UNIVERSITÀ DI TRENTO | Dipartimento di Ingegneria e Scienza dell'Informazione

Department of Engineering and Computer Science

Bachelor degree in
Computer, Communication and Electronic Engineer

Embedded Software for the Internet of Things

Professor

Kasim Sinan Yildirim

Student

Cristiano Berardo 234428

Academic Year 2024/2025

Contents

Chapter 1

Embedded Systems Overview and Key Concepts

1.1 Definition of Embedded System

Embedded System are computing system with **strongly coupled** hardware and software integration, designed to perform a **dedicate functionality.**

The world **embedded** means: build into, to be an integral part of, a large system.
The larger system is called **Embedding System**.

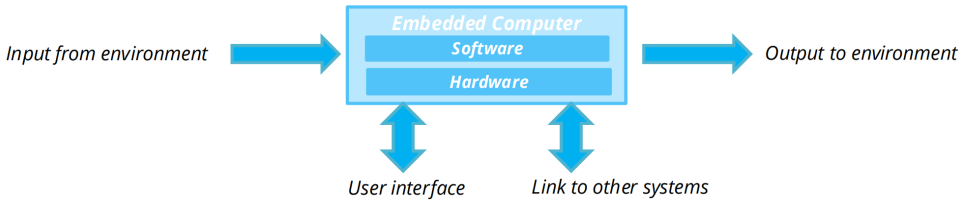


Figure 1.1: Embedded Computer

Another definition of embedded system:

Application-specific computer system often with **real-time computer constraints**. Implemented using **MCUs**, MicroControllerUnit.

Embedded systems are added to larger system for different reasons:

- Lower cost;
- Lower performance;
- Energy efficiency;
- More Functions and Features.

1.1.1 Example of some Embedded Systems

Bike Computer

- Functions: speed and distance measurements;
- Constraints: size, cost, power and energy, weight;
- Inputs: wheel rotation that indicate how fast we are;
- Output: Display the speed;
- Use Low-Performance Microcontroller: 8-bit, 10 MIPS.

We can see that all the constraint change the design, software and hardware integration.

Automotive Embedded System Nowadays, cars are fully of microprocessors that detect different events that occur while driving or not:

- 4-bit microcontroller checks seat belts;
- Microcontrollers run dashboard device;
- 16/32-bit microprocessor controls the engine...

1.2 Microprocessor VS Microcontroller

What is the difference between **Microprocessor**, CPU, and**Microcontroller**, MCU?
Both have a **CPU core** to execute instructions.

1.2.1 Microprocessor, CPU

The **microprocessor (CPU) is a single core processor** that support at least the instruction of *fetching*, *decoding* and *executing*.
Normally it is used for a general purpose computer, it also **need memory and I/O**.

Nevertheless, this CPU uses the same logic to perform **different tasks**, so we can change the **program** within the memory to perform different algorithms, thus simplifying the design and the usability.

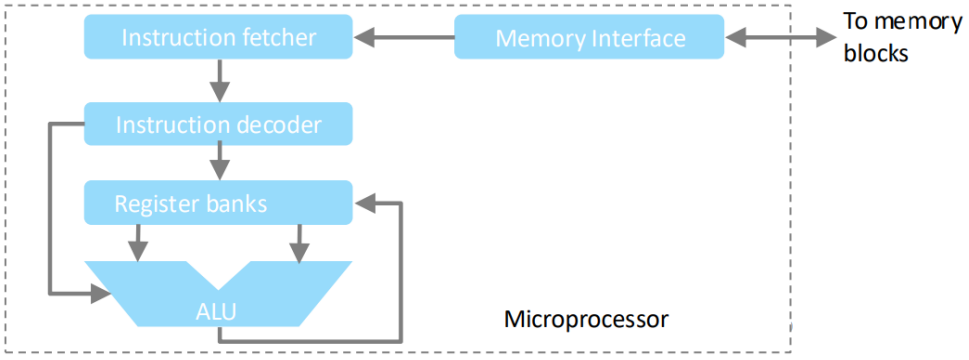


Figure 1.2: CPU schema

Alternative? Field-Programmable Gate Arrays (also known as **FPGAs**), custom logic, etc. All of this things are a **dedicate hardware** that has a special purpose, indeed they are designed for a specific function. In general they use less power than CPU but they can not be used for other functions.

Modern microprocessors also offer features to control power consumption, via software we can reduce the consumption (sleep mode).
Microprocessor are used in combination with some custom logic for a well-defined functions, while the CPU and software is used for everything else.

1.2.2 Microcontroller, MCU

As written above, the MCU generally has a single-core processor, memory blocks, Digital and Analog I/Os and other basic pheripherals. It is typically used for a **basic control purposes**.

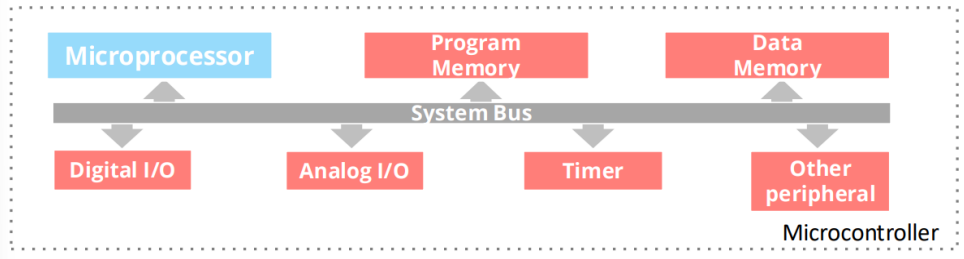


Figure 1.3: Microcontroller

1.2.3 How to chose between CPUs and MCUs?

In most embedded system, MCUs are chosen to be the best solution because they offers:

- Low development and manufacturing cost;
- Easy porting and updating;
- Light footprint;
- Relatively low power consumption;
- Good performance for low-end products.

	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	mid	easy	small	low	medium to high	very fast
Software Running on Generic Hardware	Microprocessor + memory + peripherals	low to mid	mid	easy	small to med.	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

Figure 1.4: Options for Building Embedded Systems

Unless there's a very specific reason, such as some constraints, the best choice for embedded systems is software running on generic hardware.

1.2.4 Basic functionality of an Embedded System

Monitoring Applications: monitor a process and adjust an output to maintain desired set point, e.g. temperature, speed...

Sequencing: step through different stages, called states, based on environment and system;

Digital signal processing: remove noise, select desired signal features;

IoT applications: exchange information reliably and quickly using near-by communications and networks.

1.3 Challenges in Embedded System Design

There are lots of challenges when we use the embedded system some could be: how much hardware do we need? How faster the CPU is? How do we meet our deadlines? faster HW or cleverer SW? How minimise power consumption? Turn off unnecessary logic or reduce memory accesses? How de we design for upgradeability? Is it secure?

Testing on embedded system is complex: we can not separate the testing of an embedded computer from the machine in witch it is embedded.

Limited observability and controllability: it is difficult to see how it happen, there is none screen and keyboards, we need to watch the electrical signal values.

restricted development environments: much more limited than those available for PCs.

1.4 Attributes of Embedded Systems

Interfacing with the environment

Analog signals from sensors, typically using a voltage value that rappresents a physical value.

Concurrent and reactive behaviors: real-time constraints on responses

Embedded systems are not single-threaded: a lot of things can happen simultaneously as they respond to sequences and combinations of events in a timely manner. Embedded systems typically must perform multiple separate activities concurrently, especially when having several sensors.

Fault handling

Operate independently for long periods and handle likely faults **without crashing**

Diagnostics

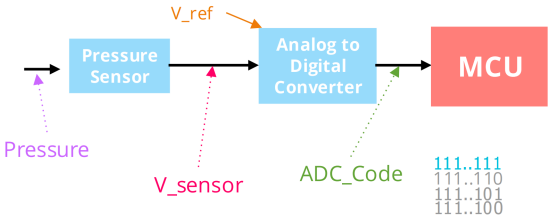
Embedded systems should help developers determine problems quickly.

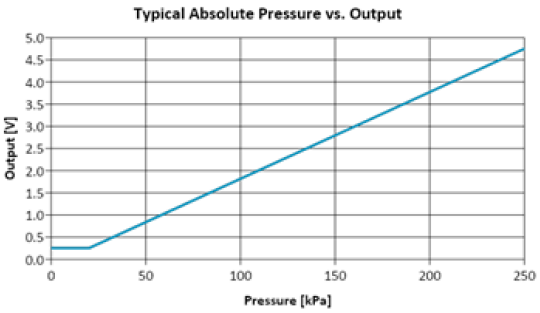
1.4.1 Example Analog Sensor - Depth Gauge

We want calculate how deep and underwater object is:

- A sensor detects pressure and generate a proportional output voltage **Vsensor**
- Hence an Analog to Digital converter calculate and transform the voltage into a digit

```
// Your Software
ADC_Code = adc_read();
V_sensor = ADC_code * V_ref / ADC_MASK;
Pressure_kPa = 250 * (V_sensor / V_supply + 0.04);
Depth_ft =33 * (Pressure_kPa - Atmos_Press_kPa) / 101.3;
//Actual relationship between depth and pressure
```



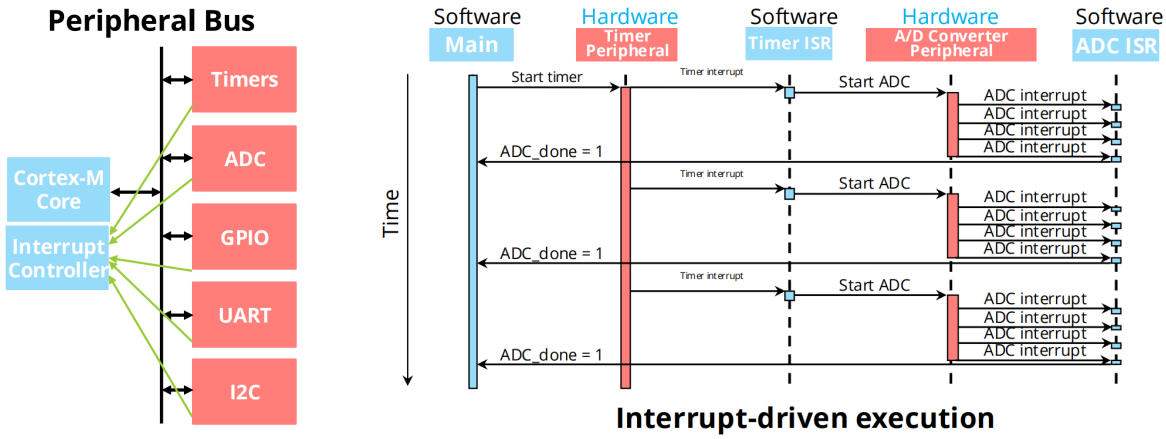


1.5 MCU Hardware & Software for Concurrency

In a microcontroller (MCU) several things happen **concurrently** while the CPU is executing instructions.

Specialized hardware peripherals add dedicated **concurrent processing**, like analog interfacing, timers, and detecting external signal events. Peripherals use interrupts to notify the CPU of events.

Embedded systems rely on both MCU hardware peripherals and software to get everything done concurrently on time



1.5.1 Embedded Software

Programmed in C rather than Java (smaller and faster code, so less expensive MCU). Some performance-critical code may be in **assembly language**.

Typically, no OS, but instead a simple scheduler (or even just interrupts + main code (foreground/background system). If OS is used, likely to be a lean **RTOS**

1.5.2 Hardware and Software Co-design Model

Commonly both the hardware and the software for embedded system are **developed in parallel**, this allow to optimize the overall product's design, functionality and manufacturing/development cost.

Push things to the software layer if functionality can be achieved in software, this reduce the overall hardware **complexity** and cost.

1.5.3 Functional vs. Non-functional Requirements

Embedded systems have **MANY non-functional requirements** (how the system should perform). Timing correctness is way more critical in embedded systems, especially in medical, transportation and military systems.

Non-functional requirements can be: the time required to compute an output, the system's size, weight, portability, power consumption, battery capacity, reliability and so on.

Functional requirements are user-centric, tangible and observable, and specify what the system must do.

1.6 Internet of Things (IoT)

An Internet-of-Things system is a networked embedded computing system. An IoT system is addressable via a network.

Some examples are smart buildings and home appliances, wearable devices, medical devices.

Networking is a **critical component** of an IoT system

1.6.1 Why IoT?

Items can have more functionality and become more intelligent

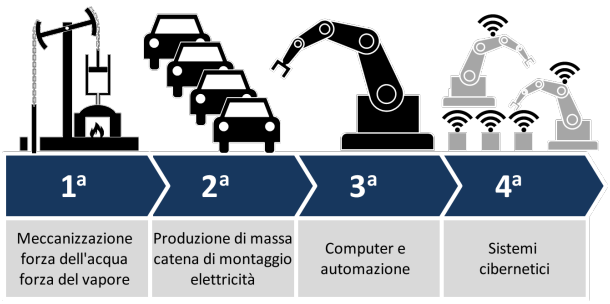
Items can be managed more easily

More information becomes available

1.6.2 Cyber-physical Systems

Combines physical devices with computers that control the device. The embedded computer is the cyber part that replaces mechanical controllers:

- more accurate and more sophisticated control
- monitors and controls the physical processes with feedback loops



1.7 Summary

An embedded system is built for a specific application. It has complex hardware and software components.

Embedded systems pose many design challenges: Functional, output as a function of input, and non-functional requirements, Deadlines, power, cost.

In real-time systems, **timing correctness** is just as important as functional or logical correctness

Chapter 2

Arm Cortex-M4 Processor and Its Architecture

2.1 What is an ARM architecture?

ARM stands for **Advanced RISC Machine**. It's a family of RISC-based processors, well-known for their power efficiency and use in mobile devices (smartphones and tablets), designed and licensed by Arm to a wide ecosystem of partners.

2.2 RISC VS CISC

Complex Instruction Set Computer (CISC):

This processor offers a variety of instructions that perform very complex tasks like string searching and a number of different instruction formats of varying lengths.

Examples of CISC processors are Intel processors.

Reduced Instruction Set Computer (RISC):

This processor offers fewer and simpler instructions that can be efficiently executed in a pipelined manner,generally uses load/store instruction sets, operates only on register and cannot operate directly on memory locations.

Overall, it's simplified hardware design and implementation which means that it will require less power to use.

What is an ARM Cortex processor? What does it offer us? What are they main components?

2.3 ARM Company

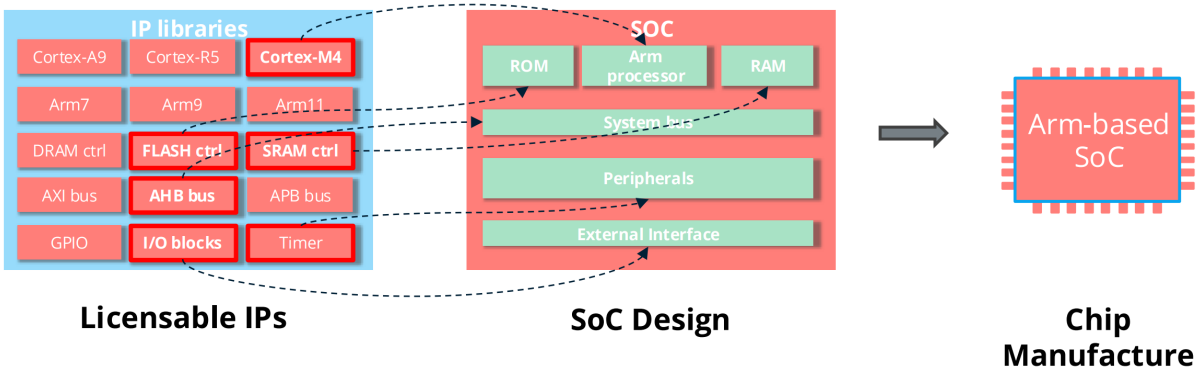
Arm is the company that designs Arm-based processor cores. It does not manufacture them, but licenses the designs to semiconductor partners which can add their own intellectual property (IP) on top of, which then can fabricate and sell to customers.

Two types of licenses:

- Microarchitectural License, which allows the vendor to use one of the various available Cortex licenses (RTL-level design of the processor).
- Architectural License, which allows the vendor to develop one's own microarchitecture based on ARM's ISA.

2.4 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



2.5 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.

2.6 Arm Cortex-M Series

The main features of a Cortex-M series processor are:

- Energy efficiency
  Low energy cost, which means longer battery life
- Smaller code
  The code is smaller which means less memory is required, which means less power is consumed and hardware costs are lower
- Lower silicon costs
  They occupy less physical space
- Ease of use and development
  Faster software development and reuse (lots of tools)
- Targets several embedded applications
  Like smart metering, human interface devices, automotive and industrial control systems, white goods (home appliances), consumer products and medical instrumentation

2.7 Cortex-M series processors ranked

The higher the number the better.

Cortex-M0 and Cortex M0+

Cortex-M0 and Cortex-M0+ are for **applications requiring minimal cost, power and space**. They're optimized for simple sensing and controlling.

Cortex-M3, Cortex-M4 and Cortex-M7

These are **designed for data intensive applications requiring higher performance**, like digital signal control applications.

Cortex-M4 and Cortex-M7 integrate Digital Signal Processing (**DSP**) and **accelerated floating point processing capability** for fast and power-efficient algorithm processing.

2.8 Architectures and Implementations

The architecture is the Instruction Set Architecture (ISA), which defines those characteristics and instructions that must be true for all implementations.

Implementations are the actual hardware implementations of the architecture, which can have varying clock speeds, different bus widths, different cache sizes and so on.

2.9 Arm Architectures vs.Arm Processors

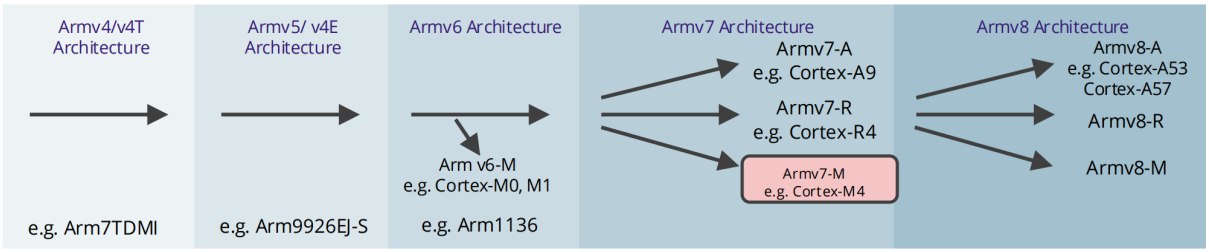
Arm architecture

The Arm architecture **describes the details of the instruction set**, the programmer's model, memory map.

Overtime, Arm architecture evolved to include architectural features that meet the growing demand for new functionality (example: AI), integrated security features, high performance and the needs of new and emerging markets.

Arm Processors

The Arm processor is the implementation of the Arm architecture.

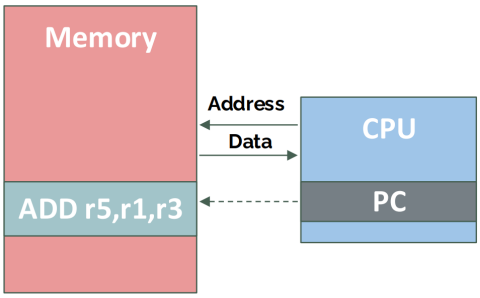


Processor	Arm Architecture	Core Architecture	Thumb	Thumb-2	Hardware Multiply	Hardware Divide	Saturated Math	DSP Extensions	Floating Point
Cortex-M0	Armv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M0+	Armv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M3	Armv7-M	Harvard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

2.10 Von Neumann Architecture

The memory holds both data and instructions. CPU **fetches** instructions from memory, **decodes** them and **executes** them.

The CPU register stores values used internally: program counter (PC), instruction register (IR), generalpurpose registers, etc.

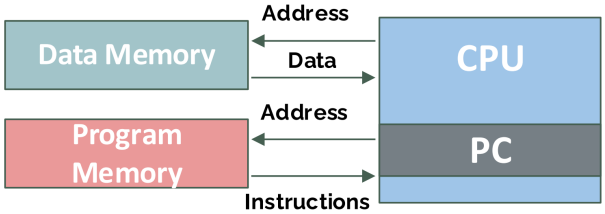


2.11 Harvard Architecture

There are two separate memories for data and program (instructions). Indeed the processor has two ports for the two memories, so they don't compete for a single port, this **allows two simultaneous memory fetches**.

The program counter (PC) always points to program memory (not data memory) which means it cannot use self-modifying code (secure).

Most DSPs are Harvard architectures. The separation of program and data memories provides **higher performance**.



2.12 Cortex-M4 Processor Overview

The Cortex-M4 processor is designed with a large variety of highly efficient signal processing features (example: single-cycle multiply accumulate instructions).

The **Multiply-Accumulate** (MAC) Instruction is an important and expensive operation mainly employed in digital signal processing and video/graphics applications, machine learning included, that computes the product of two numbers and adds that product to an accumulator.

It also features low power consumption, which means a longer battery life on the device that uses it, a critical non-functional requirement in mobile products.

Furthermore it offers enhanced determinism: critical tasks and interrupt routines can be served quickly in a known number of cycles. This is crucial for embedded system applications.

2.12.1 Cortex-M4 Processor Features

32-bit reduced instruction set computing (RISC) processor, **Harvard architecture**, so separate data bus and instruction bus and it has 3-stage + **branch speculation pipeline**.

It supports sleep modes: Features wake-up interrupts, with wait for interrupt (WFI) and wait for event (WFE) instructions.

Enhanced instructions Hardware divid, MAC, and so on.

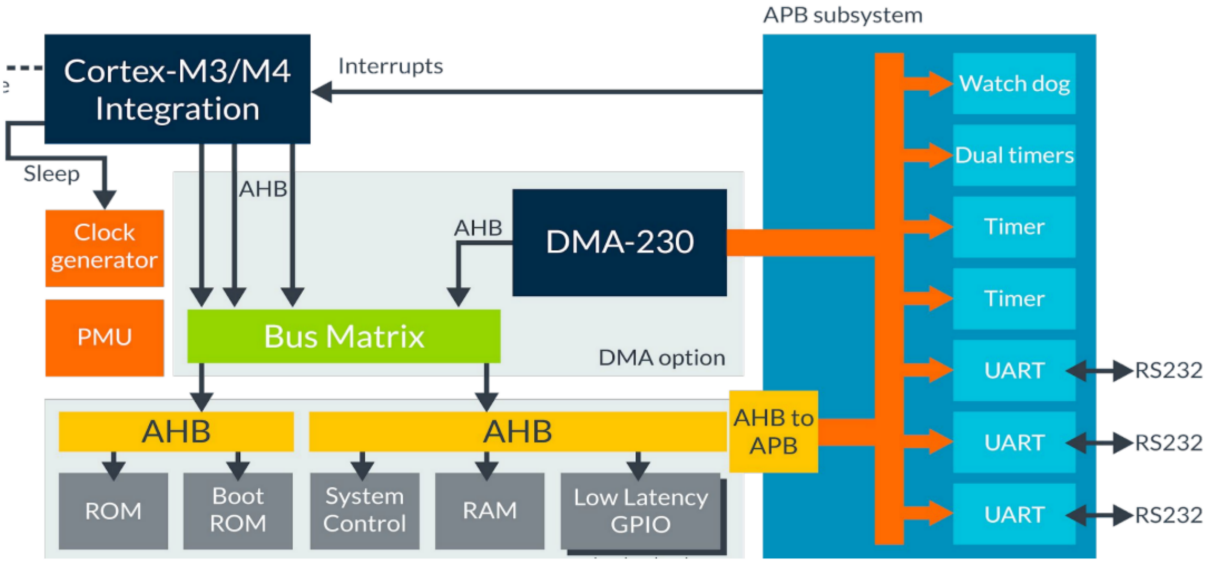
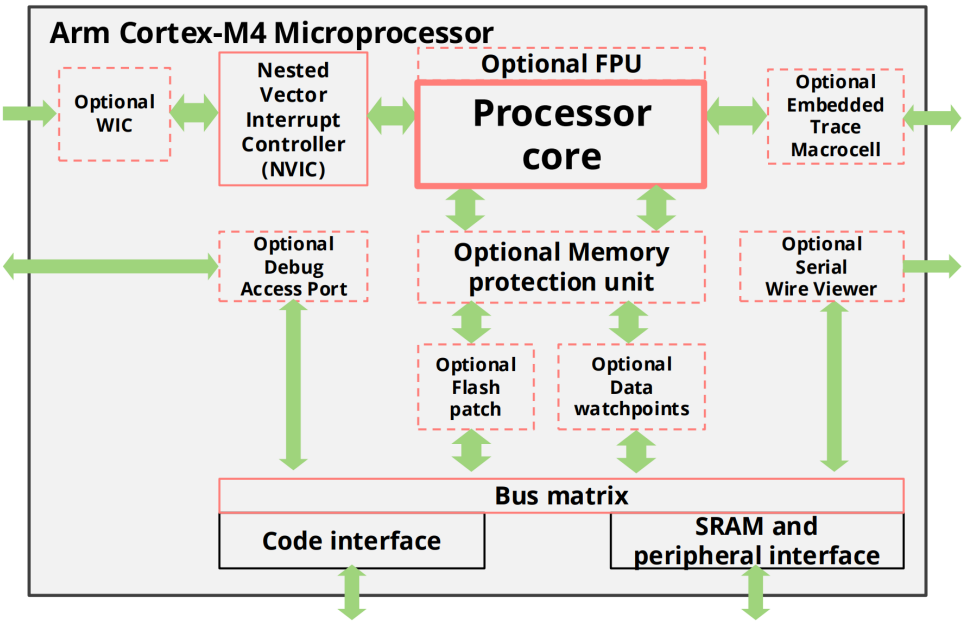


Figure 2.1: Arm M4-MCU Architecture

2.12.2 Cortex-M4 Block Diagram

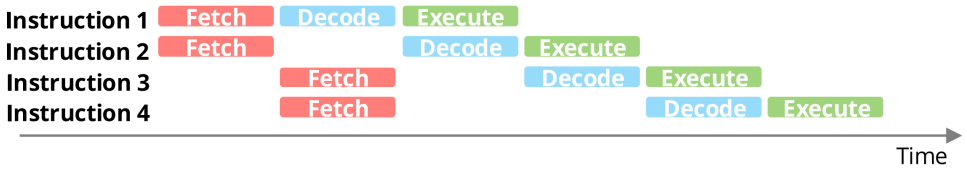


Here are the components as described:

Processor core

The processor core contains internal registers, the ALU, data path and some control logic.

It has a three-stage pipeline: fetch, decode, execution. Some instructions may take multiple cycles to execute in which case the pipeline will be stalled. Also the pipeline speculatively prefetches instructions from branch target addresses.



Nested vectored interrupt controller (NVIC)

It 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)

It is used to protect memory content. It makes some memory regions read-only and prevents user applications from accessing each other.

Debug subsystem - Embedded Trace Macrocell and Serial Wire Viewer

It handles debug control, program breakpoints, and data watchpoints.

When a debug event occurs it can put the processor core in a halted state, so that developers can analyse the status of the processor like register values and flags at that point.

Bus matrix/Bus interconnect

It provides data transfer management among hardware components and peripherals.

It also allows data transfer to take place on different buses simultaneously:

- Advanced High-performance Bus (AHB)-Lite bus for high bandwidth peripherals
- Advanced Peripheral Bus (APB) interface. Low-power, meant for peripherals such as timers, interrupt controllers, UARTs, I/O

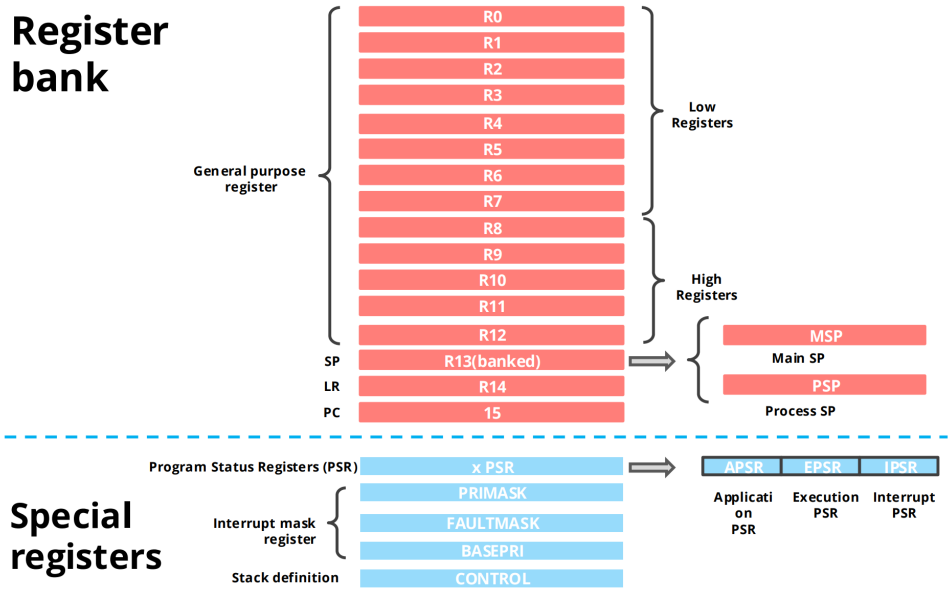
The bus matrix may include bus bridges (example: AHB-to-APB bus bridge) to connect different buses into a network using a single global memory space.

2.13 Programming Model

The programming/programmer model is the set of registers available for use by programs.

The CPU has many other special purpose registers that are used for internal operations, which are unavailable to programmers.

2.13.1 Arm Cortex-M4 Processor Registers



The processor registers store and process temporary data within the processor core quickly.

They follow a load-store architecture, in which to process memory data, they first have to be loaded from memory to registers, processed inside the processor core using register data only, and then written back to memory if needed.

The Cortex-M4's register bank is composed of 16 X 32-bit registers (R0-R12 are general-purpose, others are the stack pointer (R13), the link register (R14) and the program counter (R15)).

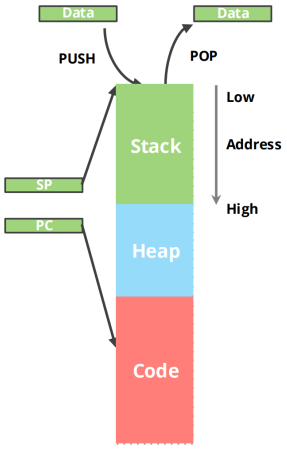
Stack Pointer

The stack pointer (SP) records the current address of the stack and is used for saving context while switching between tasks.

The Cortex-M4 has two SPs: the main SP, used in applications that require privileged access like the OS kernel, and the process SP which is used in base-level application code.

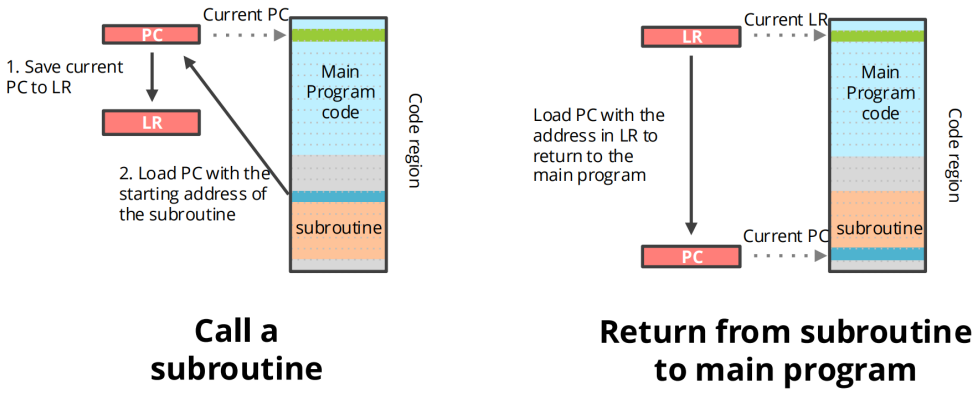
Program Counter

The program counter (PC) records the address of the current instruction code and is automatically incremented.



Linik Register

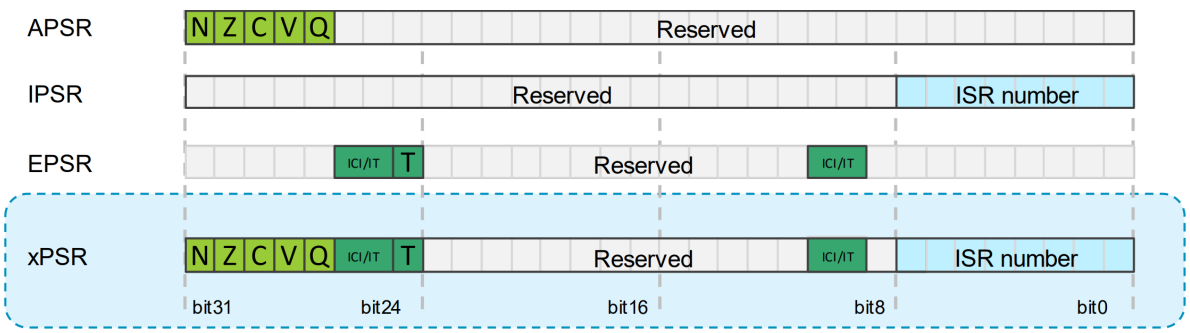
The link register (LR) stores the return address of a subroutine or function call. The PC will load the value from the LR after a function is finished.



2.13.2 Cortex-M4 Registers (Special Registers)

Then there are special registers like the interrupt mask register used to enable/disable interrupts, the program status registers (PSR) to understand what kind of exceptions, interruptions and results occurred.

The **xPSR**, **combined program status register**, provides information about program execution and Arithmetic Logic Unit (ALU) flags. It combines the Application PSR, Interrupt PSR and Execution PSR.



APSR - Application Program Status Register

- N: negative flag
- Z: zero flag
- C: carry flag
- V: overflow flag
- Q: stick saturation flag

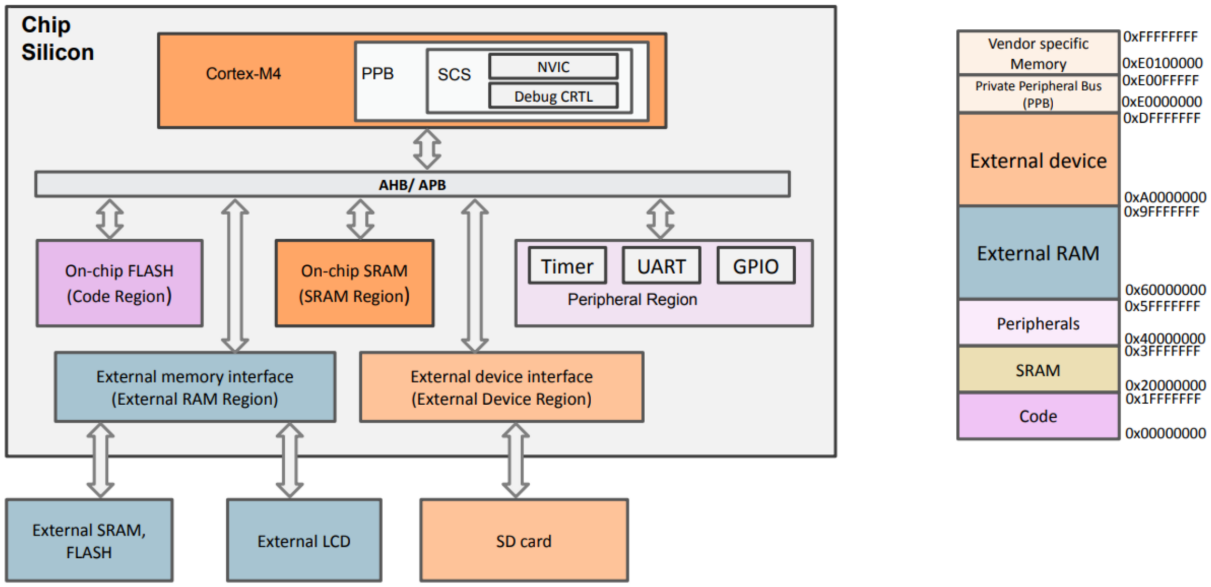
2.14 Arm Cortex-M4 Memory Map

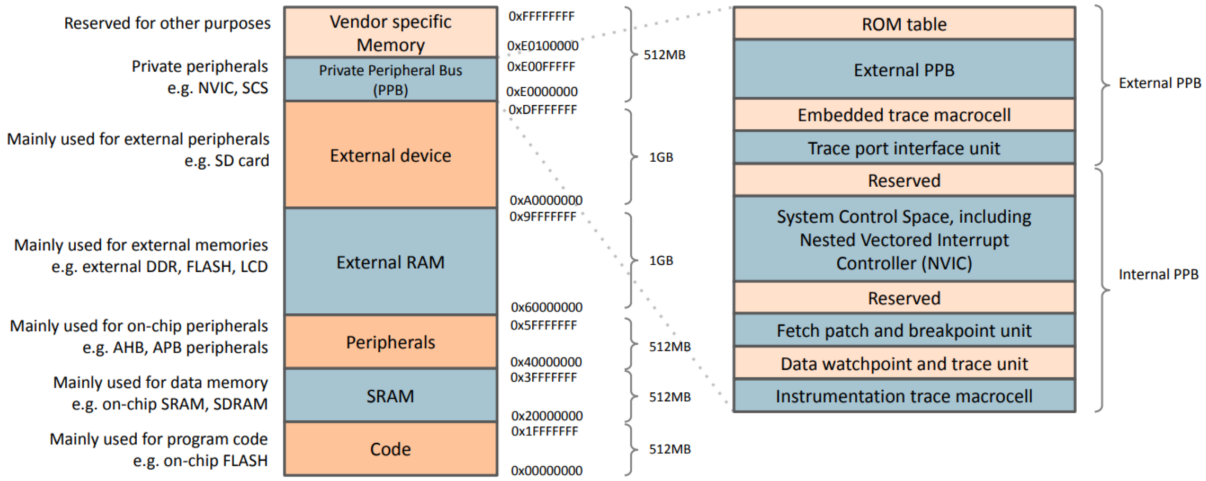
The **memory map** describes the organization of the **processor's address space**.

ARM memory is **byte-addressable** using **32-bit** addresses. The Cortex-M4 processor has **4GB of memory address space**.

The 4GB memory address space is architecturally defined with a number of regions, each one designed for particular recommended use cases, making it easy for a software programmer to port between different devices.

Memory map can also be flexibly defined by the user, apart from some fixed memory addresses such as internal private peripheral bus.





Code Region It is primarily used to store program code, can also be used for data memory. On-chip memory, such as on-chip FLASH.

SRAM Region It is primarily used to store data, such as heaps and stacks. It can also be used to store program code.

Peripheral Region It is primarily used for AHB/APB peripherals.

External RAM region It is primarily used to store large data blocks or memory caches. It's an off-chip memory, slower than onchip SRAM region.

External device region Used to map external, off-chip devices like SD cards.

Private Peripheral Bus (PPB) Provides access to internal and external processor resources.

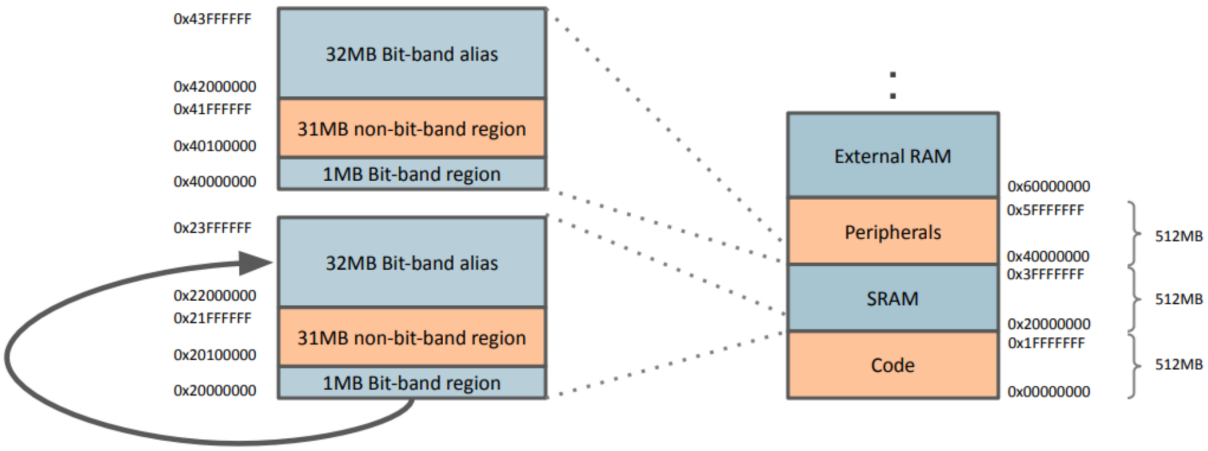
2.15 Bit-band Operations

One interesting property of Arm architectures is that they allow bit-band operations.

Bit-band operations allow a single load/store operation to access a single bit in the memory, without having to access 32 bits at once just to change a single one.

It works by directly writing a single bit (0 or 1) to the "**bit-band alias address**" of the data.

Bit-band Alias Address SRAM and Peripheral regions include **bit band** and **bit band alias** areas.



Each bit of the data is one-to-one mapped to the bit-band alias address. Each bit has an address multiple of 4.

1 byte = 8 bits = 4x8 = 32 addresses in memory space.

2.15.1 Benefits of bit-band operations

Bit-band operations allow for **faster bit operations** and **fewer instructions**. It's an **atomic operation**, so it helps prevent data conflict hazards.

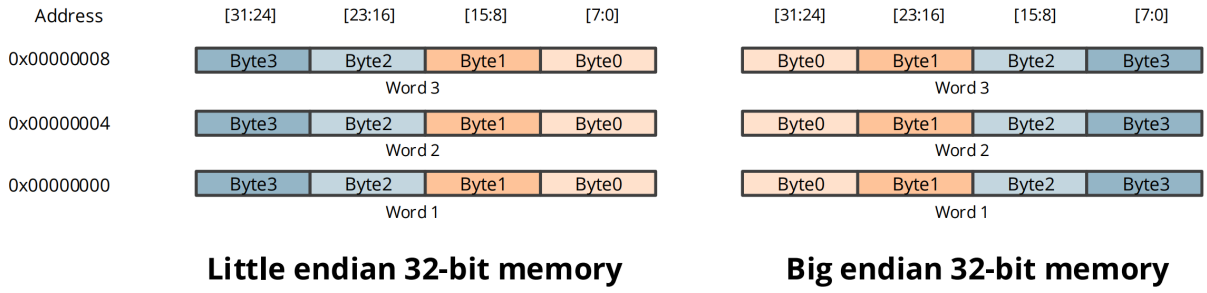
2.16 Cortex-M4 Endianness

Endian refers to the order of bytes stored in memory.

Little endian: lowest byte of a word-size data is stored in bit 0 to bit 7

Big endian: lowest byte of a word-size data is stored in bit 24 to bit 31

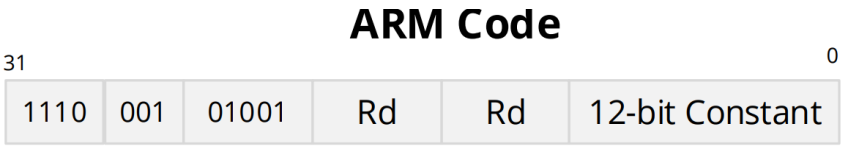
Cortex-M4 supports both little endian and big endian. However, endianness only exists in the hardware level.



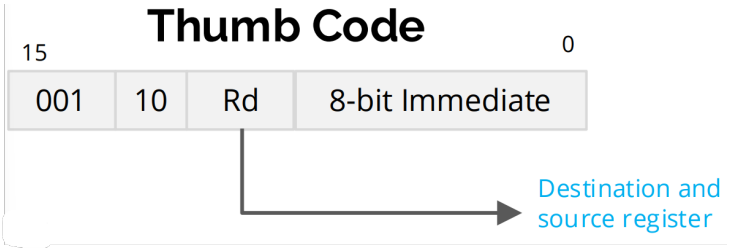
2.17 Arm and Thumb Instruction Set

The early Arm instruction set was a 32-bit instruction set called the Arm instructions. It's powerful and performs well but it required a larger program memory and a larger power consumption.

Example: ADDS Rd, Rd, #Constant



Thumb Code ADD Rd, #Constant



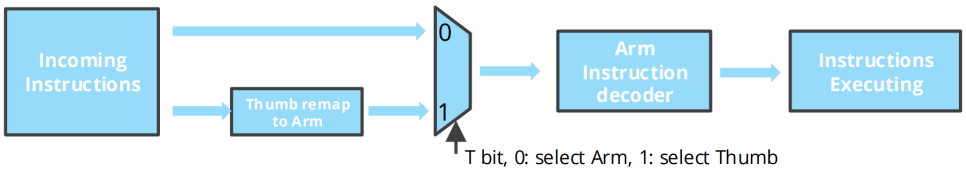
The instruction does the same operation but encoded using **less number of bits**!

Later on Arm introduced the Thumb-1 instruction set. It had 16-bit instructions as it was a subset of the Arm instruction set.

Code size is reduced by -30% but performance is also reduced by -20%.

A multiplexer is used to switch between the two sets, which requires a switching overhead. This is so one can benefit from 32-bit Arm's high performance and 16-bit Thumb-1's high code density (reduced program code size, Sometimes leads to an increased number of instructions, thus making thumb slower to execute because fetching instructions is slower than executing instructions).

The processor and the compiler manage this automatically.



The Thumb-2 instruction set consists of both 32-bit Thumb instructions and the original 16-bit Thumb-1 instruction sets. Compared to the 32-bit Arm instruction set, code size is reduced by -26% with similar performance.

The Thumb-2 instruction set also covers almost all functionality of the Arm instruction set and most instructions are **unconditional**, whereas almost all Arm instructions are conditional.

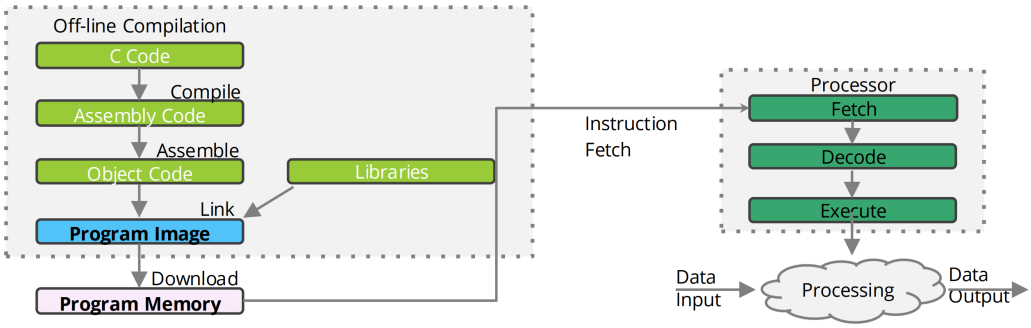
Chapter 3

Program Development

3.1 Typical Program-generation 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



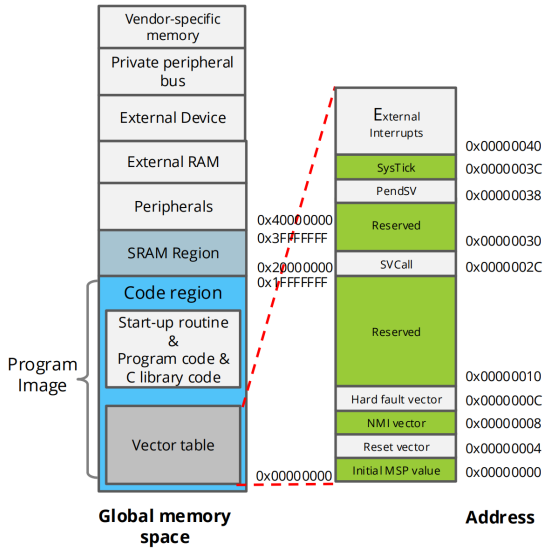
Typical program-generation flow

3.2 Cortex-M4 Program Image

A program image (or executable file) is a piece of fully integrated code that is ready to execute.

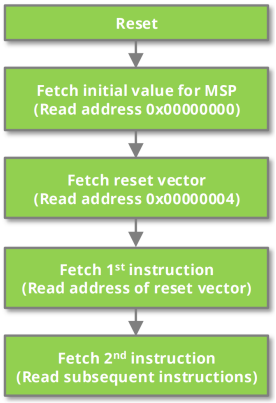
The program image includes :

- Program code
- C library code -> inserted by the linker
- Vector table : contains the starting addresses of interrupt vectors and the MSP (**main stack point**)
- C startup code : used to set up data memory and initialize global variables



3.2.1 Systems Initialization

When the system resets:

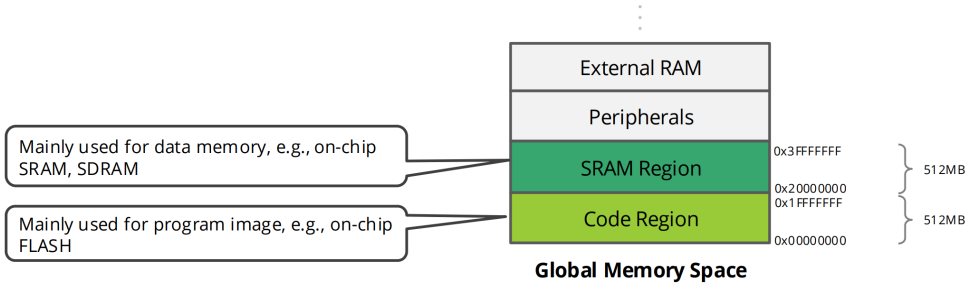


3.3 Program Image in Global Memory

The **program image** is stored in the code region in global memory and occupies at most 511 MB of data.

It is usually implemented on **non-volatile memory**, such as on-chip flash memory.

The executable image contains initialized data and uninitialized data. Normally separated from program data, which is allocated in the SRAM region (or data region)

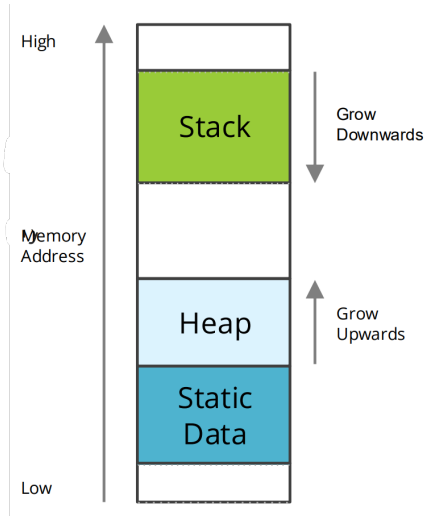


Normally separated from program data, which is allocated in the SRAM region (or data region)

3.4 How is Data Stored in RAM?

Typically, the data can be divided into three sections: **static data**, **stack**, and **heap**

- **Static data**: global variables and static variables
- **Stack**: local variables, parameter passing in function calls, registers saving during exceptions, etc.
- **Heap**: pieces of memory spaces that are dynamically reserved by function calls, such as 'alloc()' and 'malloc ()'. (Generally not preferred in embedded systems)



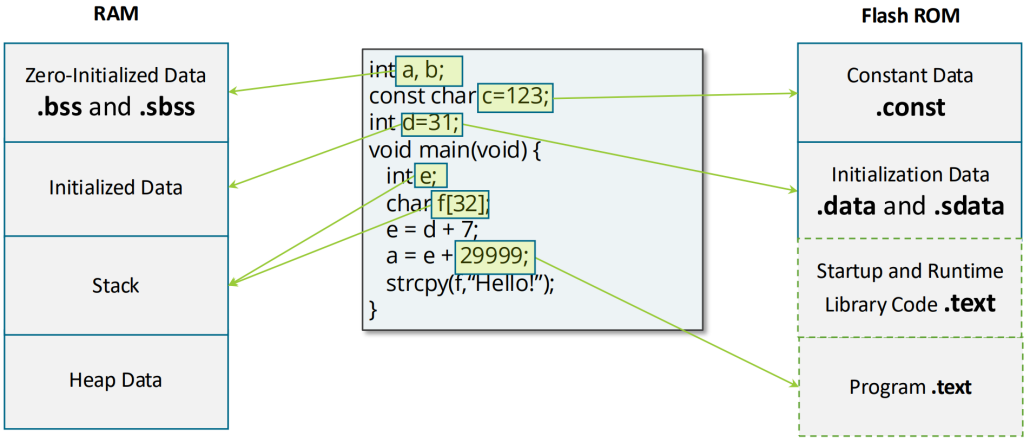
3.4.1 How to decide on the use of memory to store data?

Information do not change: Put it in read-only nonvolatile memory, e.g. Instructions, Constant strings, Constant operands, Initialization values...

Information change: Put it in read/write memory, e.g. Variables, Intermediate computations, Return address.

```
int a, b;
const char c=123; //heap
int d=31;

void main(void) {
    int e;
    char f[32];
    e = d + 7;
    a = e + 29999; //immediate value
    strcpy(f,"Hello!");
}
```

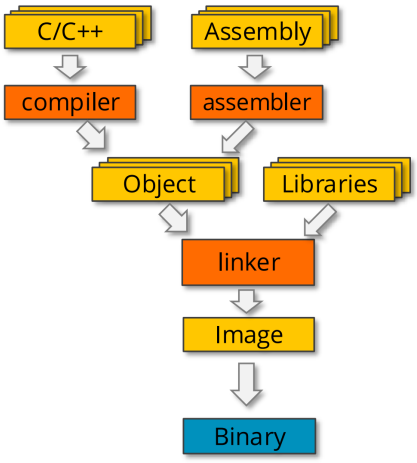


Executable image contains **initialized** data and **uninitialized** data sections:

- .data and .sdata: contain the initial values for the global and static variables
- .bss and .sbss: uninitialized (or zero initialized) data sections (their content is empty)
- .const: constant data is part of the.const section which is read-only.

3.4.2 C Run-Time Start-Up Module

After reset, MCU must: Initialize hardware... and Initialize C or C++ runtime environment: Set up heap memory, Initialize variables.



3.5 load the program mage into the target (microcontroller)

To load a program image, we need to transfer the executable image from the host to the target (for example through an USB Cable).

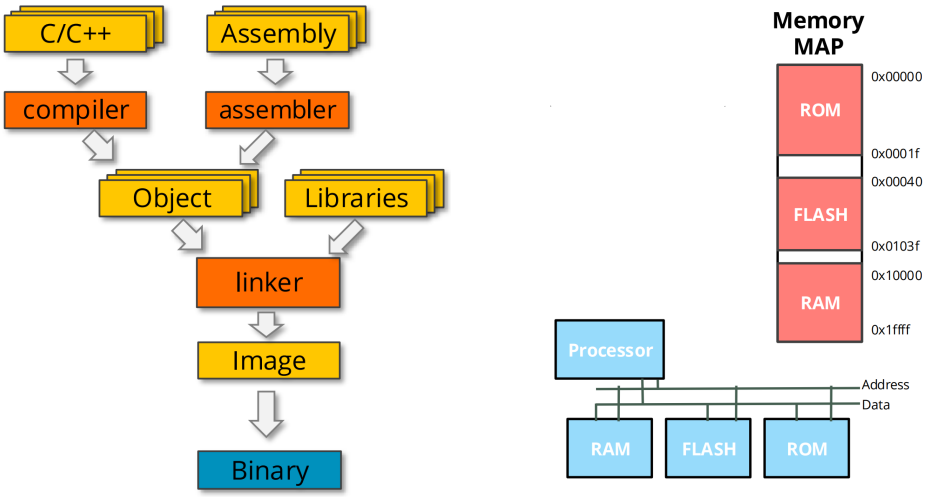
There are 3 steps :

1. Creating a program image in the host
2. Loading the program image in the microcontroller using special equipment like JTAG
3. Executing the program in the microcontroller

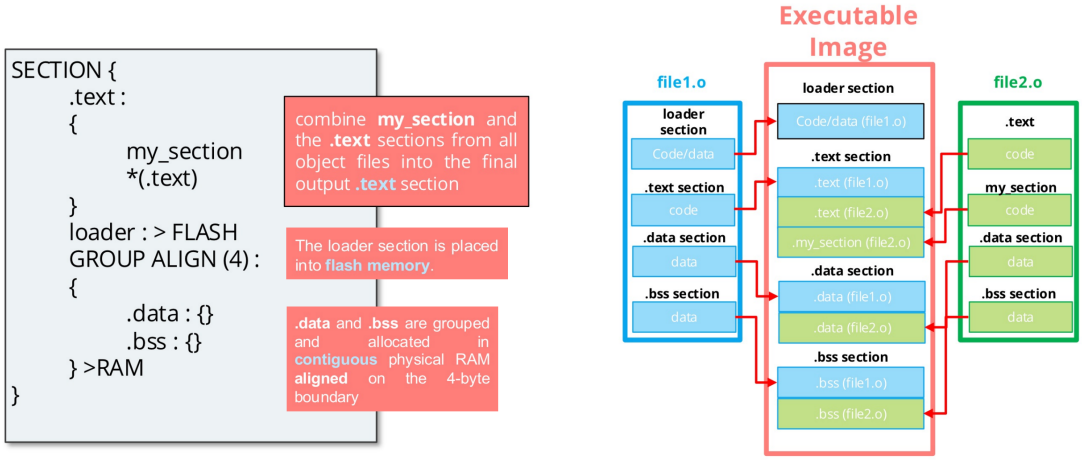
Step one: The linker create a single executable image for the target embedded system.

The linker directives control how the linker combines the sections and allocates the segments into the target system. Two of the more common linker directives are:

- **MEMORY** : used to describe the target system's memory map, types of physical memory and address range occupied by each physical memory block



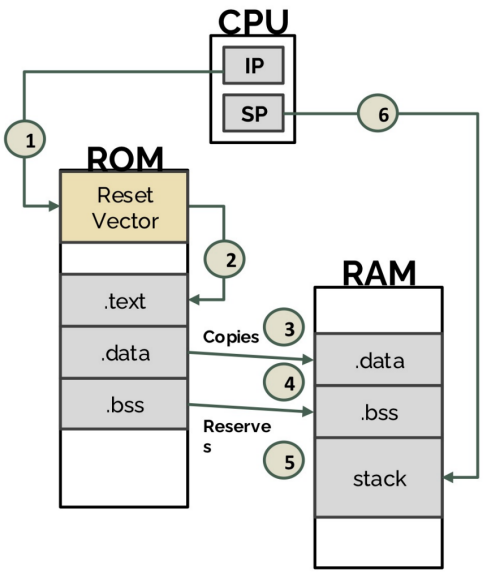
- **SECTION** tells the linker:
  - which input sections are to be combined into which output section
  - which input sections are grouped together and allocated in contiguous memory
  - where to place each section



Step two: After powered on, the target executes code from a predefined address offset. The code in this memory location is called reset vector, which is typically a jump instruction into the bootstrap code.

During booting, two CPU registers are concerned:

- The instruction pointer register points to the next instruction (IP)
- The stack pointer points to the next free location in stack (SP)



(1) IP is hardwired to execute the first instruction in memory
- the reset vector

(2) The reset vector jumps to the first instruction of the .text section in ROM

(3) The .data section of the boot image is copied into RAM
- it is both readable and writable.

(4) Space is reserved in RAM for the .bss section
- Nothing to transfer because the content for the .bss section is empty.

(5) Stack space is reserved in RAM.

(6) The SP register is set to point to the beginning of the newly created stack.

At this point, the boot completes:
- The CPU continues to execute the code in the .text section

Chapter 4

IO Operations, Interrupts, and Their Impact on Embedded Systems

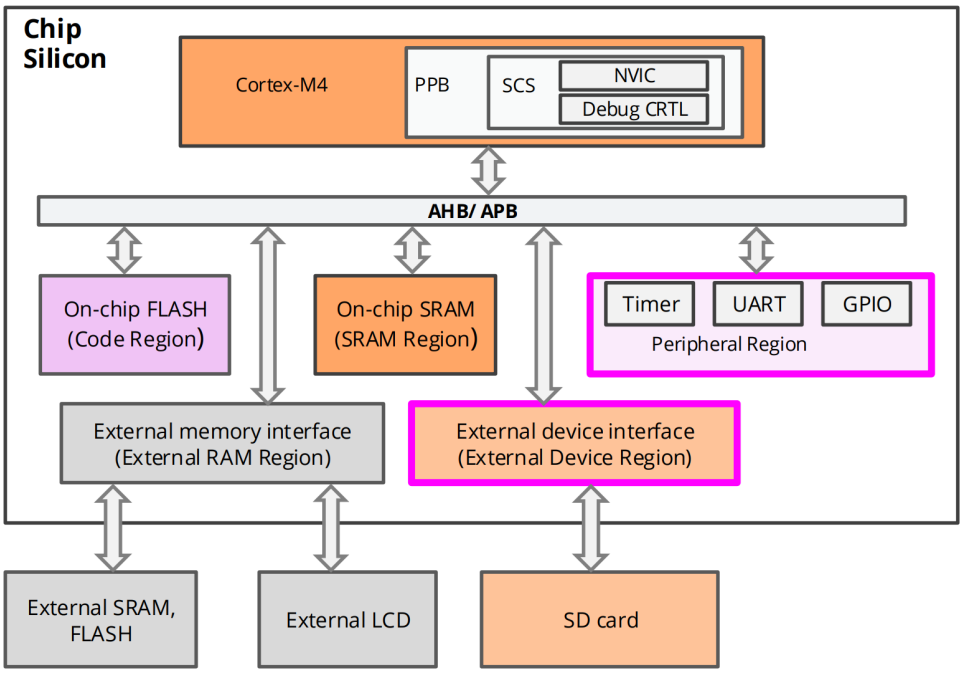


Figure 4.1: Input and Output Devices

4.1 Current state of Art

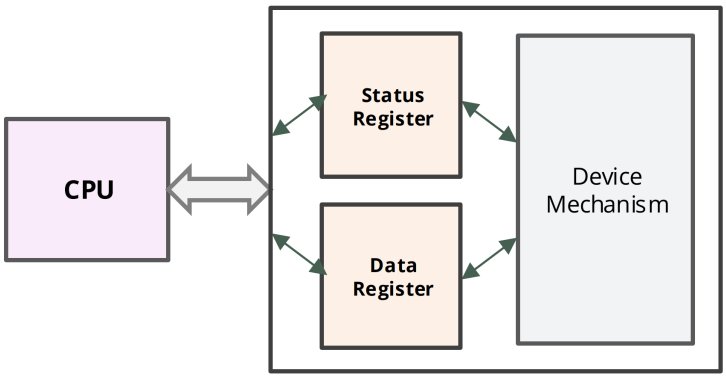
Currently Arm architecture dominates the embedded market. But there is a trend of switching to RISC-V.

The benefits is that RISC-V is open-source and more customisable. Because it's open-source this means it's cheaper to develop with as there are no licensing fees or royalties to pay. ARM is dominant in the embedded device market but RISC-V is getting more and more use in research and niche markets, allowing reduced vendor lock-in and greater innovation.

ARM processors seem to be more **power-efficient** for now. **RISC-V's ecosystem** is growing but it **is not as mature and rich as ARM's.**

4.2 Programming Input and Output

The CPU talks to the device by reading and writing device registers.



Devices typically have several registers:

- Data registers: hold data values, can be **readable or writable**;
- Status registers: provide information about the device's operation, can be **read-only**

4.2.1 How to access the registers

There are two way to access those registers:

I/O instructions: special instructions for input (read) and output (write) (in and out in case of Intel x86), requires a separate address space for each I/O device

Memory-mapped I/O: the registers are mapped to the memory space of the device, meaning that we will need to manipulate the dedicated registers via normal memory read and write to communicate with devices. Very simple way.

We will use memory-mapped I/O throughout the course.

Example: Memory-mapped I/O on ARM

Define location for device and provide Read/Write code:

```
DEV1 EQU 0x1000
LDR r1,#DEV1 ; set up device addres
LDR r0,[r1] ; read DEV1
LDR r0,#8 ; set up value to write
STR r0,[r1] ; write value to device
```

Example: write I/O in C

We can use pointers to manipulate the addresses of I/O devices

```
int read(char *location) {
    return *location;
}
void write(char *location, char newval) {
    (*location) = newval;
}
```

To read a device register we can write:

```
#define DEV1 0x1000
...
dev_status = read(DEV1); /* read device register */
```

To write to the status register, we can use the following code:

```
write(DEV1,8); /* write 8 to device register */
```

4.3 Busy-Wait I/O

Devices are typically slower than the CPU, so they may require many cycles to complete an operation.

The Busy-Wait, a.k.a. polling, is a technique in which the CPU keeps asking the I/O device if it has finished the operation by checking its status register.

Example: Busy-Wait I/O

We need to write a sequence of characters to an output device. The device has two registers, one for the character to be written and a status register.

The status register's value is 1 when the device is busy writing and 0 when the write has completed.

```
#define OUT_CHAR 0x1000 /* output device character register */
#define OUT_STATUS 0x1001 /* output device status register */
...
char *mystring = "Hello, world." /* string to write */
char *current_char; /* pointer to the current position in string */
current_char = mystring; /* point to the head of string */
while (*current_char != '\0') { /* until null character */
    write(OUT_CHAR,*current_char); /* send character to device */
    while (read(OUT_STATUS) != 0); /* keep checking status */
    current_char++; /* update character pointer */
}
```

As shown in the example, we write one character each then check if the device has finished writing to send the next one.

When a new character has been read: the input device sets its status register to 1, we must set it back to 0 so that the device is ready to read another character

To start writing: first set the output status register to 1, then wait for it to return to 0.

Example 2: Busy-Wait I/O

The following example showcases a simple application involving an input device and an output device connected to the same CPU.

```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001
#define OUT_DATA 0x1100
#define OUT_STATUS 0x1101
...
while (TRUE) { /* perform operation forever */
/* read a character into achar */

while (read(IN_STATUS) == 0); /* wait until ready */
    achar = (char)read(IN_DATA); /* read the character */
    write(OUT_DATA,achar); /* write achar */
    write(OUT_STATUS,1); /* turn on device */
    while (read(OUT_STATUS) != 0); /* wait until done */
}
```

The code is supposed to get input from a keyboard and then show it onto an LCD display:

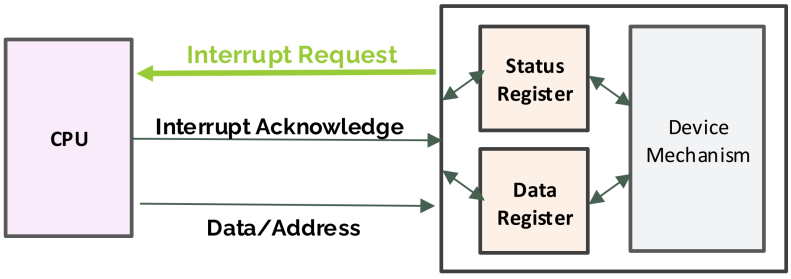
- It keeps waiting until the input device's status is set to ready, indicating a new character
- It reads the new character and stores it in memory
- It writes the new character in the data register of the output device
- It keeps waiting until the output device's status is set to ready, indicating a new character can be written

4.3.1 Cons of using Busy-Wait I/O

The bad side of busy-wait is that we waste too many CPU cycles (extremely inefficient), which is not ideal when there are many I/O devices connected.

The most effective way is not to check all the time, but to be notified automatically. This is why the interrupt mechanism is used, allowing devices to signal the CPU and force execution of a particular piece of code (interrupt handler).

4.4 Interrupt: How it Works

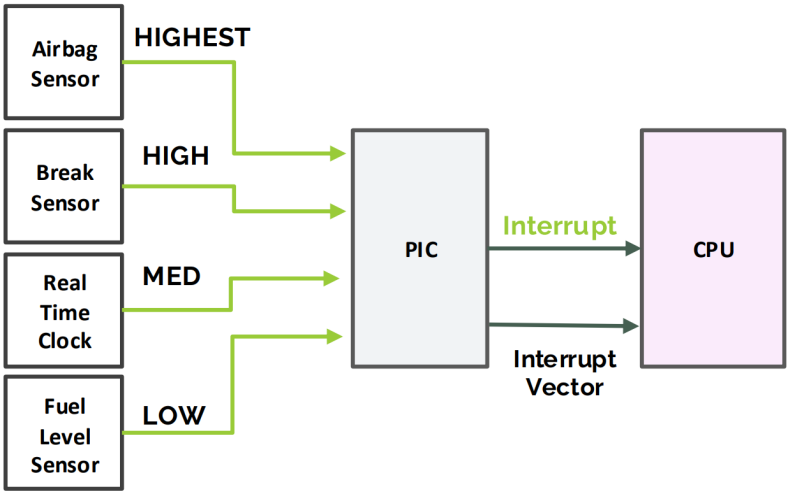


An interrupt works with the following steps (high-level overview):

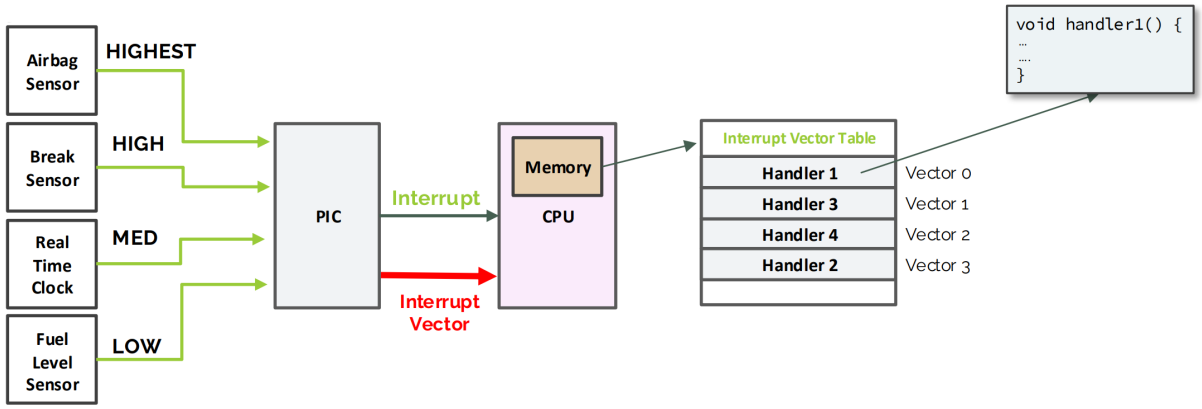
1. the I/O device decides when to interrupt, like when its status register goes into the ready state
2. the I/O device wants service from the CPU, so it sends an interrupt request signal
3. the CPU checks the interrupt request (IRQ) line at every instruction
4. the CPU is ready to handle the I/O device's request, so it sends an interrupt acknowledge signal
5. the CPU saves the current value of the program counter
6. the CPU then changes the program counter to point to the device's interrupt handler
7. the starting addresses of the interrupt handlers are stored in a table called Interrupt Vector Table/Interrupt Handler Table stored in the CPU's memory
8. after execution, the interrupt handler executes a special instruction called IRET or RETI to signal it finished execution
9. the CPU then switches back to the previous program counter to resume its normal execution

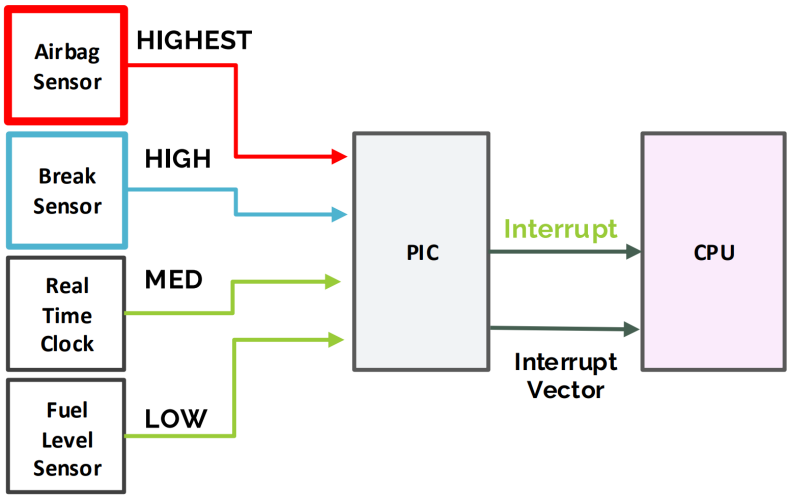
4.4.1 Priorities and Vectors

As most systems have more than I/O device, multiple devices can interrupt even at the same time. In this case, the interrupts are ran in order of priority via a **Programmable Interrupt Controller** (**PIC**).

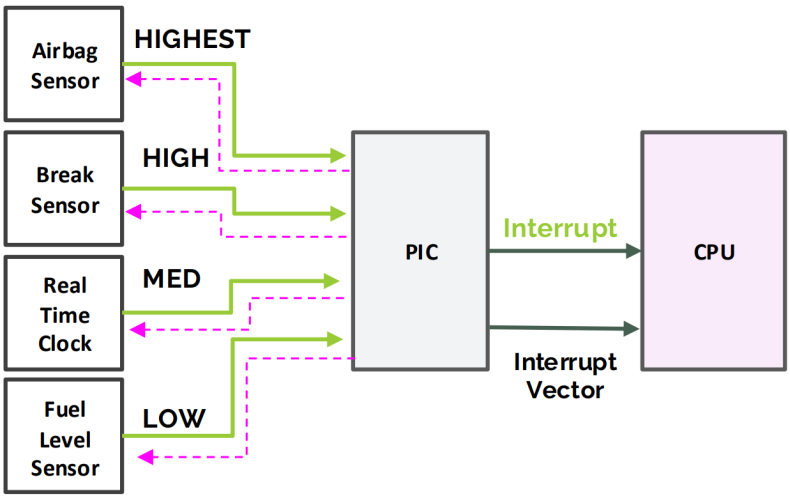


In the ARM architecture it is called the **Nested Vector Interrupt Controller** (**NVIC**). Also it is the interrupt device that specifies its interrupt handler (the index in the Interrupt Vector Table).



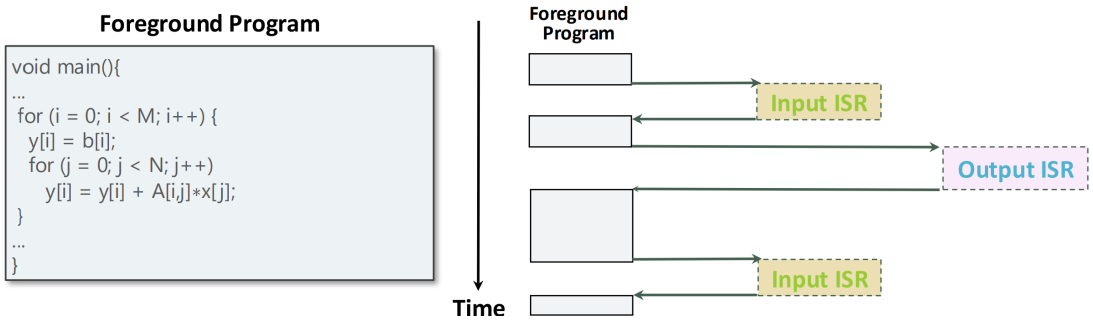


The interrupt devices are connected to the PIC, which tells them if their interrupt handlers are being executed or are in line through interrupt acknowledge lines. If they are in line, they will also know their own number of priority.



Nested interrupts must be prevented (ISR being ran while another ISR is being ran already).

NOTE: Interrupt handler = ISR = Interrupt Service Routine



4.5 Interrupt Masking

Interrupt masking is a technique used to control and prioritize interrupt handling. It is typically used to ensure that high-priority interrupts are handled promptly, especially in real-time and safety-critical systems where specific interrupts must be handled immediately to meet timing or safety requirements.

"Masking" means temporarily disabling an interrupt. This means that as soon as the CPU acknowledges a higher priority interrupt, it immediately pauses the current ISR, executes the higher priority interrupt's ISR before resuming the previous one.

The **PIC** is used to facilitate interrupt masking, via registers or configuration settings to enable or disable specific interrupt sources.

4.6 Non-Maskable Interrupt

Non-maskable interrupt (NMI) is an interrupt that cannot be masked so it will be executed 100without getting turned off. It is usually reserved for interrupts caused by power failures to save critical state in nonvolatile memory, turn off I/O devices...

NMIs are device specific.

4.7 Interrupt vs Exceptions

While **both do the same thing** (interrupt the current flow of the program), interrupt is meant when it's done by a **hardware device**, while an exception is done by the **software**.

4.7.1 Example #1: interrupt

The following example showcases the same program from the previous example, now using interrupts:

```
/* get a character and put in global (called when IN_STATUS is 1) */

void input_handler() {
    achar = read(IN_DATA); /* get character */
    gotchar = TRUE; /* signal to main program */
    write(IN_STATUS,0); /* reset status to initiate next transfer */
}

/* react to character being sent (called when OUT_STATUS is 0) */
void output_handler() {
    /* don't have to do anything */
}
```

The `input_handler` function does the following:

- stores the new character, taken from the input device's data register, into achar
- turns gotchar to true to signal to the main program that there's a new character
- resets the input device's status register to 0, ready state

All input device operations are done in the interrupt handler.

Mainprogram:

```
main() {
    while (TRUE) { /* read then write forever */
        if (gotchar){ /* write a character */
            write(OUT_DATA,achar); /* put character in device */
            write(OUT_STATUS,1); /* set status to initiate write */
            gotchar = FALSE; /* reset flag */
        }
    }
}
```

1. The gotchar variable is checked if true
2. Once it is, the character is written in the output device's data register and the output device's status register is set to 1, signaling it to initiate write
3. Then resets the gotchar variable to false as it waits for a new character

A few issues in this example:

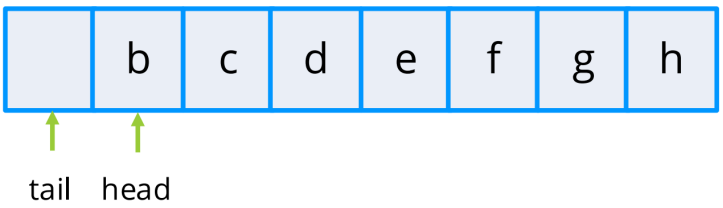
output device is not waited for when it finishes writing

character can be overwritten before it is written in the output device's data register

This example is used to just show off how interrupts work.

4.7.2 Example #2: Interrupt

The following example showcases the previous program but more sophisticated and improved thanks to the use of a **wraparound/circular buffer** (data structure) to hold the new characters, which also fixes the issues of the previous example.



Here's an implementation of the circular buffer:

```
#define BUF_SIZE 8
char io_buf[BUF_SIZE]; /* character buffer */
int buf_head = 0, buf_tail = 0; /* current position in buffer */
int error = 0; /* set to 1 if buffer ever overflows */

int empty_buffer() { /* returns TRUE if buffer is empty */
    return buf_head == buf_tail;
}

int full_buffer() { /* returns TRUE if buffer is full */
    return (buf_tail+1) % BUF_SIZE == buf_head ;
}

int nchars() { /* returns the number of characters in the buffer */
    if (buf_head >= buf_tail)
        return buf_head - buf_tail;
    else
        return BUF_SIZE - buf_tail - buf_head;
}

void add_char(char achar) { /* add a character to the buffer head */
    io_buf[buf_tail++] = achar;
    /* check pointer */
```

```
    if (buf_tail == BUF_SIZE)
        buf_tail = 0;
}
char remove_char() { /* take a character from the buffer head */
    char achar;
    achar = io_buf[buf_head++];
    /* check pointer */
    if (buf_head == BUF_SIZE)
        buf_head = 0;
    return achar;
}
```

Here are the new input/output handlers, making use of the circular buffer:

```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001

/* get a character (called when IN_STATUS is 1) */
void input_handler() {
    char achar;

    if (full_buffer()) /* error */
        error = 1;
    else { /* read the character and update pointer */
        achar = read(IN_DATA); /* read character */
        add_char(achar); /* add to queue */
    }
    write(IN_STATUS,0); /* set status register back to 0 */

    /* if buffer was empty, start a new output transaction */
    if (nchars() == 1) { /* buffer had been empty until this interrupt */
        write(OUT_DATA,remove_char()); /* send character */
        write(OUT_STATUS,1); /* turn device on */
    }
}
```

```
#define OUT_DATA 0x1100
#define OUT_STATUS 0x1101

/* react to character being sent (called when OUT_STATUS is 0) */
void output_handler() {
    if (!empty_buffer()) { /* start a new character */
        write(OUT_DATA,remove_char());/* send character */
        write(OUT_STATUS,1); /* turn device on */
    }
}
```

The input handler writes the data to the output device when the buffer is empty.

When the buffer is not empty, the output handler keeps writing data to the output device while emptying the buffer.

4.7.3 Cons of Interrupts

- Interrupts increase the program's complexity
- Finding bugs is difficult because interrupts are triggered by hardware devices
- Concurrency with interrupt handlers can cause bugs if they don't save/restore CPU registers that they modify during executions, causing variables in the foreground program to change mysteriously

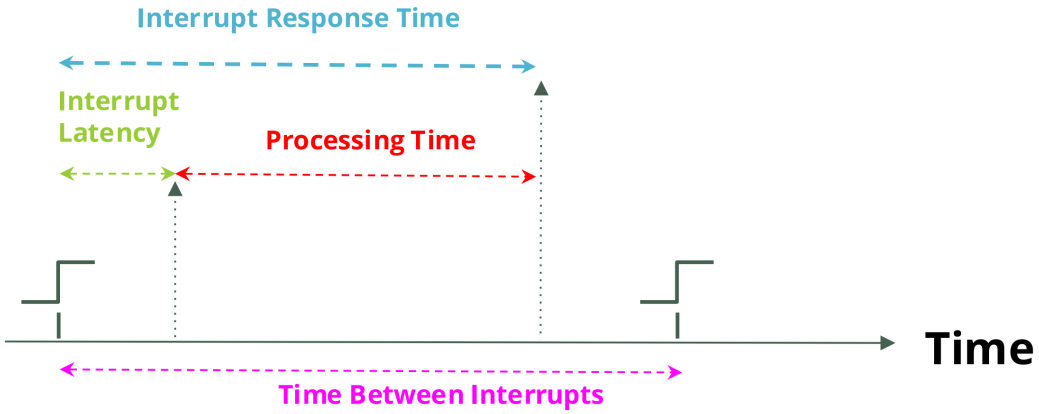
4.8 Overhead of Interrupts

An interrupt causes a change in the program counter, which incurs a branch penalty: the pipeline of instructions gets invalidated. So if interrupts occur frequently there will be a noticeable overhead.

If the interrupt automatically stores CPU registers, it requires extra cycles.

Also the acknowledgements, priorities, and the obtaining the index in the table from the device all require extra cycles.

All around interrupts add overhead and latency to the device. Real-time, or time-critical applications require great knowledge of interrupt and its mechanisms as to minimise overhead.



4.9 Co-processors

They are processors attached to the CPU that implement some instructions, mostly maths instructions. The most common co-processor adds **floating-point arithmetic capabilities**.

A co-processor has its own registers and hardware. The co-processor is ordered by the CPU to activate, receive the instruction and execute it when it receives a co-processor instruction via certain opcodes reserved in the instruction set.

The CPU can either:

- suspend execution to wait for the co-processor to finish
- continue executing instructions (super-scalar approach)

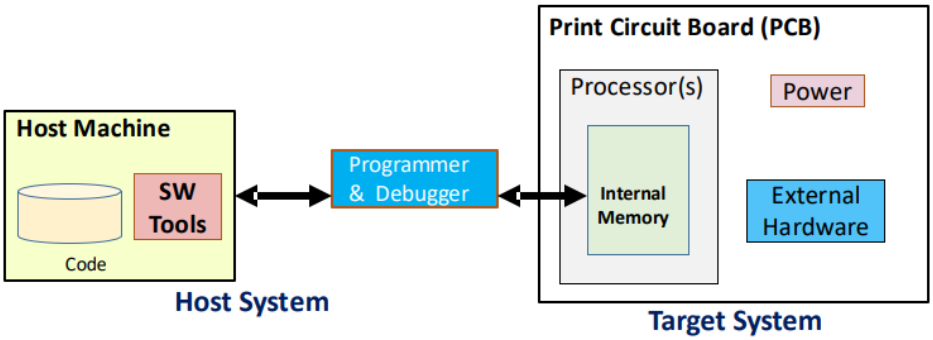
Chapter 5

Introduction to TI MSP432 Launchpad

5.1 Embedded System Development Platform

It consists of the **host system**, that has the cross compiler, linker and source-level debugger. It also consists of the **target system**, with debugger, processor, sensors...

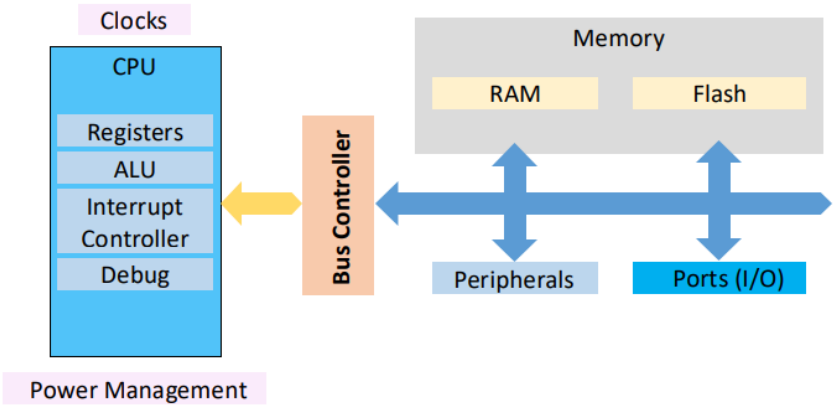
There must be connections between the host and target system to **download program images and transmit debugger information** between the host debugger and the target debug agent.



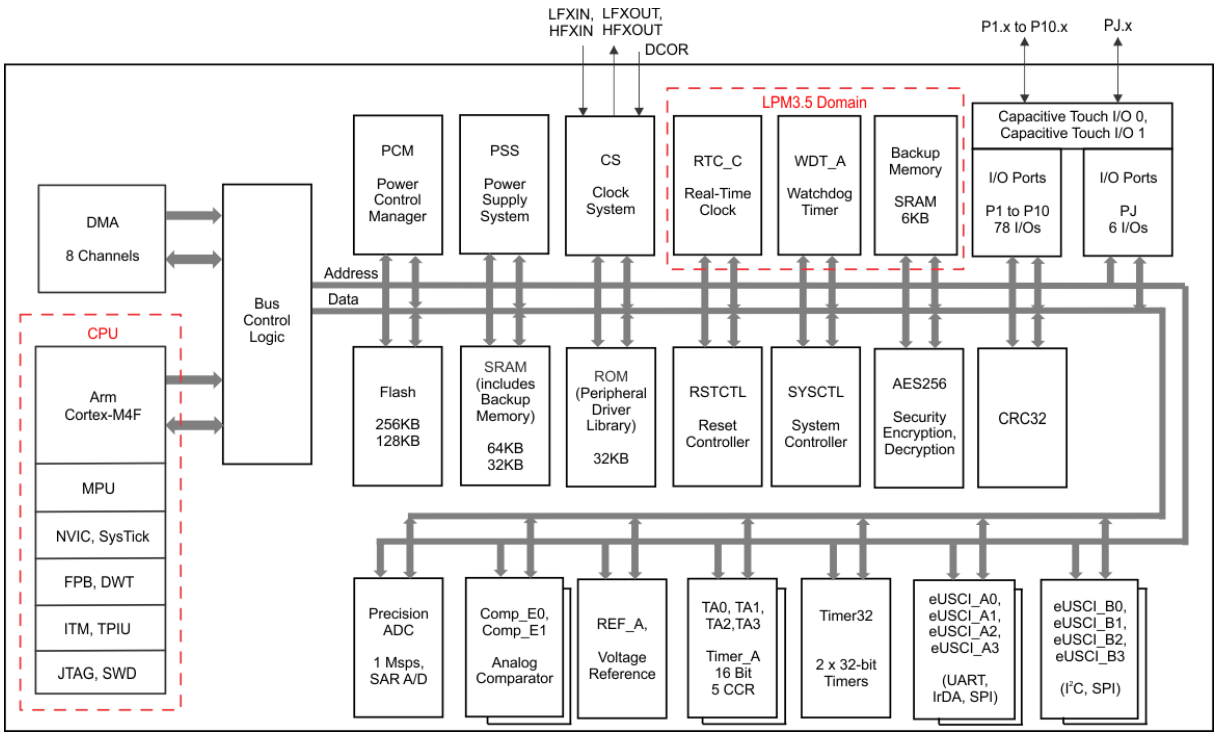
In the *TI MSP432 Launchpad* the Programmer & debugger is also inside the target system.

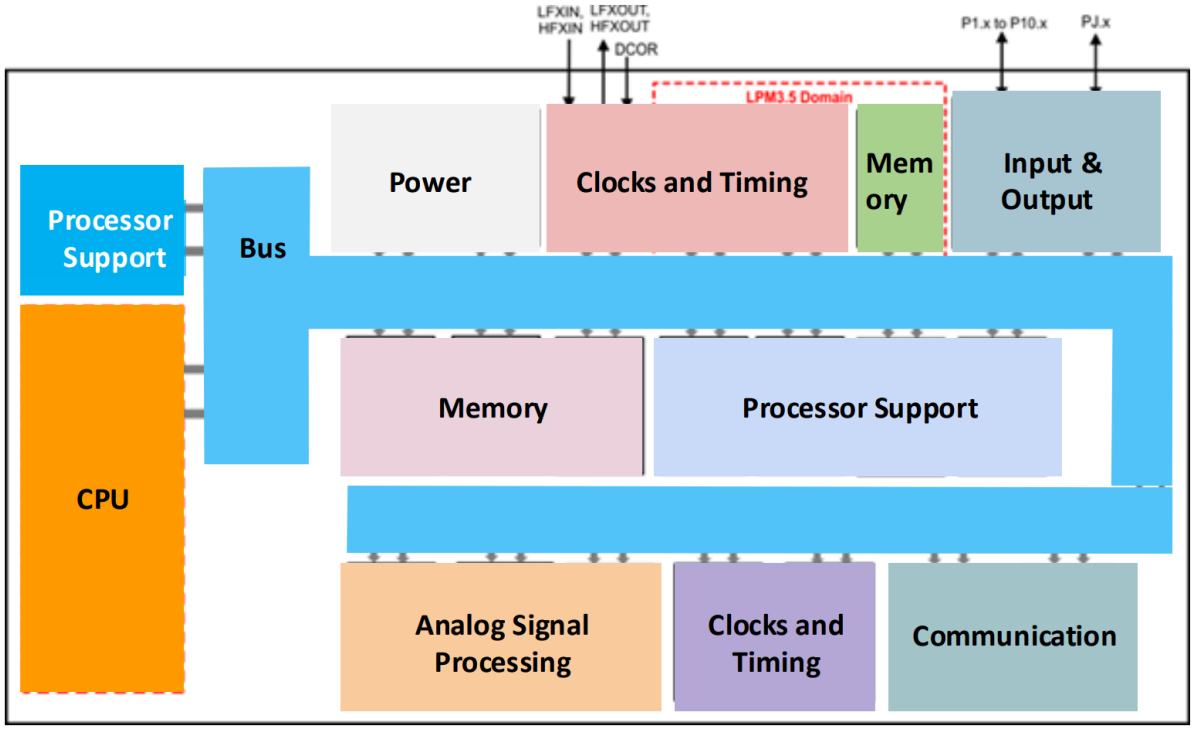
5.1.1 Launchpad

It's a whole Printed Circuit Board (PCB) used as a development kit, allowing you to mess around with the code and re-flash it each time. When the code is ready for production, and can be burned in a ROM, all you will need is the processor. Most vendors provide these kits.



5.2 Microcontroller Components





5.2.1 TI-MSP432 microcontroller main features

- **ARM(r) Cortex(tm)-M4F** processor - Designed for low-power, embedded systems application
- MSP stands for Mixed Signals Processing, can read both analogue and digital values
- 100-pin microcontroller chip
- Peripherals: 256KB on-chip Flash memory for code, 64KB on-chip SRAM for data, large number of on-chip peripherals

5.2.2 Memory Map in MSP432P401R

	Allocated size	Allocated address
Flash	256KB	0x0000 0000 to 0x0003 FFFF
SRAM	64KB	0x2000 0000 to 0x2000 FFFF
I/O	All the peripherals	0x4000 0000 to 0x4001 FFFF

5.3 Running an application on the launchpad

New project:

→ Create new project

→ Select target microcontroller

→ Empty project with main.c

→ Everything else default

Compiling the project:

→ Rebuild the project

Burn/Load the project onto the launchpad:

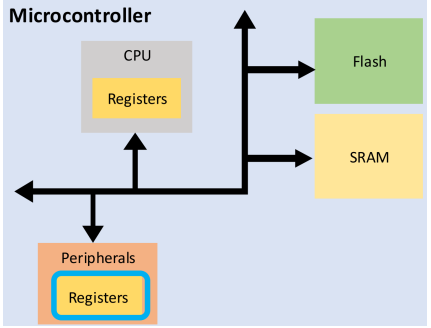
→ Debug project

The application runs on bare metal programming, thus there is no OS or external support whatsoever. You can see all instructions and manipulate memory at will. All you're facing is the actual hardware. We can see the registers, variables, memory browser, assembly instructions and so on (expressions, breakpoints...).

5.4 Peripherals

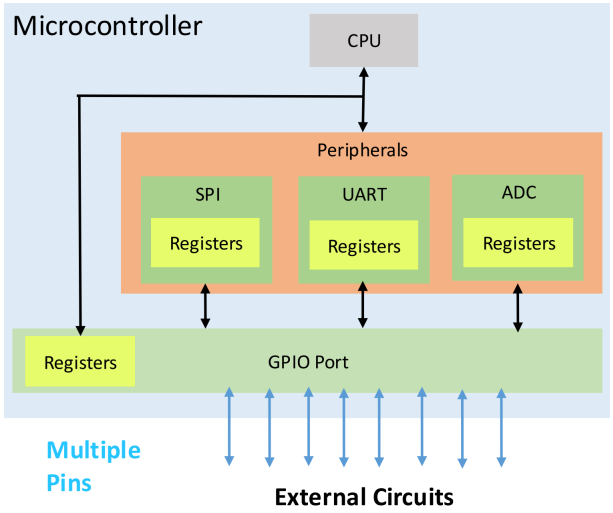
They are external to CPU:

- Memory-mapped I/O to configure peripherals
- Pointers
- Memory Reads/Writes
- Use of Bit Manipulation

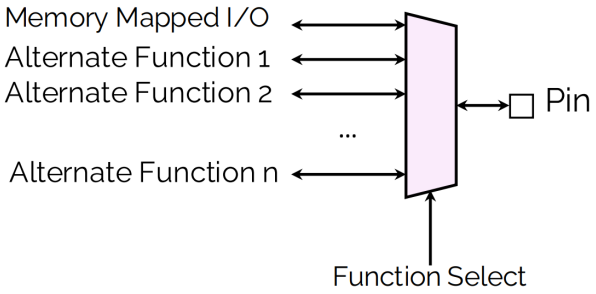


5.5 GPIO - Input/Output Systems

GPIO-General Purpose IO: Pin – physical connection to microcontroller, Port – combination of pins
Input/Output: method to get data in/out of microcontroller



Pins may have different features, to enable an alternative function, set up the appropriate register



In the microcontroller, we have two types of I/O:

General Purpose I/O (GPIO) - the GPIO ports are used for interfacing devices such as LEDs, switches, LCD, keypad, and so on.

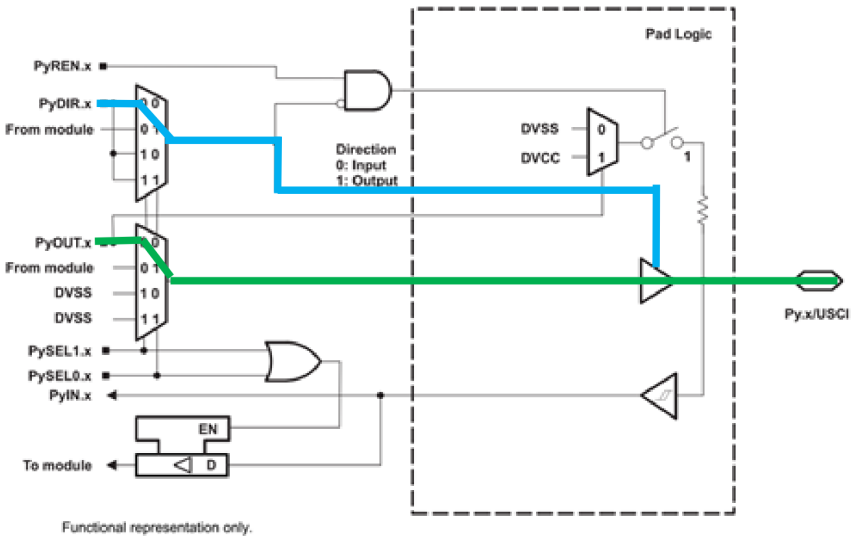
Special purpose I/O - These I/O ports have designated function such as ADC (Analog-toDigital), Timer, UART (universal asynchronous receiver transmitter), and so on.

The GPIO ports in MSP432 are designated as port P1 to P10 - Simple I/O or Digital I/O ports - and PJ - special function such as crystal oscillator and JTAG connections.

5.6 Configuring LED

To **configure** LED on P1.0

- set direction to output: **P1DIR**
- Set P1.0 output to high: **P1OUT**



To **turn on** LED, P1.0_LED1: voltage needs to be a Logical HI (VCC 3.3V).
Also Required to Set Pin to **I/O Mode** (P1SELx).

Two important registers: IO Direction Register, **P1DIR**, IO Output Register, **P1OUT**.

5.6.1 P1DIR Register

Port Direction Register sets the pin to:

- 0: an input
- 1: an output

```
/* Configure P1.0 output */
P1->DIR |= BIT0;
```

5.6.2 P1OUT Register

Port Output Register sets output to:

- 0: Output set to LOW (GND)
- 1: Output set to HIGH (VCC)

```
/* Configure P1.0 output */
P1->DIR |= BIT0;
/* Set P1.0 HIGH */
P1->OUT |= BIT0;
```

5.7 Configure the buttons

Buttons are connected to P1.1 and P1.34

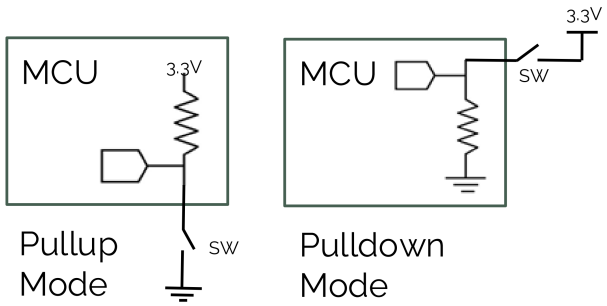
```
/* Configure P1.1 as input*/
P1->DIR = ~BIT1; //BIT0 as output
/* Select pullup*/
P1->OUT = BIT1; //BIT0 pulldown
/* Enable pullup*/
P1->REN = BIT1; //BIT0 disable
/* Set as I/O */
P1->SEL0 = 0;    //general purpose I/O selected
P1->SEL1 = 0;
...
/* Catch Button Press */
while (P1->IN & BIT1);
while (!(P1->IN & BIT1));
...
```

5.7.1 Pull-up and Pull-down Resistors

If we do not use internal pull-up (or pull-down) resistors, we have a problem. We need to ensure a known value on the output if a pin is left floating.

We want the switch SW to pull the pin to ground, so we enable the pull-up. The pin value is:

- High when SW is not pressed
- Low when SW is pressed



Chapter 6

Fundamental Building Blocks

Embedded software must run at a required rate to meet system deadlines, must meet power consumption requirements, must meet timing requirements and must fit into the allowed amount of memory.

Given these constraints, everything is mostly done from scratch and it's hard to find perfect libraries for our applications.

Despite this, there are basic building blocks that can compose a program.

6.1 Finite State Machines - FSMs

An FSM is simply a machine that reacts to inputs via states.

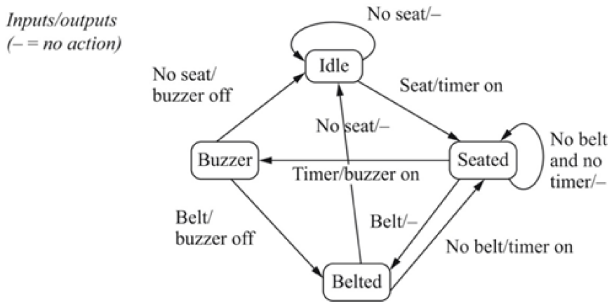


Figure 6.1: Simple example of FSM

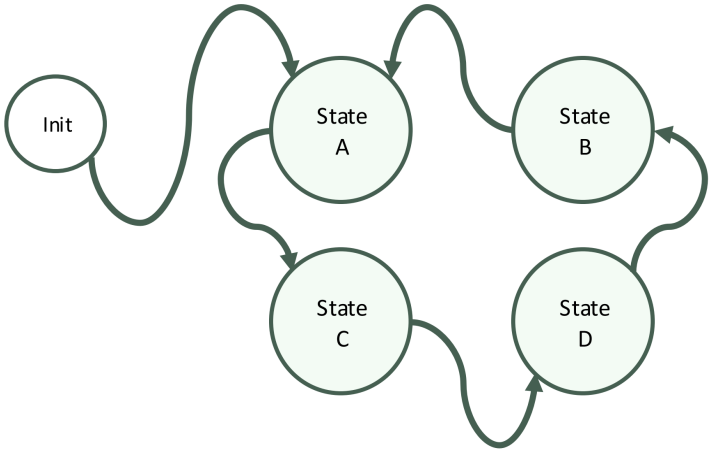
We can model it using this Pseudo-Code:

```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3

switch(state) { /* check the current state */

    case IDLE:
         if (seat){ state = SEATED; timer_on = TRUE; }
         /* default case is self-loop */
         break;
    case SEATED:
         if (belt) state = BELTED; /* won't hear the buzzer */
         else if (timer) state = BUZZER; /* did not put on belt in time */
         /* default case is self-loop */
         break;
    case BELTED:
         if (!seat) state = IDLE; /* person left */
         else if (!belt) state = SEATED; /* person still in seat */
         break;
    case BUZZER:
         if (belt) state = BELTED; /* belt is on---turn off buzzer */
         else if (!seat) state = IDLE; /* no one in seat-turn off buzzer */
         break;
}
```

NOTE: Using function pointers decreases the complexity of the FSM program.



6.1.1 Define states and and state machine structure:

```
typedef enum{
    STATE_A;
    STATE_B;
    STATE_C;
    STATE_D;
    NUM_STATES
}State_t;

typedef struct{
    State_t state; /* defines the command */
    void (*func)(void); /* defines the function to execute */
} StateMachine_t;
```

6.1.2 Declare functions that implement states, variable to hold current state and the state machine itself:

```
/* state machine function prototypes */
void fn_StateA(void);
void fn_StateB(void);
void fn_StateC(void);
void fn_StateD(void);

/* variable that holds the current state */
State_t cur_state = STATE_A:

StateMachine_t StateMachine[] = {
    {STATE_A, fn_StateA},
    {STATE_B, fn_StateB},
    {STATE_C, fn_StateC},
    {STATE_D, fn_StateD}
} ;
```

6.1.3 Fill the functions:

```
void fn_StateA(void){
    /* add the necessary code here */
    cur_state = STATE_B;
}
void fn_StateB(void){
    /* add the necessary code here */
    cur_state = STATE_C;
}
void fn_StateC(void){
    /* add the necessary code here */
    cur_state = STATE_D;
}
void fn_StateD(void){
    /* add the necessary code here */
    cur_state = STATE_A;
}
```

6.1.4 Run function:

```
void run(void){
    if(cur_state < NUM_STATES){
        (*StateMachine[cur_state].func)();
    }else{
        // error
    }
}
```

6.2 Circular Buffer

Circular buffers are a data structure useful for stream-oriented processing (where data comes regularly and must be processed on the fly, basically we can read values but also must consume values).

6.3 Queue

Queues are a data structure useful for data that arrives and departs at unpredictable times. A synonym of queue is **elastic buffer.**

One way to implement a queue is through linked lists, that allows the queue to grow to an arbitrary size at the cost of overhead of dynamic memory allocation (therefore not recommended). Another way is through an array to hold all the data.

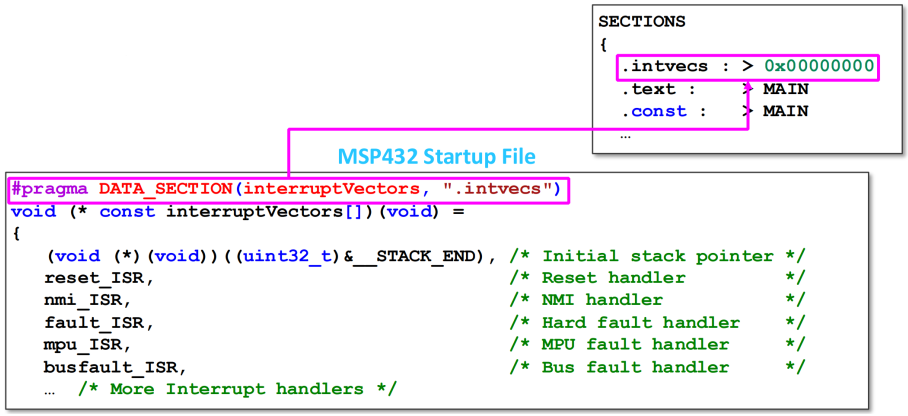
A queue may have varying numbers of elements in it as it doesn't have to accommodate all of the incoming data, but a circular buffer will always have a fixed number of data.

6.4 Producer/Consumer problems

Some aspects of an application can be a producer and others can be a consumer. This is meant in regards to writing (producing) and reading (consuming) data. Basically we should strive to keep the loads balanced

6.5 Pragma

A `pragma` is a compiler directive, usually platform-specific.



In this case, it is used to specify the data section of interruptVectors in .intvects interruptVectors is the Interrupt Vector Table.

Since it is a constant array it will be placed in the Flash memory. The first entry is the Initial Stack Pointer to initialised the Core CPU registers.

The other five entries are High-Priority ARM Core Exceptions. The rest of the handlers are for GPIO interrupts

6.6 Default Handler

```
...
/* Forward declaration of the default fault handlers. */
void Default_Handler (void) __attribute__((weak));
...
/* This is the code that gets called when the processor receives an unexpected */
/* interrupt. This simply enters an infinite loop, preserving the system state */
/* for examination by a debugger. */

void Default_Handler(void){

    /* Fault trap exempt from ULP advisor */
    #pragma diag_push
    #pragma CHECK_ULP("-2.1")

    /* Enter an infinite loop. */
    while(1){

    }

    #pragma diag_pop
}
```

`__attribute__((weak))` means that it can be overridden by the developer

6.6.1 Overriding Default Handlers

```
...
extern void PORT1_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
extern void PORT2_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
extern void PORT3_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
extern void PORT4_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
extern void PORT5_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
extern void PORT6_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
...
```

alias means that it defaults to `Default_Handler` if it is not implemented. *extern* means that the function can be defined in other source files.

6.7 Implementing Interrupts

We can do this by using Interrupt **Edge Select Registers** (PxIES), **Interrupt Flag Registers** (PxIFG), and **Interrupt Enable Registers** (PxIE).

Example: left button P1.1

```
P1->IES = BIT1; //PxIFG flag set with high-to-low-transition, BIT0 viceversa
P1->IE = BIT1;  //Port interrupt enabled, BIT0 viceversa
P1->IFG = 0;    //No interrupt is pending/Clear all interrupt flags, 1 viceversa
```

Now we need to program the NVIC, enabling the corresponding ISR on the NVIC IRQ assignment, by using NVIC's ISER (Instruction Set Enable Registers):

- We know that the IRQ# (IRQ number) of the I/O Port P1 is 35 from the documentation
- We know that the ISER1 register covers all IRQ# from 32 to 63

```
NVIC->ISER[1] = 1 << ((PORT1_IRQn) & 31);
```

The instruction first calculates the bitwise AND (&) between `PORT1_IRQn` and 31, the maximum index of the `ISER[1]` register. This is a common safety precaution in embedded software to prevent unintentional modification of the bits outside the valid range of the register. After that, the number 1 is then shifted X places to the left, where X is the result of the bitwise AND.

This is because the IRQ# is 35, so we needed to shift it 3 places from 32. 3 was the exact result of the bitwise AND between the IRQ# and the number 31.

Then we write the actual ISR:

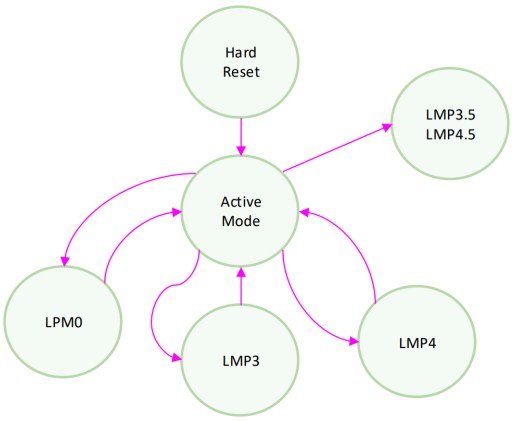
```
/* Port1 ISR */
void PORT1_IRQHandler(void){
    // Toggling the output on the LED
    if(P1->IFG & BIT1)
    P1->OUT ^= BIT0;
    // clear the flag
    P1->IFG &= ~BIT1;
}
```

It checks if there's an interrupt; if there is one it toggles the LED's output and then clear the flag (removes the pending interrupt so we can catch the next one).

6.8 Low-Power Modes

MSP432 supports several power modes for operation: allow for the optimisation of power for a given application scenario

MSP432	Description	Comments	Current
Active	Active Mode	CPU & peripherals	**100uA/MHz**
LPM0	Sleep	Peri. on, CPU off	**65uA/MHz**
LPM3	Deepsleep, Stand-by w/ RAM & RTC	some peripherals available	**<900nA**
LPM4	Stand-by with RAM	No clocks, some peripherals available	**<900nA**
LPM3.5	Shutdown	RTC w/o RAM	**<670nA**
LPM4.5	Shutdown	Shutdown	**<100nA**



To enter LPM0 mode, simply use: `__sleep();`

Chapter 7

Timer Overview

A hardware timer is a digital counter that:

- counts regular events, normally using a fixed frequency clock source
- increments or decrements at a fixed frequency
- resets itself when reaching zero, or a predefined value
- generates an interrupt when reset

A software timer is a function block implemented in software:

- usually based on a hardware timer, increments/decrements when interrupted
- provides lower time precision
- can have multiple instances, notably more than hardware timers

7.1 Timer use-cases

Timers can be used to:

generate periodic events

measure time passed to perform some computational tasks

generate pulse width modulation

7.2 Components of a Standard Timer

A timer is made of multiple components:

- **Clock source/Oscillator**
- **Prescaler**

  Takes the clock source as input
  Divides the input frequency by a predefined value (4, 8, 16...)
  Outputs the divided frequency to other components

- **Timer Register**

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

7.2.1 Prescalers and software performance

Prescalers significantly improve system performance when there is a need to:

- reduce power consumption
- count longer time intervals
- In real-time systems, the choice of prescalers should be carefully considered as to meet critical timing requirements

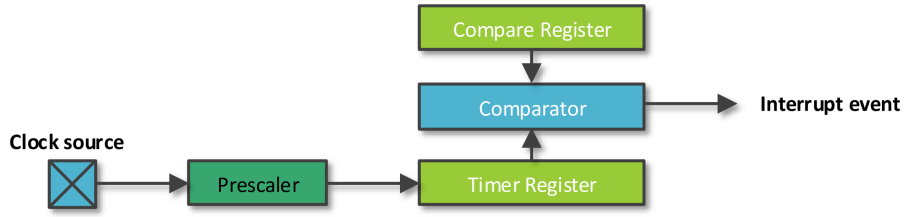
7.3 Timer Operation Modes

A standard timer typically has three operation modes.

7.3.1 Compare Mode

The compare mode has a compare register. The compare register is loaded with a value that is periodically compared with the value in the timer register.

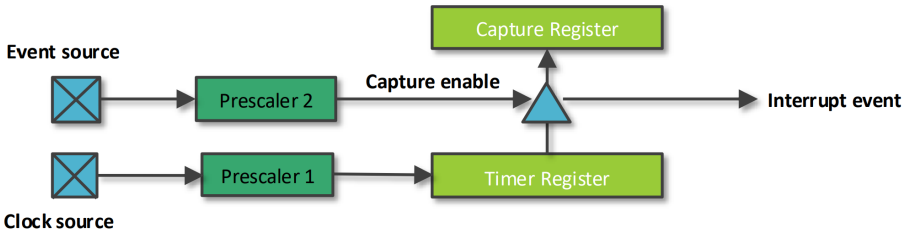
Once the two values are the same an interrupt can be generated (e.g. every 15 seconds).



7.3.2 Capture Mode

The capture mode has a capture register that captures the current value of the timer register upon the capture of external events. It can also generate an interrupt.

Optionally, the prescaler can be used to divide the frequency of the events.



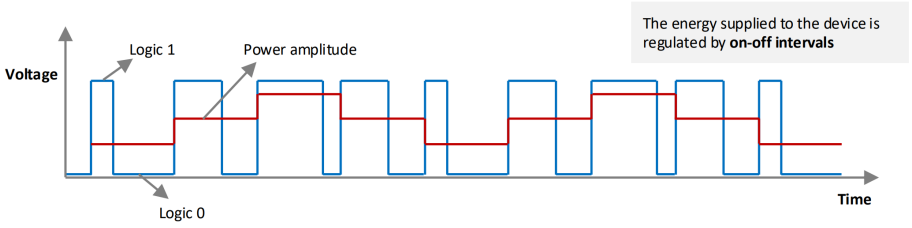
Capture/Compare Registers

Capture/Compare Registers, also known as CCRs, are registers used to configure and control various timer-related functionalities. They are registers within the timer module.

7.3.3 Pulse-width modulation (PWM) Mode

This mode uses the width of a pulse to modulate an amplitude, which reflects the duty cycle, which describes the proportion of the 1 state in one pulse period.

The PWM mode is mainly used to control the power supplied to electrical devices (via on-off intervals).



7.4 MSP432 Timer A

In our launchpad are 4 16-bit timers: TA0, TA1, TA2, TA3. Each one has 7 capture/compare registers, allowing software timers.

There are TAxR registers that can be manipulated. They can be cleared by setting the TACLR bit.

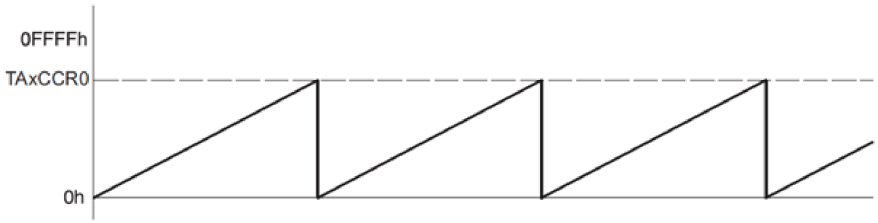
The timer clock can be sourced from: ACLK, SMCLK, or externally from TAxCLK or INCLK. The clocks are connected to crystal oscillators. The clock source is selected with the TASSEL bits.

The selected clock source may be passed directly to the timer, or it can be slowed down by 2, 4, 8 using ID bits. Or 2, 3, 4, 5, 6, 7, 8 using TAIDEX bits.

MC	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TAxCCR0
10	Continuous	The timer repeatedly counts from zero to 0FFFFh.
11	Up/down	The timer repeatedly counts from zero up to the value of TAxCCR0 and back down to zero.

7.4.1 Up Mode

In Up Mode the timer repeatedly counts up to the value of the compare register, TAxCCR0. Once the two values match the TAxCCR0 CCIFG (capture compare register interrupt flag) is set. We need to enable the interrupt first. We do this via the CCIE bit in the TAxCCTL0 (Capture/Compare Control Register) register.



7.5 Periodic timer interrupt setup

We first set the CCIE (Capture Compare Interrupt Enable) bit in TA0 CCTL0 register to enable interrupt.

```
TIMER_A0->CCTL[0] = TIMER_A_CCTLN_CCIE;
```

We are using the first capture/compare register of the first timer A. Then we set the value for the comparison.

```
TIMER_A0->CCTL[0] = TIMER_A_CCTLN_CCIE;
TIMER_A0->CCR[0] = 50000;
TIMER_A0->CTL = TIMER_A_CTL_SSEL__SMCLK | TIMER_A_CTL_MC__CONTINUOUS;
```

In this case the timer will count up to 50000. We then set the clock source as SMCLK (3 MHz) in continuous mode. We use the TA0CTL Control Register for this register.

We can also use an input divider (Prescaler) for SMCLK to slow down the input clock signal. By default the SMCLK is 3 MHz, which means 3,000,000 ticks per second. By dividing by 8, we get 375,000 ticks per second, the equivalent of 7 interrupts per second.

```
TIMER_A0->CCTL[0] = TIMER_A_CCTLN_CCIE;
TIMER_A0->CCR[0] = 50000;
TIMER_A0->CTL = TIMER_A_CTL_SSEL__SMCLK | TIMER_A_CTL_MC__CONTINUOUS |
    TIMER_A_CTL_ID_3;
NVIC->ISER[0] = 1 << ((TA0_0_IRQn) & 31);
```

We then have to enable the timer IRQ. There is a specific IRQ line for CCR0 (the first compare register), while all others (TA0CCR1) share the same IRQ. Weird design choice. To counter this (pun intended), we just need to check which CCR triggered the interrupt within the ISR to distinguish between them.

bit	Name	Description
0	TAIFG	Timer_A Interrupt Flag 0: Timer did not overflow 1: Timer overflowed
1	TAIE	Timer_A Interrupt Enable (0: Disabled, 1: Enabled)
2	TACLR	Timer_A Clear
4-5	MC	Mode Control: 00: Stop mode: timer is halted 01: Up mode: Timer counts up to TAxCCR0 10: Continuous mode: Timer counts up to 0xFFFF 11: Up/down mode: Timer counts up to TAxCCR0 then down to 0.
6-7	ID	Input divider: These bits select the divider for the input clock: 00: divide by 1 01: divide by 2 10: divide by 4 11: divide by 8
8-9	TASSEL	Timer_A clock Source Select: These bits select the Timer_A clock source: 00: TAxCLK (external clock): The timer uses external clock which is fed to the PM_TAxCLK pin. 01: ACLK (internal clock) 10: SMCLK (internal clock) 11: INCLK

We then override the Timer Interrupt Handler:

```
// will be called when TA0CCR0 CCIFG is set
    void TA0_0_IRQHandler(){
    // clear the interrupt flag
    TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG;
    // toggle LED
    P1->OUT ^= BIT0;
}
```

Just like the other interrupts, we clear the interrupt flag and then do our thing.

7.6 Timer Overflow and Counters

We have to keep in mind timer overflow and counters.

Timer overflows can be used with counters to measure time (e.g. count 1 second if the timer overflows 6 times).

Timer overflows happen because Time Registers have a finite count range (in MSP432 they are 32-bit). Counters can also be used to measure performance, frequencies, duty cycles and other performance-related metrics.

7.7 Watchdog Timer

Apart from timers we have watchdog timers. Watchdog module performs a controlled system restart after a software problem occurs/the system gets stuck. Watchdog timer counts down from a selected interval. If the selected time interval expires, a system reset is generated.

To edit the control register, the WDTCTL register, we have to insert the password first:

```
WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD
```

7.8 Clocks in MSP432

Microcontrollers usually have two types of clock resources:

- **on-chip** oscillator (internal oscillator). **Advantage**: always present
- oscillator **connected** to external crystal (external oscillator). **Advantage**: higher precision

The clock system contains the sources of the various clocks in the device. It also controls the mapping between the sources and the different clocks in the device.

7.8.1 External clock resources:

- LFXTCLK: Low-Frequency Oscillator (LFXT) 32-kHz or below
- HFXTCLK: High-Frequency Oscillator (HFXT) 1-MHz to 48-MHz range

7.8.2 Internal clock resources:

- DCOCLK: Internal Digitally Controlled Oscillator (DCO) Programmable frequencies (3-MHz frequency by default)
- VLOCLK: Internal Very-Low-Power Low-Frequency Oscillator (VLO) 9.4-kHz typical frequency
- REFOCLK: Internal Low-Power Low-Frequency Oscillator (REFO) Selectable 32.768-kHz or 128-kHz frequencies
- MODCLK: Internal Low-Power Oscillator 25-MHz typical frequency
- SYSOSC: Internal Oscillator 5-MHz typical frequency

Five primary system clock signals are available from the clock module:

ACLK (Auxiliary Clock)

MCLK (Master Clock) -> To CPU

HSMCLK (Subsystem Master Clock)

SMCLK (Low-Speed Subsystem Master Clock)

BCLK (Low-Speed Backup Domain Clock)

The clock system can be configured or reconfigured by software at any time during program execution. We have selection and divider signals.

Table 2. Default Clock Operation

Clock	Default Clock Source	Default Clock Frequency	Description
MCLK	DCO	3 MHz	Master Clock Sources CPU and peripherals
HSMCLK	DCO	3 MHz	Subsystem Master Clock Sources peripherals
SMCLK	DCO	3 MHz	Low-speed subsystem master clock Sources peripherals
ACLK	LFXT (or REFO if no crystal present)	32.768 kHz	Auxiliary clock Sources peripherals
BCLK	LFXT (or REFO if no crystal present)	32.768 kHz	Low-speed backup domain clock Sources LPM peripherals

- LFXTCLK : 32-kHz crystal
- HFXTCLK: 48-MHz crystal

Example Configuration

We first need to unlock the CS module for register access, which is protected by password to prevent inadvertent access.

```
CS->KEY = CS_KEY_VAL;
 CS->CTL1 |= CS_CTL1_SELA_2 | CS_CTL1_SELS_3 | CS_CTL1_SELM_3
 CS->KEY = 0;
```

We could also use a divider to slow down the CPU frequency, by doing this:

```
CS->KEY = CS_KEY_VAL;
 CS->CTL1 |= CS_CTL1_SELA_2 | CS_CTL1_SELS_3 | CS_CTL1_SELM_3 | CS_CTL1_DIVM_7
 CS->KEY = 0;
```

This way we divide the Master Clock signal by 128. The DCO by default is $3MHz$, so it's reduced to $23,437.5Hz$

7.8.3 Slowing CPU to the smallest frequency possible

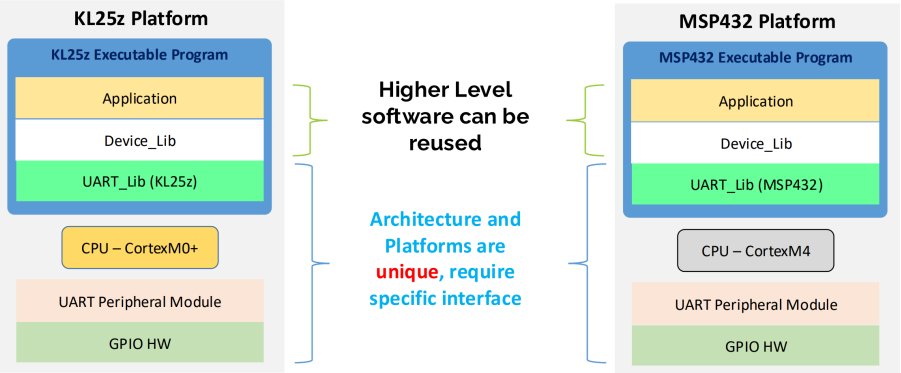
We can do this by setting the Master Clock to use the DCO then set the DCO's frequency to 1.5 MHz then divide it by 128. We achieve $\sim 24kHz$ by doing so. Or we can set the Master Clock to use VLOCK, which is 9.5 kHz.

Chapter 8

Software Independence

A good practice is to write as much software as independent as possible from architectures and platforms to maximize software **portability** and **reusability**.

It's impossible to make everything independent as firmware layers still intersct with hardware and Assembly code is archiecture dependent as it depends on that hardware's IS.



8.1 Binary Interfaces

The Embedded Application Binary Interface (EABI) is a set of conventions and standards to provide guidelines for:

- how functions are called
- how data is organized in memory
- how exceptions are handled in embedded applications.

Binary interfaces specify details of how the executable must run on this architecture (done by compilers in most cases).

In architecture terms, an instruction is a fundamental unit of work or operation (arithmetic, logical, program flow control, load/store), while a word is a fundamental operand size for each operation.

Instruction Sizes: Instruction size can vary between instruction sets (like ARMv6-M and Thumb-2, for example).

Instruction – Fundamental unit of work or operation (Arithmetic, Logical, Program Flow Control, Load/Store)

Word – fundamental operand size for each operation

Standard Integer Sizes: Variable length types can cause portability issues, we want **portable data types**. Explicitly defined types that specify storage and design are defined in the `stdint.h` header file.

```c
#ifndef __STDINT_H__
#define __STDINT_H__

/* 8-bit signed/unsigned Integers */
typedef signed char int8_t;
typedef unsigned char uint8_t;

/* 16-bit signed/unsigned Integers */
typedef signed short int int16_t;
typedef unsigned short int uint16_t;

/* 32-bit signed/unsigned Integers */
typedef signed long int int32_t;
typedef unsigned long int uint32_t;

/* 64-bit signed/unsigned Integers */
typedef signed long long int int64_t;
typedef unsigned long long int uint64_t;

#endif /* __STDINT_H__ */
```

8.2 Pointer Types

All Pointers are the same length. This because pointer hold addresses in memory of 32-bit.
We need to cast in order to obtain the data inside the memory.

$$sizeof(uint8_t*) = sizeof(int16_t*) = sizeof(uint32_t*) = sizeof(float*) = 32\text{-Bits!}$$

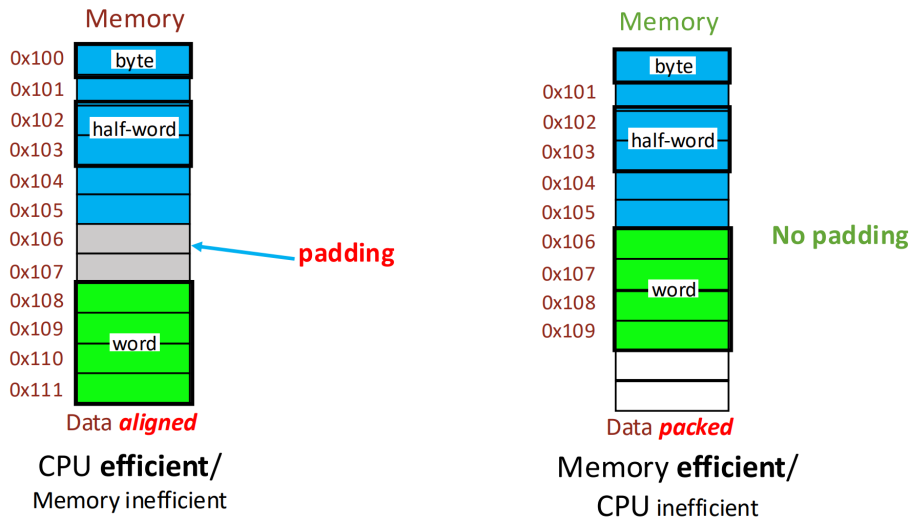
```c
uint8_t * ptr1 = (uint8_t *) 0x00;
uint16_t * ptr2 = (uint16_t *) 0x04;
uint32_t * ptr3 = (uint32_t *) 0x08;
float * ptr4 = (float *) 0x0C;
```

$$sizeof(ptr1) = sizeof(ptr2) = sizeof(ptr3) = sizeof(ptr4) = 32\text{-Bits!}$$
$$sizeof(*ptr1) \neq sizeof(*ptr2) \neq sizeof(*ptr3) \neq sizeof(*ptr4)$$

8.2.1 NULL Pointers

In embedded platforms it is always the device (no OS, no other software) that uses pointers carefully.
Null Pointers point to nothing, dereferencing a NULL Pointer can cause an **exception**.

8.3 Data alignment



Load/store data occurs only at aligned addresses in memory.

8.3.1 How to control alignment of data in memory

We use a compiler attribute which changes based on the compiler.

```
struct struct_name {
    int8_t var1 int8_t var1 __attribute__ ((aligned(4)));
    int32_t var2;
    int8_t var3;
} __attribute__ ((packed)); //if not appear default is aligned:  __attribute__ ((
    aligned));
```

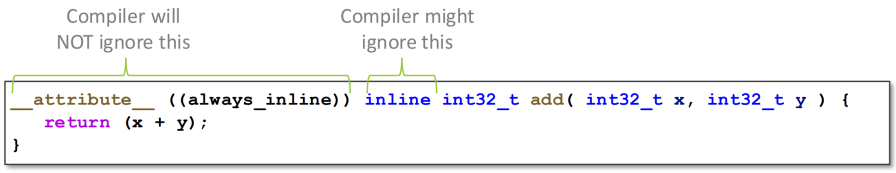
In this case, since it's packed, the size of the structure will be 6 bytes rather than 12 bytes when data is aligned/padded. It will be less efficient for the CPU, however.

8.4 Function Attributes

Compiler attributes can apply to functions.

`inline` is a C99 compiler directive to `inline` a function: rather than calling small functions, since calling functions adds overhead, functions are automatically added where they are called. The compilers won't necessarily `inline` every function unless `((always_inline))` (GCC attribute) is used.

Inlining is good for small functions.

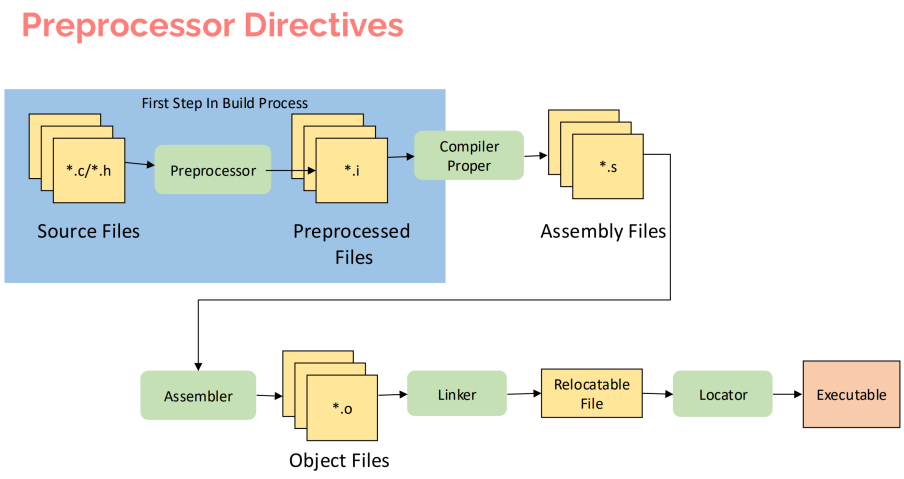


8.4.1 Function Pragmas

Pragmas are preprocessor directives that provide special instructions to the compiler via software and not command line.

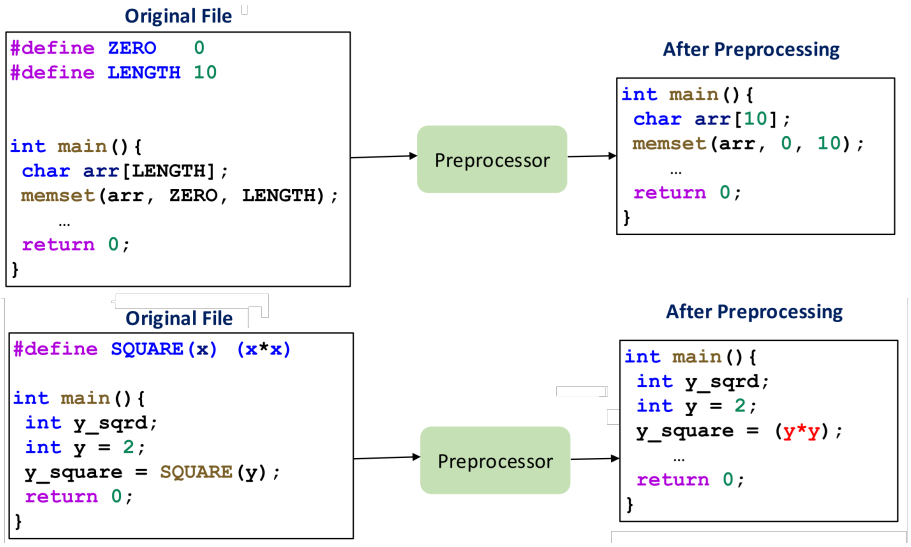
```
#pragma GCC push
#pragma GCC optimize ("O0")
int32_t add( int32_t x, int32_t y ){
    return (x + y);
}
#pragma GCC pop
```

8.5 Preprocessor Directives



Preprocessor directives are special keywords used by the preprocessor before compilation. All of them start with # sign:

- Important Directives
- #define, #undef
- #ifndef, #ifdef, #endif
- #include
- #warning, #error
- #pragma



NOTE: pay attention of undefined behavior (y++, it becames $2 * 3$).

8.6 TI DriverLib

The DriverLib is a software layer to the programmer: facilitates higher level of programming compared to direct register accesses