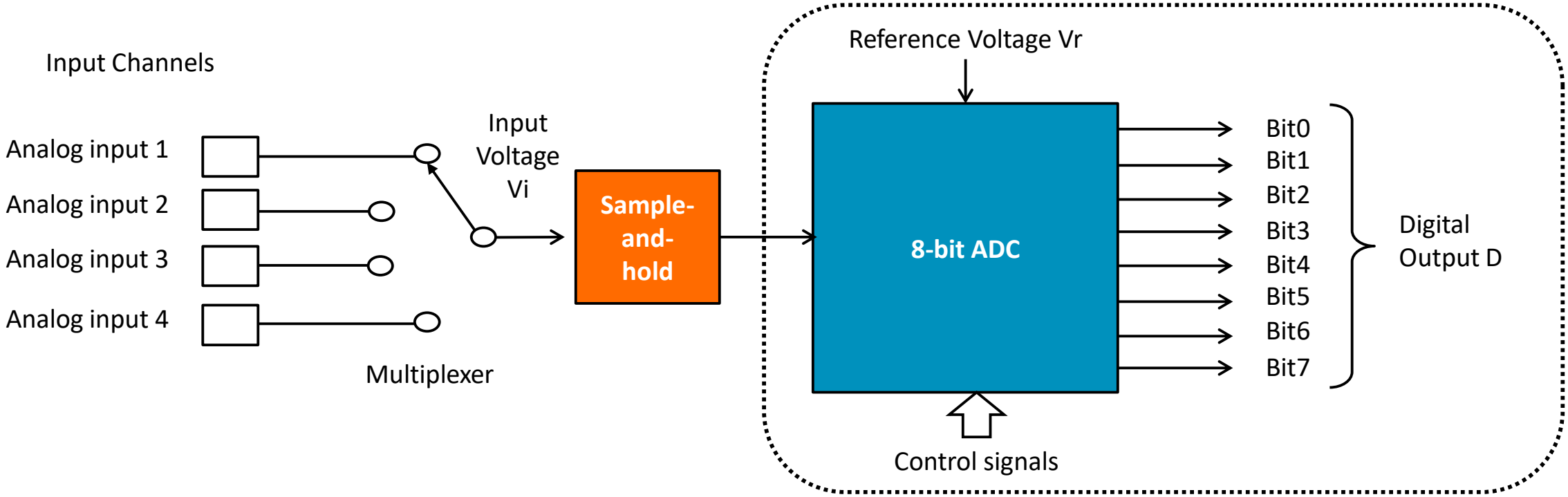
Digital-to-Analog Converter

- For example, to generate a triangular wave:
  - Input: 3-bit binary data
  - Output: analog voltage ranging from 0 to 7/8 V_r



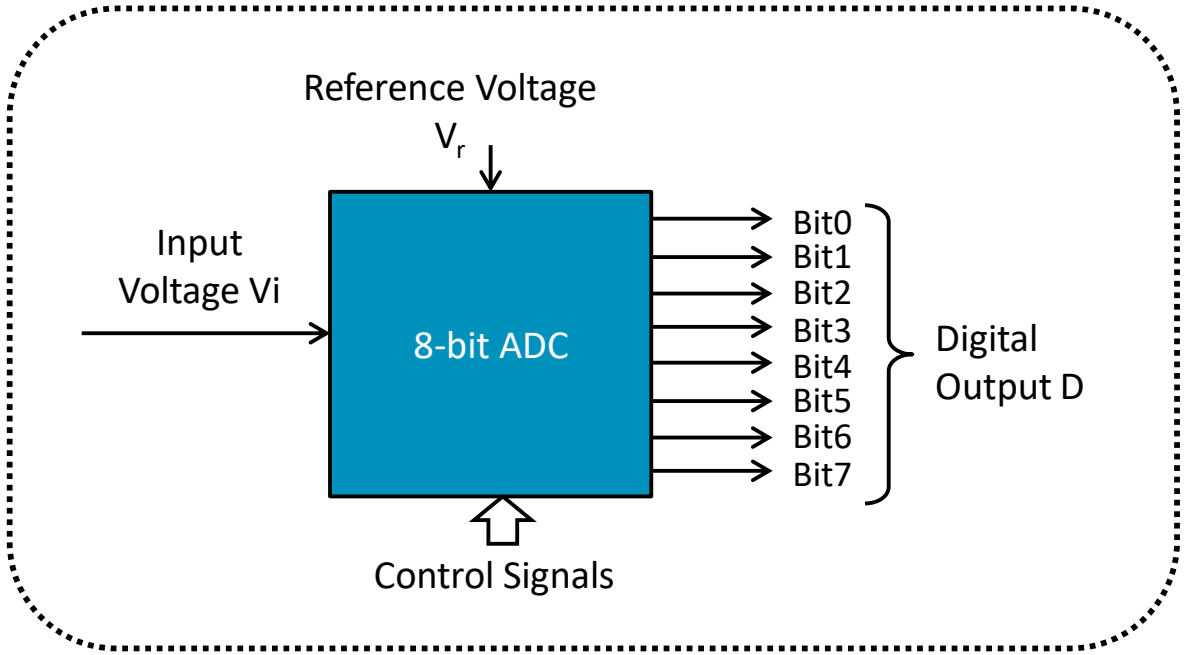
Analog-Digital-Conversion-System

- An ADC-System usually consists of three stages:
  - Multiplexer: reduces the number of required ADC components at the expense of conversion speed
  - Sample-and-hold: keeps the analog signal during the conversion constant. Time discretization!
  - ADC: converts the analog into digital signal



Analog-to-Digital Converter

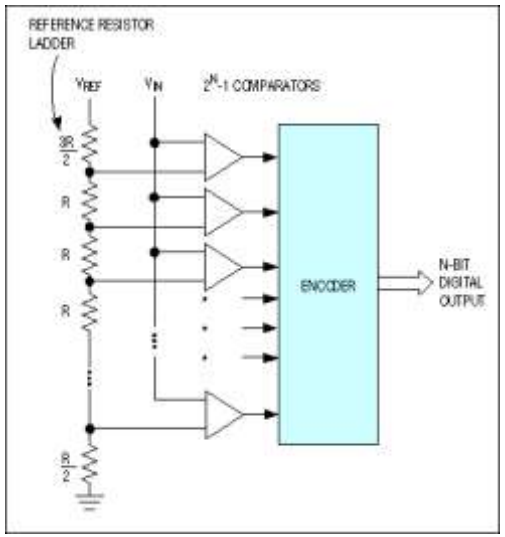
- Analog-to-Digital Converter (ADC):
  - Input: time-discrete due to the prior S+H-stage physical signals
  - Output: discrete numbers that are proportional to the analog input amplitude
  - The following picture gives an example of an 8-bit ADC:



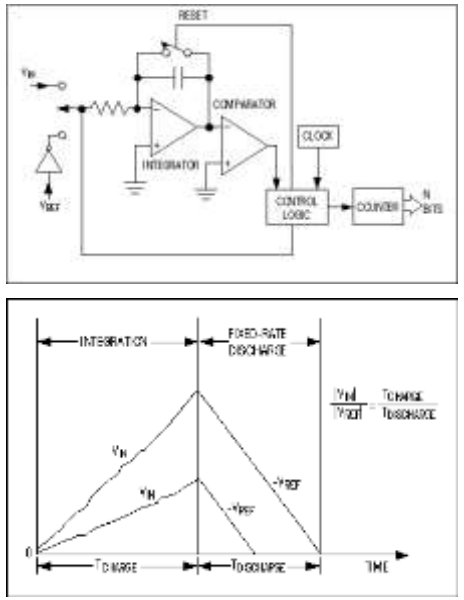
arm

ADC Types

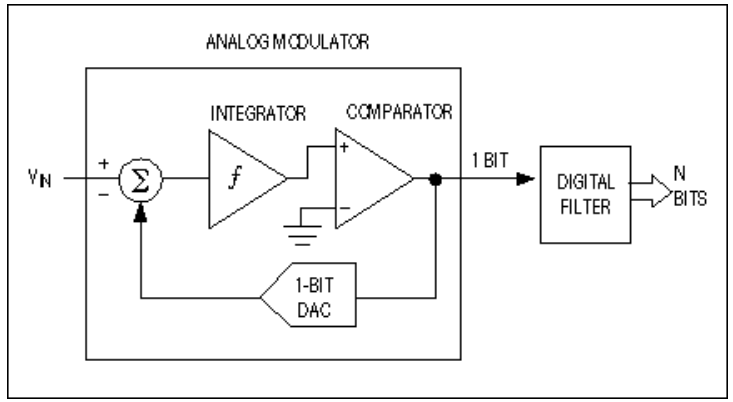
- There are 3 common types for ADC:
  - Flash converters, comparing the input voltage with different reference voltage levels
  - Slope converters, converting the voltage into a charge capacitor and measuring the time for charging and discharging
  - Feedback converters, trying to generate a signal proportional to the input voltage. The internal proportionality-coefficient represents the digital word.



flash converter



slope converter



feedback converter
here: delta-sigma

source:
http://www.skillbank.co.uk/SignalConversion/adc.htm

ADC Range

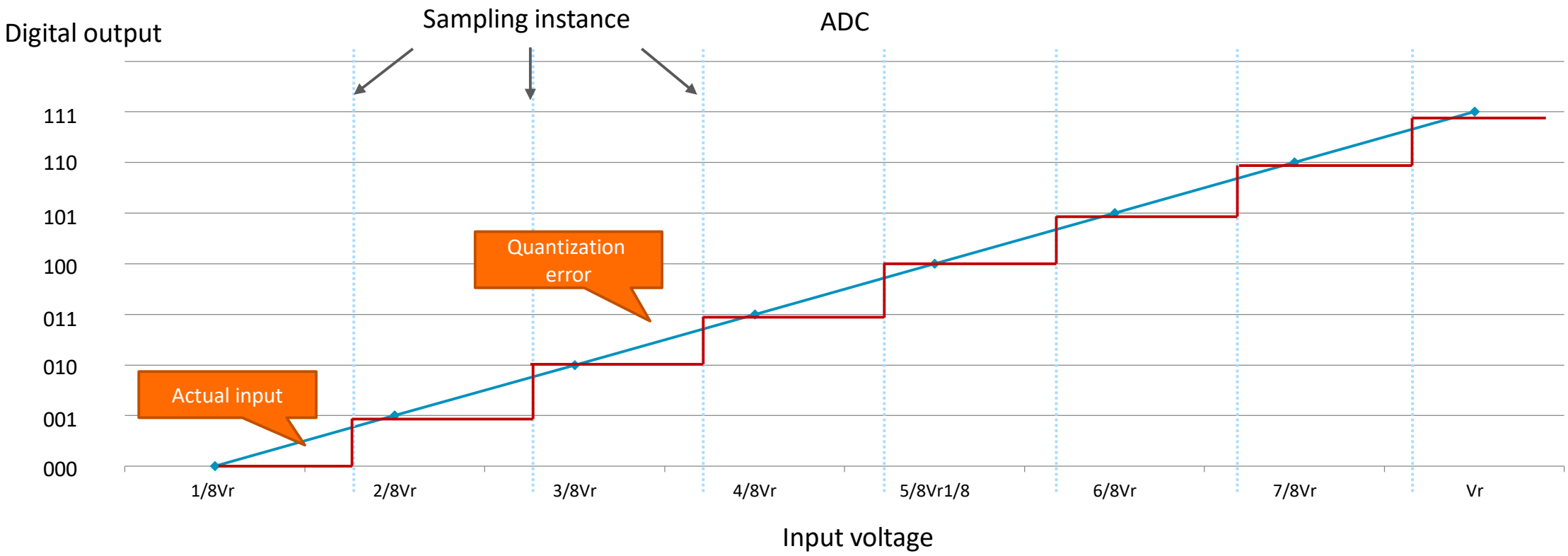
- ADCs have minimum and maximum permissible input values, which determine the range of an ADC.
- The range and the resolution of the ADC are linked to the reference voltage and the physical principle of the ADC.
  - Flash converter and feedback converter: $D = \mathrm{int}\left(\frac{V_i}{V_{ref}} 2^n \right)$
  - Slope converter: $D = \mathrm{int}\left(\frac{t_{\mathrm{charge}} V_i}{t_{clock} V_{ref}} \right)$
  - V_i: input voltage; V_{ref}: reference voltage; n: digital resolution in bit; t_{charge}: charging time; t_{clock}: reference clock

ADC Resolution and Quantization

- The resolution of the ADC refers to the number of discrete data that can be produced over the range of the analog values.
  - Resolution determines the minimum magnitude of the quantization and the quantization error.
  - Can be expressed in bits, e.g., an 8-bit ADC has a resolution of 256 values.
  - Can also be expressed in volts: the minimum change in the digital output is called the Least Significant Bit (LSB) voltage, e.g., an 8-bit ADC with a range of 10V has a resolution of 10/256V.
  - Resolution also determines the maximum possible average Signal to Noise Ratio (SNR).

Quantization Error

- Quantization error is the noise introduced by quantization in an ideal ADC.
  - Rounding error between the actual analog input and the digitized value

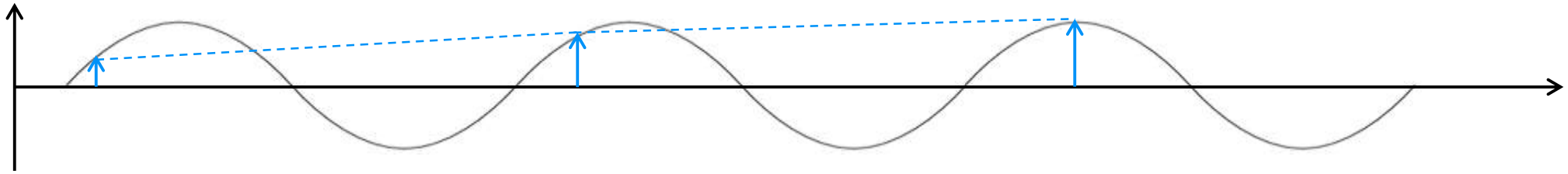


arm

ADC Sampling Frequency

- In ADC, the sampling rate refers to the rate of collecting new values
  - The analog signal is continuous in time
  - Digital data are is discrete
  - The sampling rate has to be high enough to reproduce the original signal

- According to the Shannon-Nyquist sampling theorem, the sampling frequency must be at least double that of the highest frequency in the incoming signal.
  - For example, the accepted standard range of audible frequencies is in the range of 20 to 20,000 Hz, approximately, so standard audio CDs are sampled and played at 44.1kHz.
  - Reduced sampling frequency will result in signal aliasing effect as shown below:



Input Analog Signal Using Mbed

- The Mbed API provides libraries for both ADC and DAC.
- For example, the following functions are provided to input an analog signal:

Function Name	Description
AnalogIn (PinName pin)	Create an AnalogIn connected to the specified pin
float read ()	Read the input voltage, represented as a float in the range [0.0, 1.0]
unsigned short read_u16 ()	Read the input voltage, represented as an unsigned short in the range [0x0, 0xFFFF]
operator float ()	An operator shorthand for read()

Example of Analog Input

- The following code gives an example of using analog input functions:

```
#include "mbed.h"

AnalogIn ain(Analog Input Pin);
DigitalOut led(Output Pin);

int main() {
    while (1){
        if(ain > 0.3) {
            led = 1;
        } else {
            led = 0;
        }
    }
}
```

Output Analog Signal Using Mbed

- Mbed API functions for analog output

Function Name	Description
AnalogOut (PinName pin)	Create an AnalogOut connected to the specified pin.
void write (float value)	Set the output voltage, specified as a percentage (float)
void write_u16 (unsigned short value)	Set the output voltage, represented as an unsigned short in the range [0x0, 0xFFFF]
float read ()	Return the current output voltage setting, measured as a percentage (float)
AnalogOut operator= (float percent)	An operator shorthand for write()
AnalogOut operator float ()	An operator shorthand for read()

Example of Analog Output

- The following code gives an example of using analog output functions:

```
#include "mbed.h"

AnalogOut signal(Analog Output Pin);

int main() {
    while(1) {
        for(float i=0.0; i<1.0; i+=0.1) {
            signal = i;
            wait(0.0001);
        }
    }
}
```

Resources

- Mbed: https://www.mbed.com/en/
- Official Mbed GitHub Repository: https://github.com/ARMmbed/mbed-os
- Mbed OS 5: https://os.mbed.com/docs/mbed-os/v5.14/introduction/index.html

arm

arm

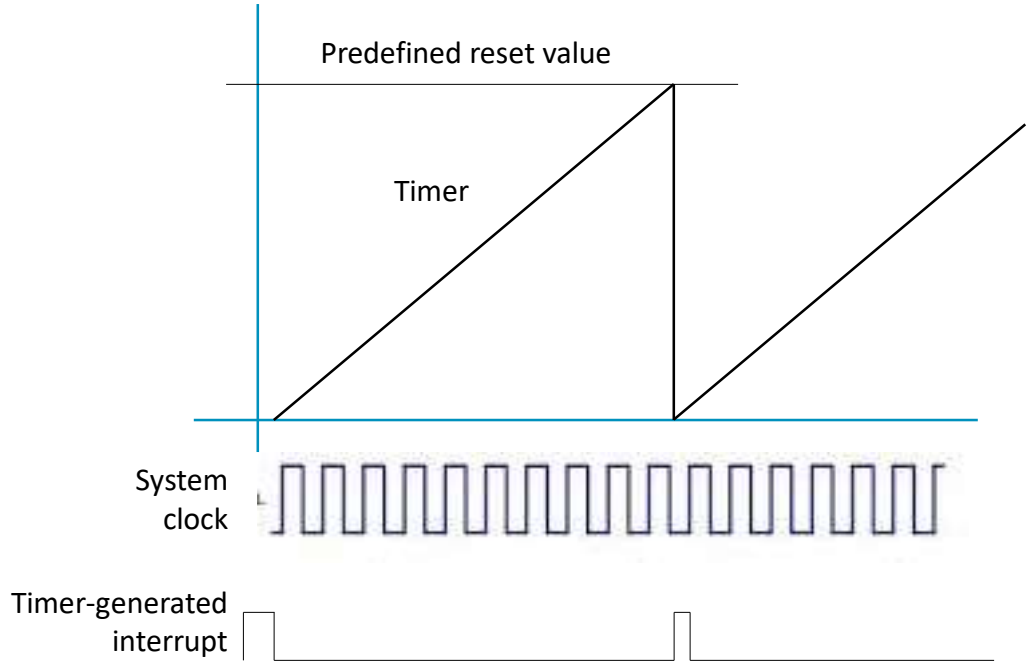
Timer and Pulse-width Modulation

Module Syllabus

- Timer and pulse-width modulation (PWM)
  - Timer overview
  - Components of a standard timer
  - Compare mode
  - Capture mode
  - PWM mode
- Mbed timer and PWM
  - Mbed timer
  - Mbed time ticker
  - Mbed PWM

Timer Overview

- A hardware timer is a digital counter that:
  - Counts regular events, normally using a clock source with a relatively high and fixed frequency
  - Increments or decrements at a fixed frequency
  - Resets itself when reaching zero, or a predefined value
  - Generates an interrupt when reset
- In contrast, a software timer is a similar function block but implemented in software. A software timer usually:
  - Is based on a hardware timer
  - Increments or decrements when interrupted
  - Provides lower time precision compared with hardware timers
  - Can have multiple instances, notably more than hardware timers



arm

Components of a Standard Timer

A prescaler

- Takes the clock source as its input
- Divides the input frequency by a predefined value, e.g., 4, 8, 16
- Outputs the divided frequency to other components

A timer register

- Is incremented or decremented at a fixed frequency
- Is driven by the output from the prescaler, often referred to as “ticks”

A capture register

- Loads the current value from the timer register upon the occurrence of certain events
- Can also generate an interrupt upon events

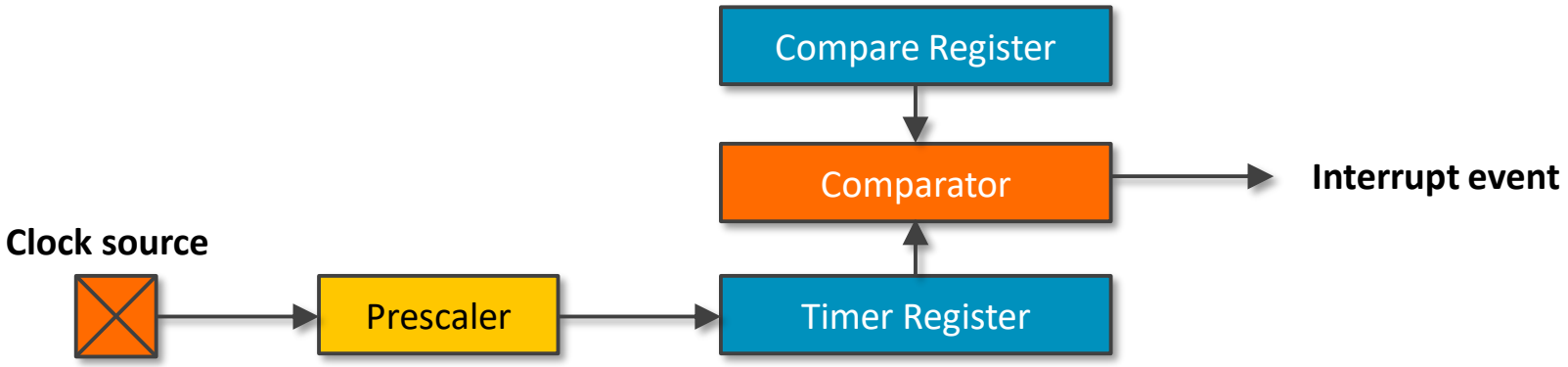
A compare register

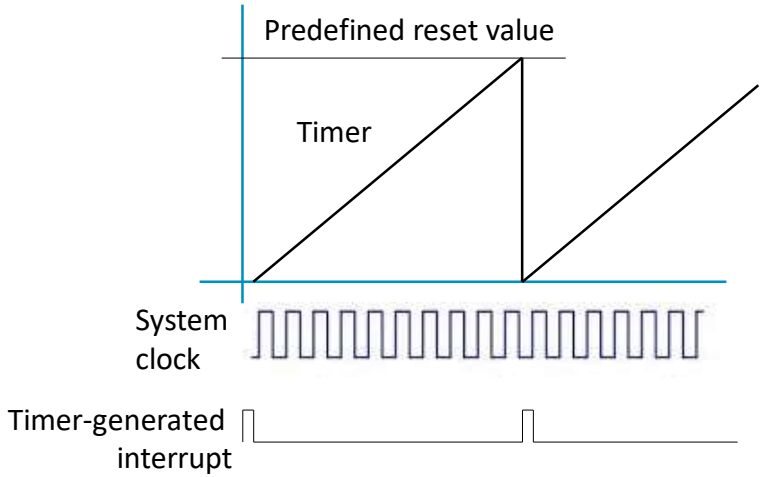
- Is loaded with a value, which is periodically compared with the value in the timer register
- If the two values are the same, an interrupt can be generated.

Timer Operation Mode

- Typically, a standard timer may have three operation modes: compare mode, capture mode, and PWM mode

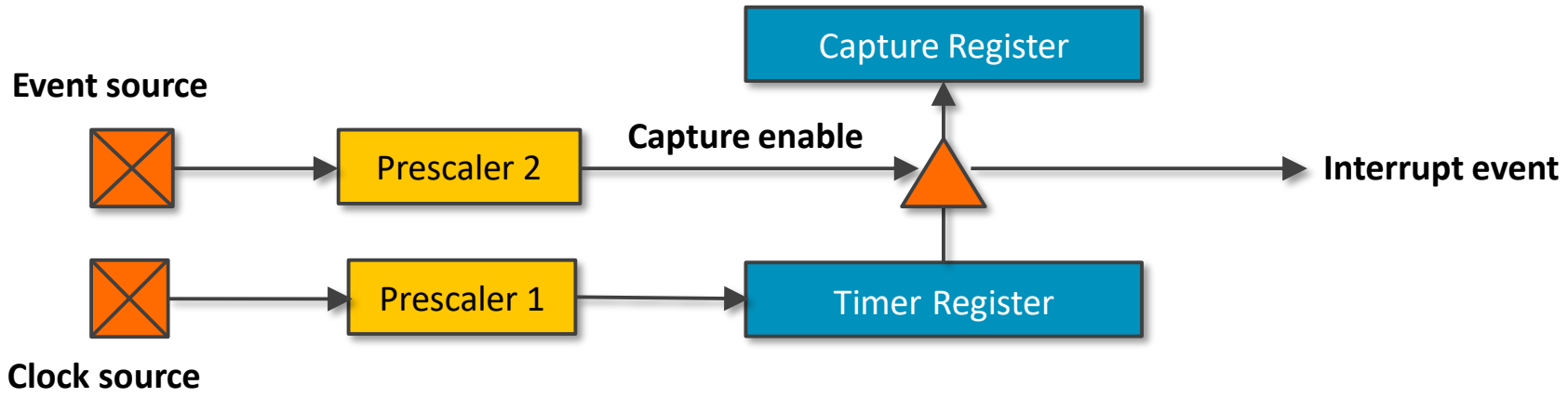
- Compare mode example:
  - Preload the compare register with a desired value
  - Timer register is incremented or decremented automatically at a certain frequency dictated by the prescaler
  - The values in the compare register and the timer register are automatically compared. Once equal, an interrupt can be generated, and the timer register should be reset





Timer Operation Mode

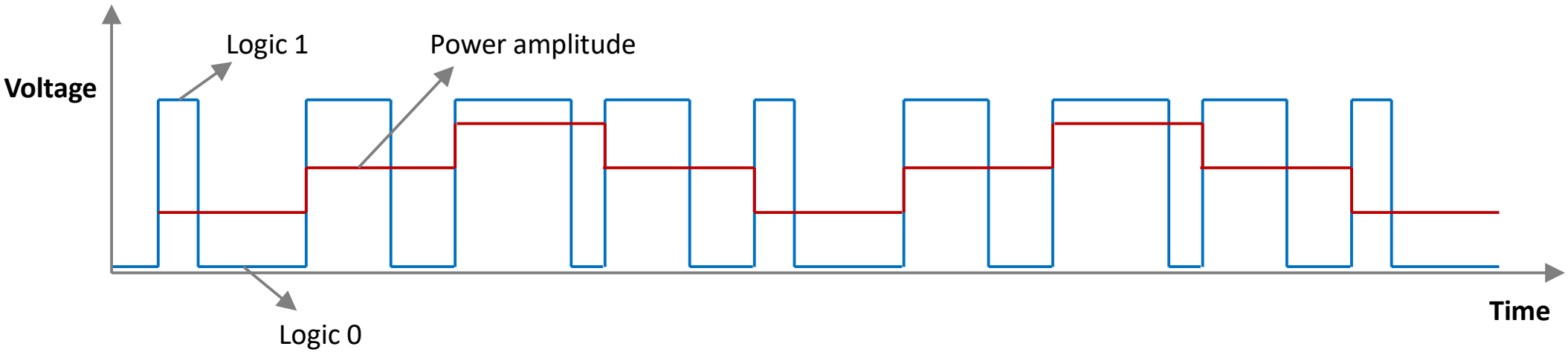
- Capture mode example:
  - The event source generates a sequence of pulses
  - Optionally, the prescaler can be used to divide the frequency of the events
  - Once event (or frequency divided event) occurs, the capture will be enabled
  - The capture register then takes a ‘snapshot’ of the timer register at the moment when the event occurs
  - Optionally, the interrupt can be generated to notify the processor to perform some actions



arm

Timer Operation Mode

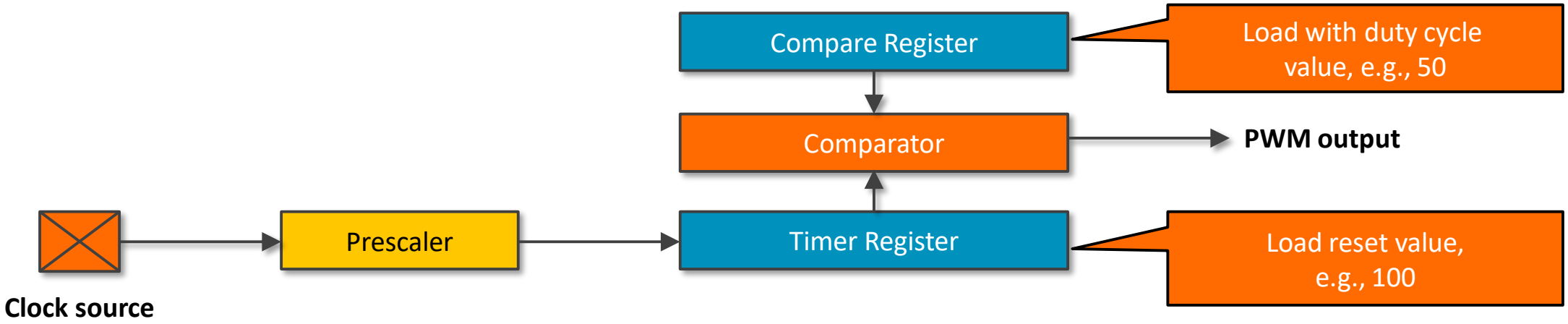
- Pulse-width modulation (PWM) mode example:
  - Uses the width of a pulse to modulate an amplitude
  - The amplitude reflects the duty cycle, which describes the proportion of the ‘1’ state in one pulse period
  - Mainly used to control the power supplied to electrical devices
  - Pulse frequency ranges from few KHz (e.g. motor drive) to tens or hundreds of KHz (e.g. audio amplifier, computer power supplies)



arm

Timer Operation Mode

- Example of PWM mode
  - The PWM mode is similar to the compare mode
  - For example, to generate a 50% power output:
    - Set timer register to reset when reaching 100
    - Set compare register to 50



The implementation of the three operation modes can differ among various devices

Mbed Timer

- The timer interface is used to create, start, stop, and read a timer for measuring small times (between microseconds and seconds)
- Any number of timer objects can be created, and can be started and stopped independently

Function Name	Description
void start ()	Start the timer
void stop ()	Stop the timer
void reset ()	Reset the timer to 0
float read ()	Get the time passed in seconds
int read_ms ()	Get the time passed in milliseconds
int read_us ()	Get the time passed in microseconds

- Since the Mbed API is high-level platform-independent, you do not need to directly access low-level registers. The specification of registers are usually documented in the user guide of the hardware platform

Example of Mbed Timer

- The following code gives a timer example using the Mbed API:

```
#include "mbed.h"

Timer t;

int main() {
    t.start();
    printf("Hello World!\n");
    t.stop();
    printf("The time taken was %f seconds\n", t.read());
}
```

- Note that timers are based on 32-bit int microsecond counters, which means:
  - Only time up to a maximum of 2^31-1 microseconds i.e., 30 minutes, can be registered
  - For longer times, you should consider the time()/Real time clock

Mbed Time Ticker Interrupt

- The timer can be configured to generate a recurring interrupt, to repeatedly call a function at a specified rate, which is known as a time ticker
- Any number of ticker objects can be created, allowing multiple outstanding interrupts at the same time
- The function can be a static function, or a member function of a particular object

Function Name	Description
void attach (void(*fptr)(void), float t)	Attach a function to be called by the Ticker, specifying the interval in seconds
void attach (T *tptr, void(T::*mptr)(void), float t)	Attach a member function to be called by the Ticker, specifying the interval in seconds
void attach_us (void(*fptr)(void), unsigned int t)	Attach a function to be called by the Ticker, specifying the interval in microseconds
void attach_us (T *tptr, void(T::*mptr)(void), unsigned int t)	Attach a member function to be called by the Ticker, specifying the interval in microseconds
void detach ()	Detach the function
static void irq (uint32_t id)	The handler registered with the underlying timer interrupt

Example of Mbed Time Ticker Interrupt

- The following code gives a simple program to set up a ticker to invert an LED repeatedly.
- Note that in the ISR, you should avoid any call to wait, infinitive while loop, or blocking calls in general.

```
#include "mbed.h"

Ticker flipper;
DigitalOut led1(Output Pin 1);
DigitalOut led2(Output Pin 2);

void flip() {
    led2 = !led2;
}

int main() {
    led2 = 1;

    // the address of the function to be attached (flip) and the interval (2 seconds)
    flipper.attach(&flip, 2.0);

    // spin in a main loop. flipper will interrupt it to call flip
    while(1) {
        led1 = !led1;
        wait(0.2);
    }
}
```

Mbed PWM Functions

- Normally, the implementation of PWM is based on a conventional timer
- In the Mbed API, the PWM functions are separated from the timer classes
- The Mbed API functions for PWM are listed below:

Function Name	Description
PwmOut (PinName pin)	Create a PwmOut connected to the specified pin
Void write (float value)	Set the ouput duty-cycle, specified as a percentage (float)
float read ()	Return the current output duty-cycle setting, measured as a percentage (float)
void period (float seconds)	Set the PWM period, specified in seconds (float), keeping the duty cycle the same

Mbed PWM Functions

Function Name	Description
void period_ms (int ms)	Set the PWM period, specified in milliseconds (int), keeping the duty cycle the same
void period_us (int us)	Set the PWM period, specified in microseconds (int), keeping the duty cycle the same
void pulsewidth (float seconds)	Set the PWM pulsewidth, specified in seconds (float), keeping the period the same
void pulsewidth_ms (int ms)	Set the PWM pulsewidth, specified in milliseconds (int), keeping the period the same
void pulsewidth_us (int us)	Set the PWM pulsewidth, specified in microseconds (int), keeping the period the same
PwmOut & operator= (float value)	An operator shorthand for write()
PwmOut & operator float ()	An operator shorthand for read()

Example of Mbed PWM

- The default period is 0.020s, and the default pulse width is 0
- The following code gives an example using PWM to change the luminance of an LED:

```
#include "mbed.h"

PwmOut led(Output Pin);

int main() {
    while(1) {
        for(float p = 0.0f; p < 1.0f; p += 0.1f) {
            led = p;
            wait(0.1);
        }
    }
}
```

arm

Serial Communication

In this material, we use the terms ‘Controller’ and 'Target' in the context of the the I2C and SPI protocols, instead of the terms ‘Master’ and ‘Slave’, which were conventionally used until recently. As a result, the related concepts of 'MISO' and 'MOSI' become 'CITO' and 'COTI'.

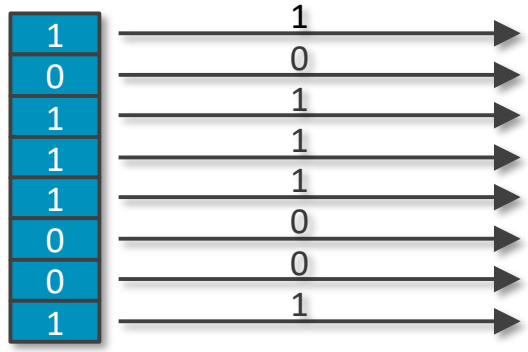
Module Syllabus

- Serial communication overview
- Universal Asynchronous Receiver/Transmitter (UART) communication
  - UART protocol
  - Character-encoding scheme
  - Mbed UART API
- Serial Peripheral Interface (SPI) bus
  - SPI protocol
  - Mbed SPI API
- I2C bus
  - I2C protocol
  - Mbed I2C API

Serial Communication

- Serial communication:
  - Transmits data one bit at a time, in a sequential fashion
  - Contrast with parallel communication, in which multiple bits are transmitted at the same time
  - Commonly used for long-haul communication, modems, and non-networked communication between devices
  - Examples are UART, SPI, I2C, USB, Ethernet PCI Express, etc.

Serial communication



Parallel communication

arm

Serial vs. Parallel Communication

- Cost and weight
  - Serial communication is lower cost and weight than parallel communication, as fewer wires and smaller connectors are needed.
- Serial communication is more reliable.
  - Parallel communications may introduce more clock skews, as well as crosstalk between different wires.
- Higher clock rate
  - Due to higher reliability, serial communication can be clocked at a higher frequency than parallel communication, hence increase throughput.
- On the other hand, the conversion between serial and parallel data may consume extra overheads.

Types of Serial Communication

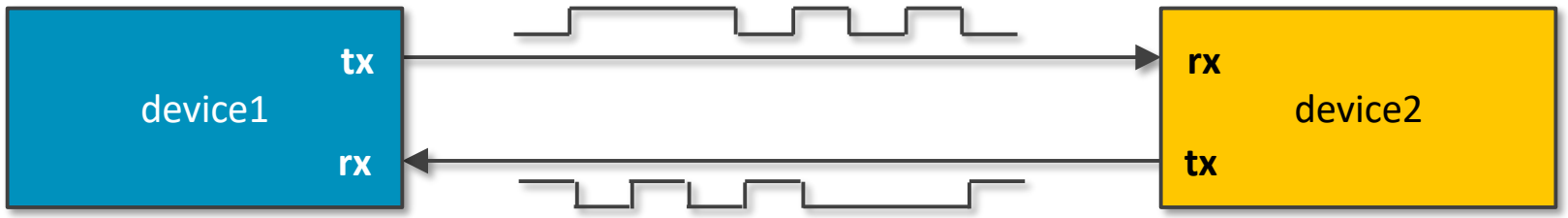
- Synchronous Serial Transmission
  - A common clock is shared by both the sender and the receiver
  - More efficient transmission, since one wire is dedicated to data transfer
  - Costly if an extra clock wire is required

- Asynchronous Serial Transmission
  - The sender does not have to send a clock signal
  - Both sender and receiver agree on timing parameters in advance
  - Special bits are added to synchronize transmission

UART Overview

- Universal Asynchronous Receiver/Transmitter (UART)
  - Asynchronous communication, no clock wire required, pre-agreed baud rate
  - Separate transmit and receive wires

- UART communication
  - Converts data from parallel to serial
  - Sequential data is transferred through serial cable
  - Receives the sequential data and reassembles it back to parallel



arm

UART Protocol

- Data transfer starts with a starting bit, by driving logic low for one clock cycle
- In the next 8 clock cycles, 8 bits are sent sequentially from the transmitter
- Optionally, 1 parity bit can be added to improve transfer reliability
- In the end, the data wire is pulled up high to indicate the end of transfer

Transfer one byte without parity bit

Transfer one byte with parity bit

arm

Character-Encoding Scheme

- What is the data to be sent to the UART in order to display a text?

- American Standard Code for Information Interchange (ASCII):
  - Encodes 128 characters
    - 95 printable characters, such as ‘a ’, ‘b’, ‘1’, ‘2’, etc.
    - 33 non-printing control characters, e.g., next line, back space, escape
  - Can be represented by 7 bits, commonly stored as one byte for storage convenience

- UTF-8 (UCS Transformation Format—8-bit):
  - Derived from ASCII (2007)
  - Variable-width encoding scheme, which avoids the complication of endianness and byte order marks
  - Widely used for the World Wide Web
  - Compatible with the original ASCII

ASCII Encoded Characters

- The table below lists some frequently used characters coded in ASCII:

Hex	Character	Hex	Character	Hex	Character
0x30	0	0x41	A	0x61	a
0x31	1	0x42	B	0x62	b
0x32	2	0x43	C	0x63	c
0x33	3	0x44	D	0x64	d
0x34	4	0x45	E	0x65	e
0x35	5	0x46	F	0x66	f
0x36	6	0x47	G	0x67	g
0x37	7	0x48	H	0x68	h
0x38	8	0x49	I	0x69	i
0x39	9	0x4A	J	0x6A	J
…		…		…	

Mbed API for UART

- The Mbed API provides the following functions to access the UART peripheral:

Function Name	Description
Serial (PinName tx, PinName rx, const char *name=NULL)	Create a Serial port connected to the specified transmit and receive pins
void baud (int baudrate)	Set the baud rate of the serial port
Void format (int bits=8, Parity parity=SerialBase::None, int stop_bits=1)	Set the transmission format used by the serial port
int readable ()	Determine if there is a character available to read
int writeable ()	Determine if there is space available to write a character
void attach (void(*fptr)(void), IrqType type=RxIrq)	Attach a function to call whenever a serial interrupt is generated
void send_break ()	Generate a break condition on the serial line
void set_flow_control (Flow type, PinName flow1=NC, PinName flow2=NC)	Set the flow control type on the serial port

Mbed API for UART

- Serial channels have a number of configurable parameters:
  - Baud rate: There are a number of standard baud rates ranging from a few hundred bits per seconds, to megabits per second. The default setting for a serial connection on the mbed microcontroller is 9600 baud
  - Data length: Data transferred can be either 7 or 8 bits long. The default setting for a serial connection on the mbed microcontroller is 8 bits
  - Parity: An optional parity bit can be added. The parity bit will be automatically set to make the number of 1's in the data either odd or even. Parity settings are odd, even or none. The default setting for a serial connection on the mbed microcontroller is for the parity to be set to none
  - Stop bits: After data and parity bits have been transmitted, 1 or 2 stop bit(s) is/are inserted to ‘frame’ the data. The default setting for a serial connection on the mbed microcontroller is for one stop bit to be added
  - The default settings for the mbed microcontroller are described as 9600 8N1, and this is common notation for serial port settings

Example of Mbed UART API

- With the support of the UART peripheral, a number of standard IO functions can be directly used, such as:

Function Name	Description
int putc(int ch, FILE *stream)	Writes the character ch to stream. Function returns the character written, or EOF if an error happens
int getc(FILE *stream)	Read a character from the stream, an EOF indicates the end of file is reached
int printf(const char *format, ...)	Prints both text string and data, according to format and other arguments passed to printf()

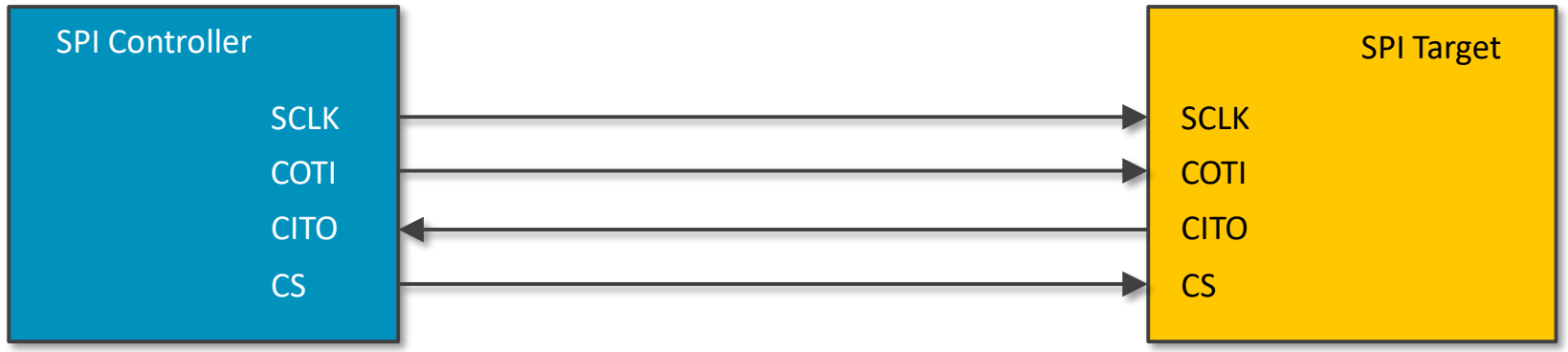
Example of Mbed UART API

- The following example first prints a string to the host PC, and then receives/sends characters from/to the PC via UART:

```
#include "mbed.h"

Serial pc(UART_TX, UART_RX); // tx, rx

int main() {
    pc.printf("Hello World!\n");
    while(1) {
        pc.putc(pc.getc() + 1);
    }
}
```

SPI Overview

- Serial Peripheral Interface (SPI) bus:
  - Synchronous communication, clock wire required
  - Named by Motorola
  - Separate transmit and receive wires
  - Four-wire serial bus
  - Devices communicate in controller/target mode
    - Controller initiates data transfers
    - Multiple targets are enabled individually by the select lines

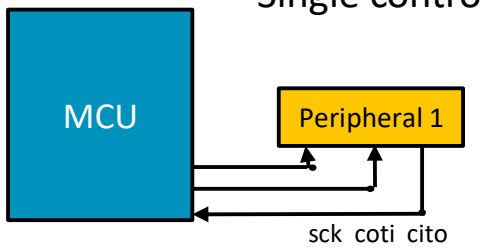


arm

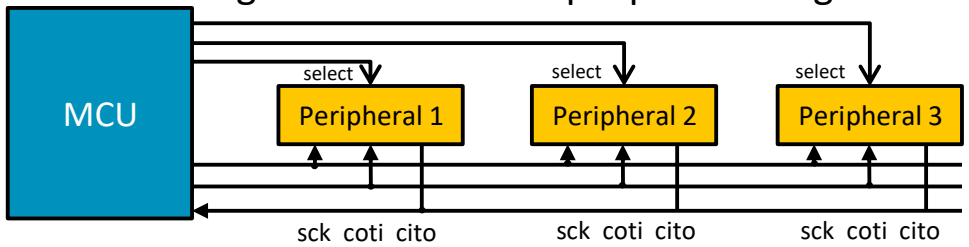
SPI Overview

- All chips share bus signals
  - Clock SCK
  - Data lines COTI (controller out, target in) and CITO (controller in, target out)
- Each peripheral has its own chip select line (CS)
  - Controller (MCU) only asserts the CS line of the peripheral with which it wants to communicate

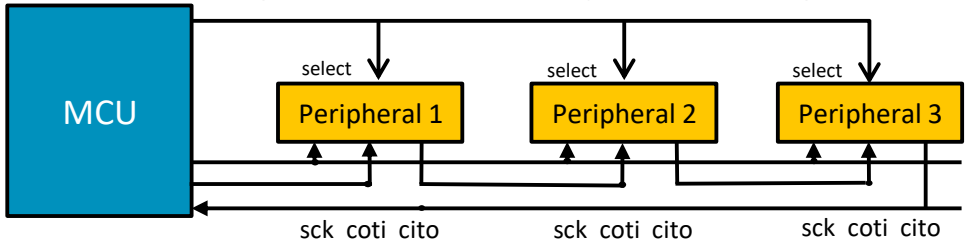
Single controller single target



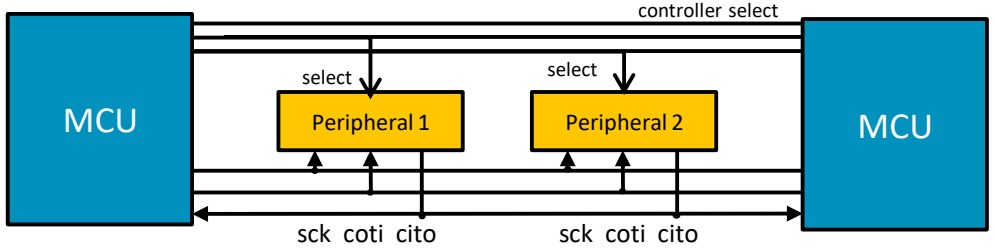
Single controller multiple parallel targets



Single controller multiple serial targets

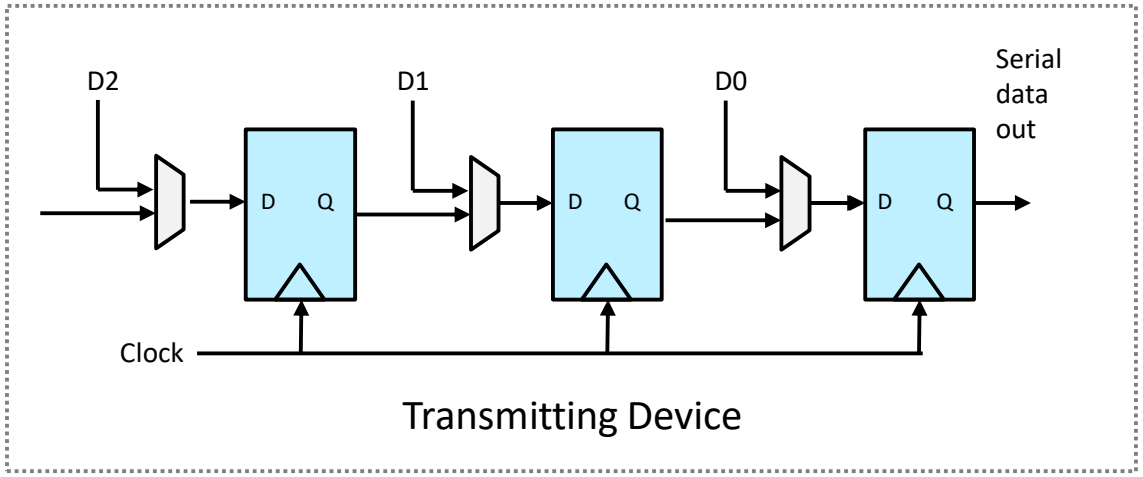


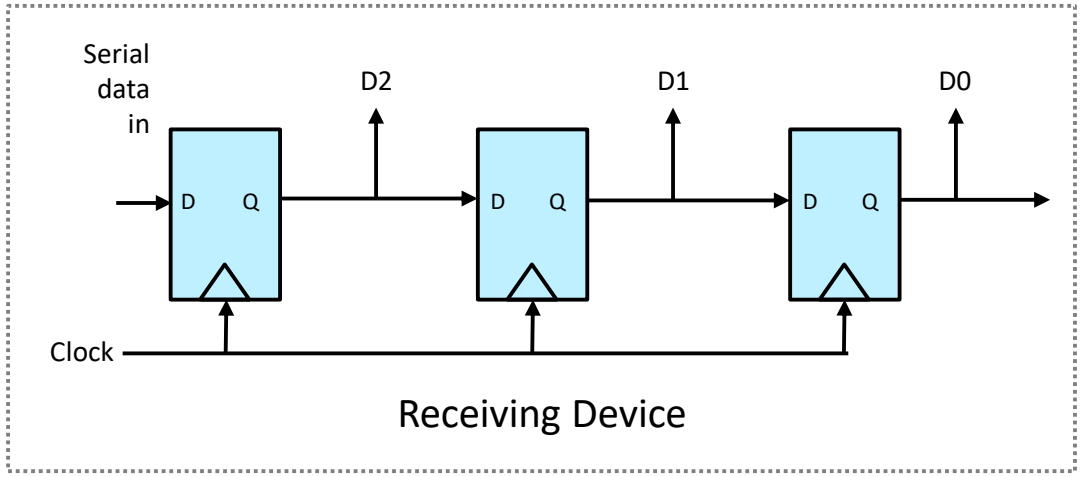
Multiple controllers multiple parallel targets

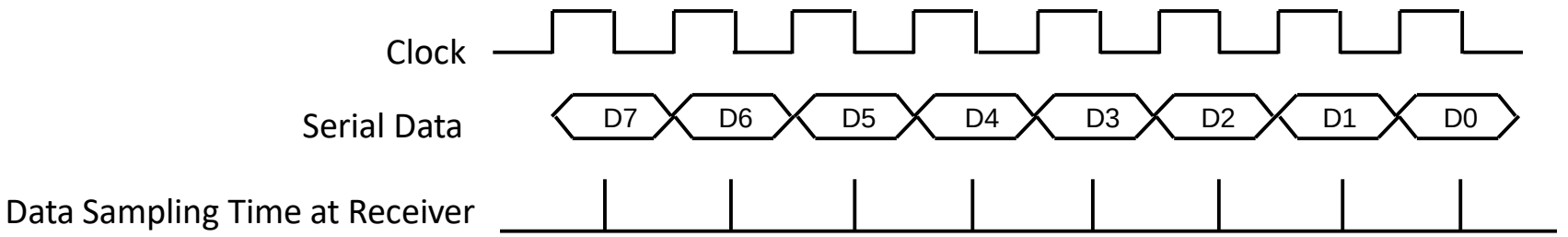


Serial Data Transmission

- Uses shift registers and a clock signal to convert between serial and parallel formats
- Synchronous: an explicit clock signal along with the data signal







Mbed API for SPI

- The mbed API provides the following functions to access the SPI peripheral:

Function Name	Description
SPI (PinName coti, PinName cito, PinName sclk, PinName _unused=NC)	Create a SPI controller connected to the specified pins
void format (int bits, int mode=0)	Configure the data transmission format
void frequency (int hz=1000000)	Set the spi bus clock frequency
virtual int write (int value)	Write to the SPI Target and return the response

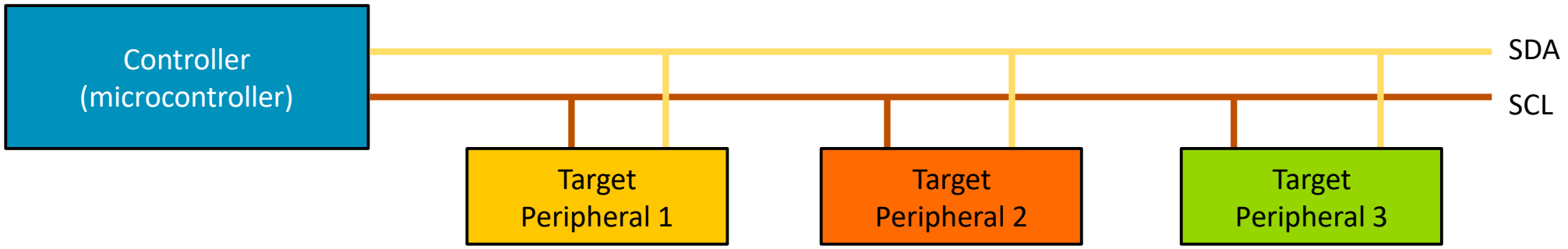
- The SPI Interface can be used to write data words out of the SPI port, returning the data received back from the SPI target.
- The SPI clock frequency and format is configurable. In addition to the frequency, the controller must also configure the mode defined as the clock polarity and phase with respect to the data (see http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus).
- The default settings of the SPI interface are 1MHz, 8-bit, Mode 0.

Example of Mbed SPI API

```
#include "mbed.h"

SPI spi(SPI_COTI, SPI_CISO, SPI_SCLK);
DigitalOut cs(SPI_CS);

int main() {
    // Chip must be deselected
    cs = 1;
    // Setup the spi for 8 bit data, high steady state clock,
    // second edge capture, with a 1MHz clock rate
    spi.format(8,3);
    spi.frequency(1000000);
    // Select the device by setting chip select low
    cs = 0;
    // Send 0x8f, the command to read the WHOAMI register
    spi.write(0x8F);
    // Send a dummy byte to receive the contents of the WHOAMI register
    int whoami = spi.write(0x00);
    printf("WHOAMI register = 0x%X\n", whoami);
    // Deselect the device
    cs = 1;
}
```

I²C Bus Overview

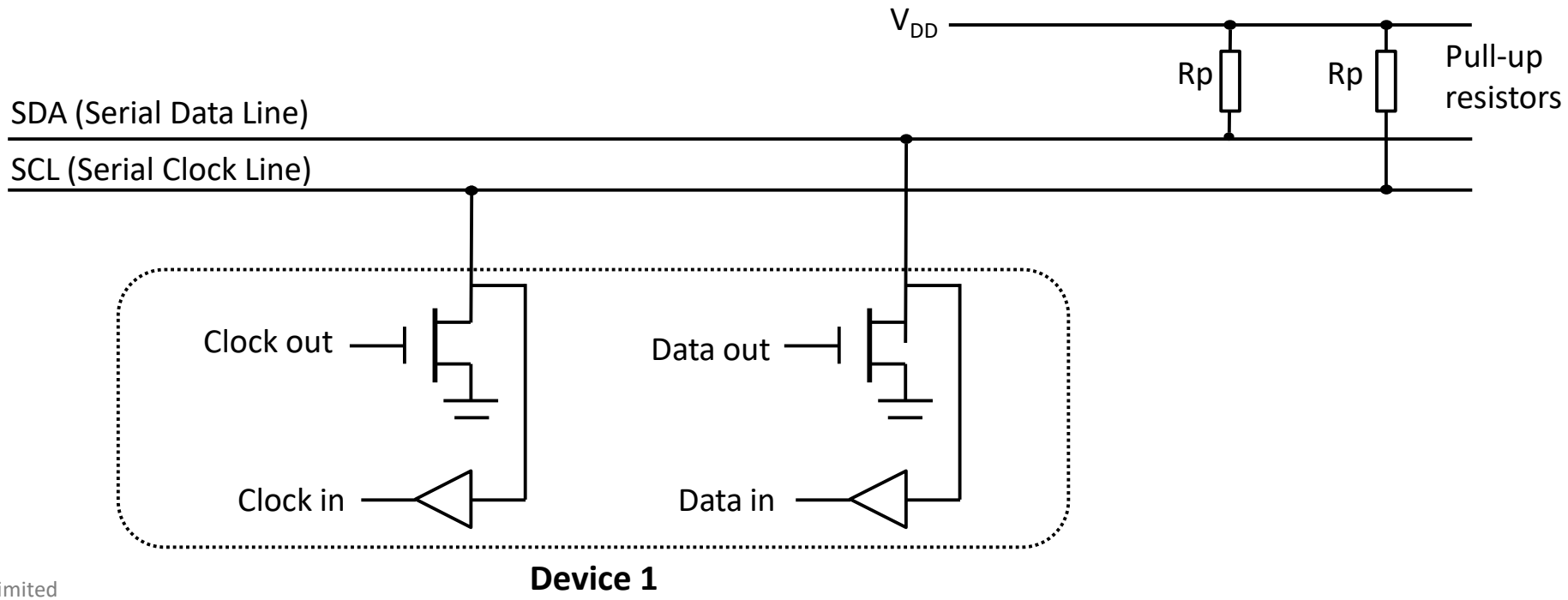
- Inter-integrated Circuit (I2C) bus:
  - Multi-controller serial single-ended computer bus
  - Invented by Philips semiconductor division (today: NXP Semiconductors)
  - Communicates with low-speed peripherals
  - Two signal lines
    - SCL: Serial clock
    - SDA: Serial data



arm

I²C Bus Connections

- Bus is typically controlled by controller device; targets respond when addressed.
- Resistors pull up lines to VDD
- Open-drain transistors pull lines down to ground
- Controller generates SCL clock signal
  - Can range up to 400 kHz, 1 MHz, or more



I²C Message Format

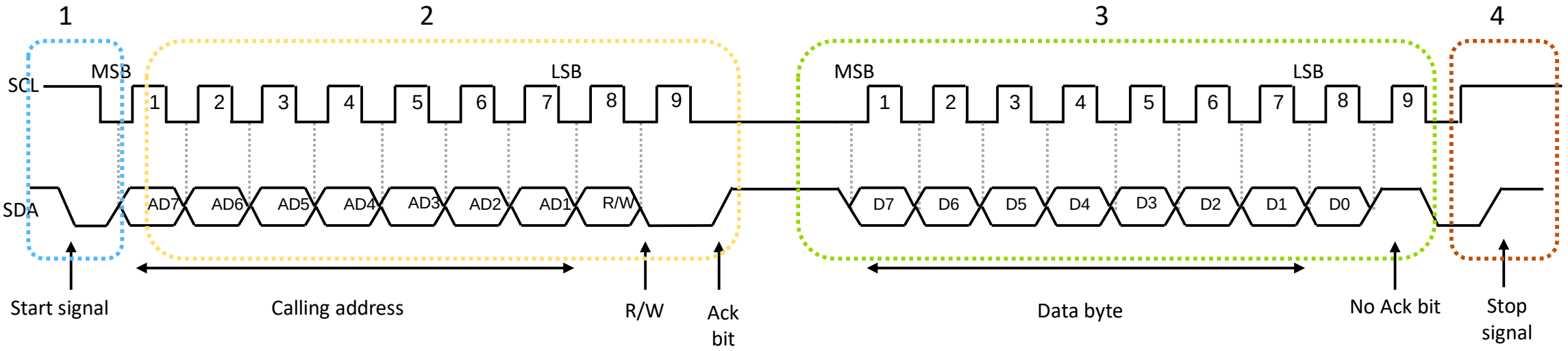
- **Message-oriented data transfer with four parts:**

1. Start condition

2. Target address transmission
   - Address
   - Command (read or write)
   - Acknowledgement by receiver

3. Data fields
   - Data byte
   - Acknowledgement by receiver

4. Stop condition



Mbed API for I^2C

- The mbed API provides the following functions to access the I2C peripheral:
  - The default frequency of the I^2C interface is 100KHz

Function Name	Description
I2C (PinName sda, PinName scl)	Create an I2C Controller interface connected to the specified pins
void frequency (int hz)	Set the frequency of the I2C interface
int read (int address, char *data, int length, bool repeated=false)	Read from an I2C target
int read (int ack)	Read a single byte from the I2C bus
int write (int address, const char *data, int length, bool repeated=false)	Write to an I2C target
int write (int data)	Write single byte out on the I2C bus
void start (void)	Creates a start condition on the I2C bus
void stop (void)	Creates a stop condition on the I2C bus

Example of Mbed I²C API

- The example below shows how to write and read a register of an I²C device:

```
#include "mbed.h“

I2C i2c(I2C_SDA, I2C_SCL);    //Initialize i2c peripheral

const int addr = 0x10;        //Device address

int main() {
    char data[2];

    //Write operation
    data[0] = 0x00;           //Register address
    data[1] = 0x0A;           //Data to be written
    i2c.write(addr, data, 2); //Write data to the register

    //Read operation
    data[0] = 0x02;           //Register address
    i2c.write(addr, data, 1); //Set register address
    i2c.read(addr, data, 2);  //Read data from the register
}
```

arm

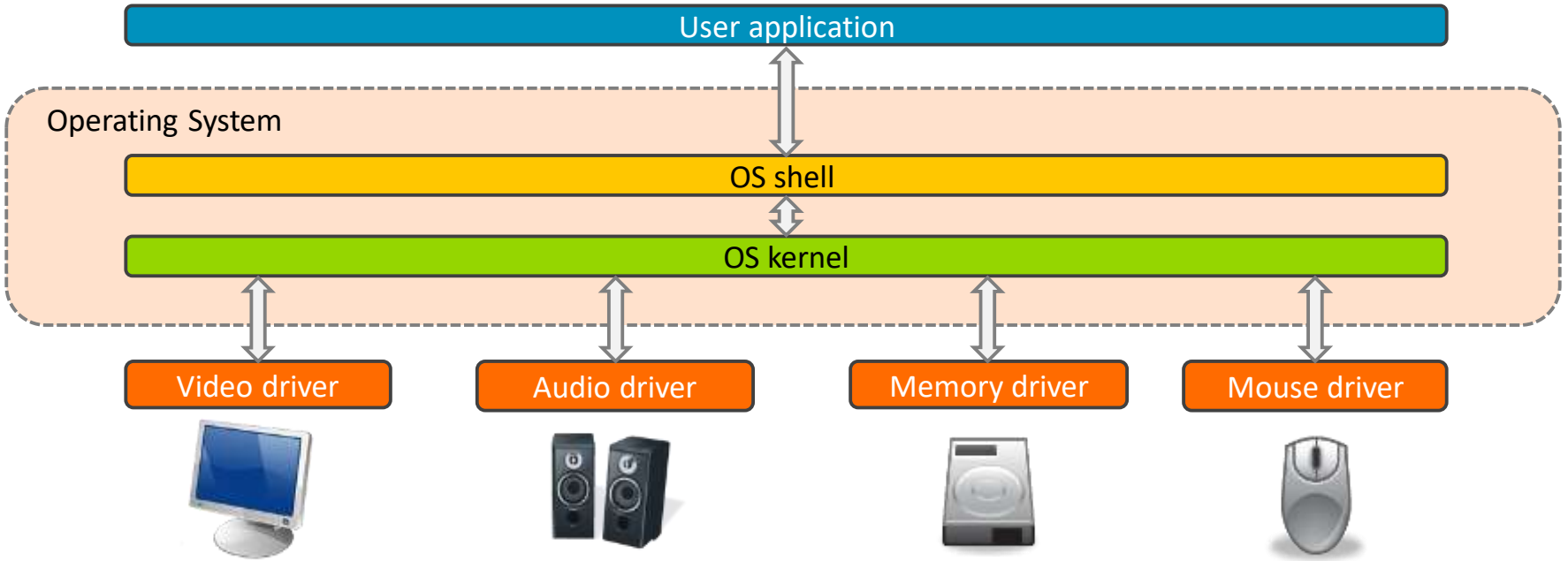
Real-Time Operating Systems

Module Syllabus

- Operating System Overview
  - What is an Operating System?
  - Functions, types, and services of Operating Systems
- Real-Time Operating System (RTOS)
  - RTOS overview
  - RTOS task scheduling
  - Keil RTX RTOS
- RTOS on Mbed Platform
  - Mbed RTOS API
  - Using Mbed RTOS API for your project
  - Threads, Mutex, and Semaphore

Operating System Overview

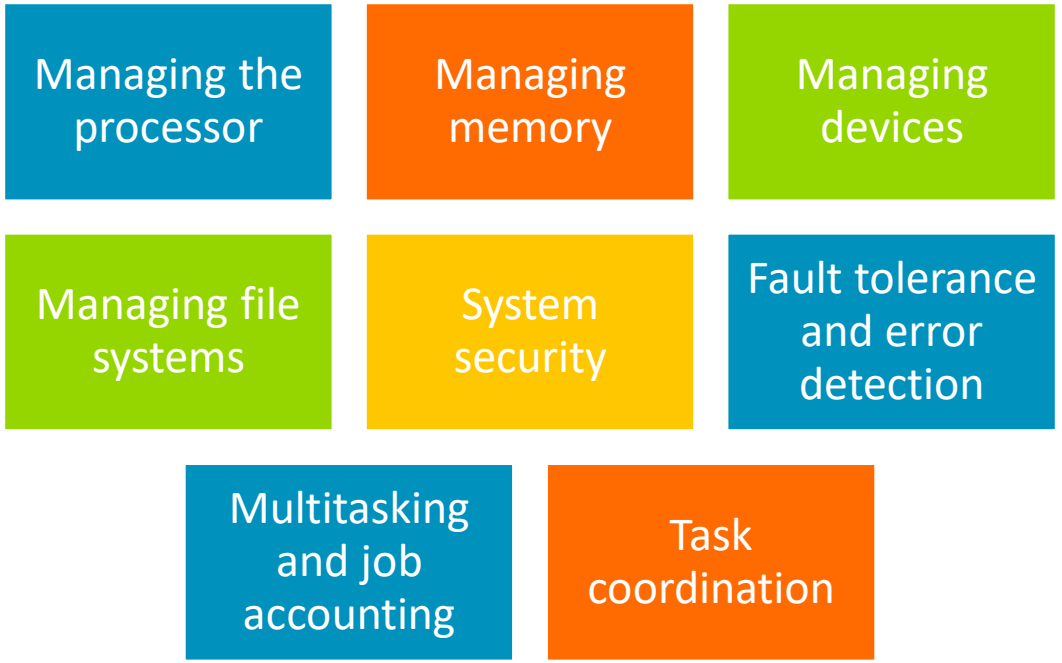
- Operating System (OS):
  - An intermediary interface between user applications and computer hardware
  - Facilitates application development (convenience and efficiency)
  - Various OSs are available in the market for various hardware platforms, e.g., Windows, Linux, Unix, Mac OS, Android, iOS.



arm

Functions of an Operating System

- An operating system acts as an interface between the high-level user application and the low-level hardware components, and it is usually capable of:



arm

Operating System Services

- Basic operating system services may include:

Types of Operating Systems

- Single-user OS
  - Only one user can access it
  - May allow multiple programs to run at the same time

- Multi-user OS
  - Allows multiple users to access it at the same time

  - Batch operating OS
    - Users prepare their jobs offline
    - Similar jobs submitted by different users are batched together and run as a group
    - Aims to maximize processor usage – less interaction with user applications

  - Time-sharing OS
    - Processor's time is shared among multiple users
    - More interaction with user applications

Types of Operating System

- Distributed OS
  - Processing is distributed across multiple CPUs
  - Processors are interconnected via communication lines, such as internal high-speed buses or various types of networks
- Embedded OS
  - Designed to be used in embedded computer systems
  - Limited resources such as memory, IOs, clock speed
  - Compact and energy efficient
- Real-Time OS
  - Multitasking OS targeted at real-time applications, with real-time constraints
  - Must quickly and predictably respond to events

Real-Time Operating System Overview

- A Real-time OS (RTOS) is an OS that:
  - Serve real-time applications
  - Responds to requests within a guaranteed and predictable delay
  - Processes data in deterministic cycles

- RTOS aims at deterministic performance foremost, rather than high throughput
  - Performance can be measured by jitter: the variability of the time it takes to complete a task
  - Soft RTOS: more jitter, but usually or generally meets a deadline
  - Hard RTOS: less jitter, can meet a deadline deterministically

Real-time (definition): Real-time guarantees the completion of a process within a defined time interval. Real-time does not make any statement about the total time duration or the processing speed.

RTOS Design Philosophies

- There are two common designs for RTOS:
  - Event-driven RTOS
    - Pre-emptive task scheduling
    - An event of higher priority needs to be served first
    - Processes data more responsively
  - Time-sharing RTOS
    - Non-pre-emptive task scheduling
    - Tasks are switched on a regular clocked interrupt, and on events, e.g., round robin scheduling
    - Tasks are switched more often, giving a smoother multitasking
    - However, unnecessary task switching results in undesired overheads

RTOS Task Scheduling

- In a typical RTOS, a task can have at least three states:
  - Running: task is currently executed by the CPU
  - Ready: task is ready to be executed by the CPU
  - Blocked: task is paused and waiting for an event, such as from I/O, to resume its execution

- Task scheduling
  - Usually, only one task per CPU can run at any one time.
  - To efficiently switch between tasks, usually a task scheduler is used. Various scheduling algorithms exist, including:
    - Cooperative scheduling: no pre-emption, tasks can end themselves in a cooperative manner
    - Pre-emptive scheduling: tasks can be interrupted by other tasks of higher priorities
    - 'Earliest deadline first' approach: the task with the earliest deadline is served first

- Examples of RTOS include LynxOS, OSE, Windows CE, FreeRTOS, Arm Keil RTX, etc.

Highlights of Arm Keil RTX RTOS

- Deterministic RTOS designed for Arm Cortex-M-based devices
- Multitasking with flexible scheduling: round-robin, pre-emptive, and collaborative
- High-speed real-time operation with low interrupt latency
- Small footprint for resource-constrained systems
- Royalty-free, deterministic RTOS with source code
- Unlimited number of tasks each with 254 priority levels
- Support for multithreading and thread-safe operation
- Inter-task communication manages the sharing of data, memory, and hardware resources among multiple tasks
- Unlimited number of mailboxes, semaphores, mutex, and timers (hardware-permitting)
- Defined stack usage - each task is allocated a defined stack space, enabling predictable memory usage

arm

Mbed RTOS

- Mbed RTOS API:
  - Provides easy-to-use API for programming mbed-enabled devices
  - Based on Keil RTX RTOS kernel implementation
  - Uses the CMSIS-RTOS API open standard

- CMSIS-RTOS API :
  - Common API for real-time operating systems
  - Foundation of the official Mbed RTOS
  - Provides a standardized programming interface that is portable to many RTOSs
  - Hence enables software templates, middleware, libraries, and other components that can work across various supported RTOS systems

- The following slides show how to use some mbed RTOS API features, such as threads, mutexes, and semaphores

Threads

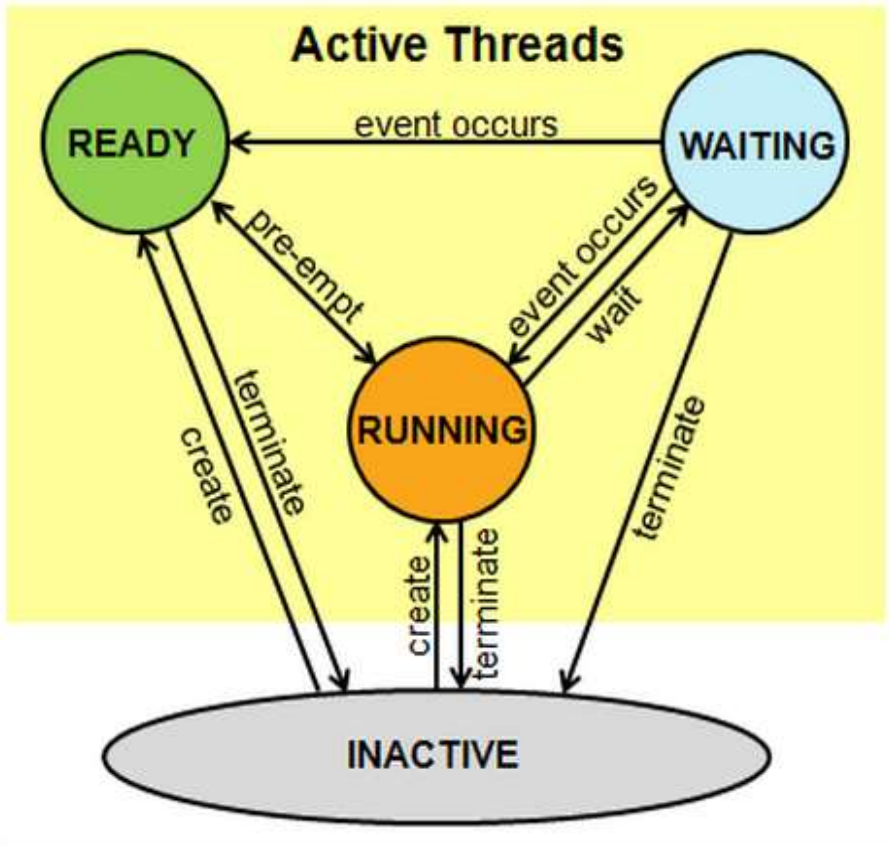
- A task can sometimes be referred to as a thread in multitasking systems
- A Thread in mbed API can be in the following states:

•**Running:** The thread that is currently running is in the Running state. Only one thread at a time can be in this state.

•**Ready:** Threads that are ready to run are in the Ready state. Once the Running thread has terminated or is Waiting, the next Ready thread with the highest priority becomes the Running thread.

•**Waiting:** Threads that are waiting for an event to occur are in the Waiting state.

•**Inactive:** Threads that are not created or terminated are in the Inactive state. These threads typically consume no system resources.



arm

Mbed API for Threads

- The following table lists some thread-related basic functions of the Mbed RTOS API:

Function Name	Description
Thread (void(*task)(void const *argument), void *argument=NULL, osPriority priority=osPriorityNormal, uint32_t stack_size=DEFAULT_STACK_SIZE, unsigned char *stack_pointer=NULL)	Create a new thread, and start it executing the specified function
osStatus terminate ()	Terminate execution of a thread and remove it from active threads
osStatus set_priority (osPriority priority)	Set priority of an active thread
osPriority get_priority ()	Get priority of an active thread
int32_t signal_set (int32_t signals)	Set the specified Signal Flags of an active thread
State get_state ()	State of this thread

Thread Example

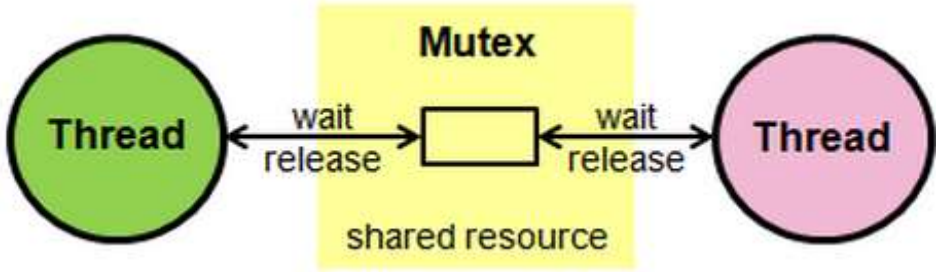
- The example below shows how to create threads to blink LEDs:

```
#include "mbed.h"
#include “rtos.h"

DigitalOut led1(Ouput Pin 1);
DigitalOut led2(Output Pin 2);

void led2_thread(void const *args) {
    while (true) {
        led2 = !led2;
        Thread::wait(1000);
    }
}

int main() {
    Thread thread1(led2_thread);

    while (true) {
        led1 = !led1;
        Thread::wait(500);
    }
}
```

Mutex

- Mutex: Mutual Exclusion Object
  - Ensures no two threads are in their critical section accessing a shared resource at the same time

- In RTOS, the mutex is used to:
  - Synchronize the execution of threads
  - Protect access to a shared resource, such as a shared memory image



arm

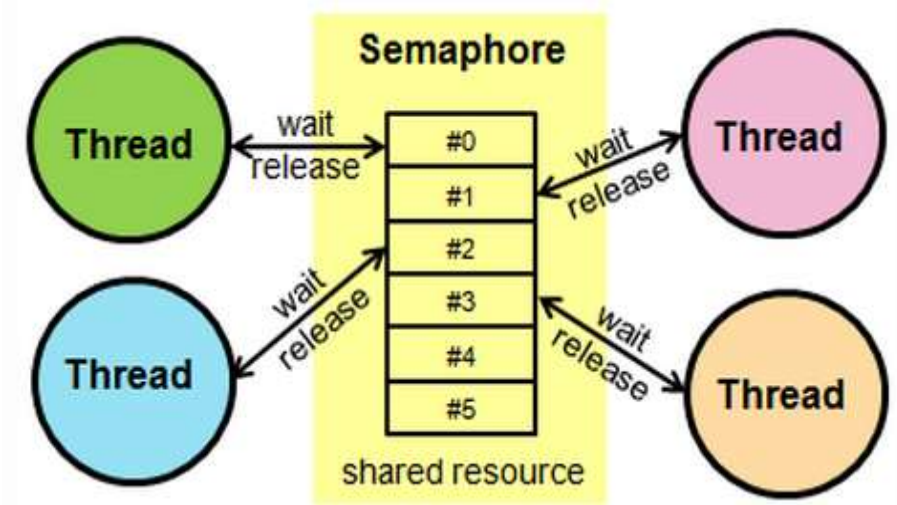
Mutex Example

- The following example shows how a mutex is used to synchronize two threads both writing output to standard output:

```
#include "mbed.h"
#include "rtos.h"

Mutex stdio_mutex;

void notify(const char* name, int state) {
    stdio_mutex.lock(); // lock standard output if it not locked already
    printf("%s: %d\n\r", name, state); // Current thread is the owner, print to standard output
    stdio_mutex.unlock(); // unlock standard output
}

void test_thread(void const *args) {
    while (true) {
        notify((const char*)args, 0); Thread::wait(1000);
        notify((const char*)args, 1); Thread::wait(1000);
    }
}

int main() {
    Thread t2(test_thread, (void *)"Th 2");
    Thread t3(test_thread, (void *)"Th 3");

    test_thread((void *)"Th 1");
}
```

Semaphore

- A semaphore is a variable of abstract data type used to manage and protect access to shared resources. Unlike a mutex, it does not have the concept of an owner
  - Unlike a mutex, a semaphore can control access to several shared resources
  - For example, a semaphore enables access to and management of a group of identical peripherals



Semaphore Example

- The following example shows how to use a semaphore to manage thread access to a pool of shared resources of a certain type:

```
#include "mbed.h"
#include "rtos.h"

Semaphore two_slots(2); // a semaphore to control 2 resources

void test_thread(void const *name) {
    while (true) {
       // block if no resource is available, otherwise decrement semaphore (resource count)
       two_slots.wait();
       // when standard output is available, print thread name
        printf("%s\n\r", (const char*)name);
        // delay
        Thread::wait(1000);
       // release semaphore, increment semaphore (resource count)
       two_slots.release();
    }
}

int main (void) {
    Thread t2(test_thread, (void *)"Th 2");
    Thread t3(test_thread, (void *)"Th 3");

    test_thread((void *)"Th 1");
}
```