Introduction to ARM Cortex-M and STM32 MCU

Overview about ARM Cortex and STM32 MCUs, with general information about registers, memory, interrupts and CMSIS. Different types of STM32 MCUs.

#arm #stm32

Last update: April 23, 2021

Table of Content

1. ARM Cortex microcontrollers

ARM processors use *Reduced Instruction Set Computing* (RISC) architectures, and nowadays have many revisions (ARMv6, ARMv6-M, ARMv7, ARMv7-A, etc.). ARM means *Advanced RISC Machines.*

ARM Cortex is a wide set of 32/64-bit core architectures, which are based on ARM architecture revision. For example, a processor based on the Cortex-M4 core is designed on the ARMv7-M architecture.

ARM Cortex microcontrollers are divided into three main subfamilies:

- **Cortex-A** which stands for **A**pplication
- **Cortex-R** which stand for **R**eal-Time
- **Cortex-M** which stands for E**M**bedded

1.1. Core Registers

Like all RISC architectures, Cortex-M processors are load/store machines, which perform operations only on CPU registers except for two categories of instructions: `load` and `store`, used to transfer data between CPU registers and memory locations
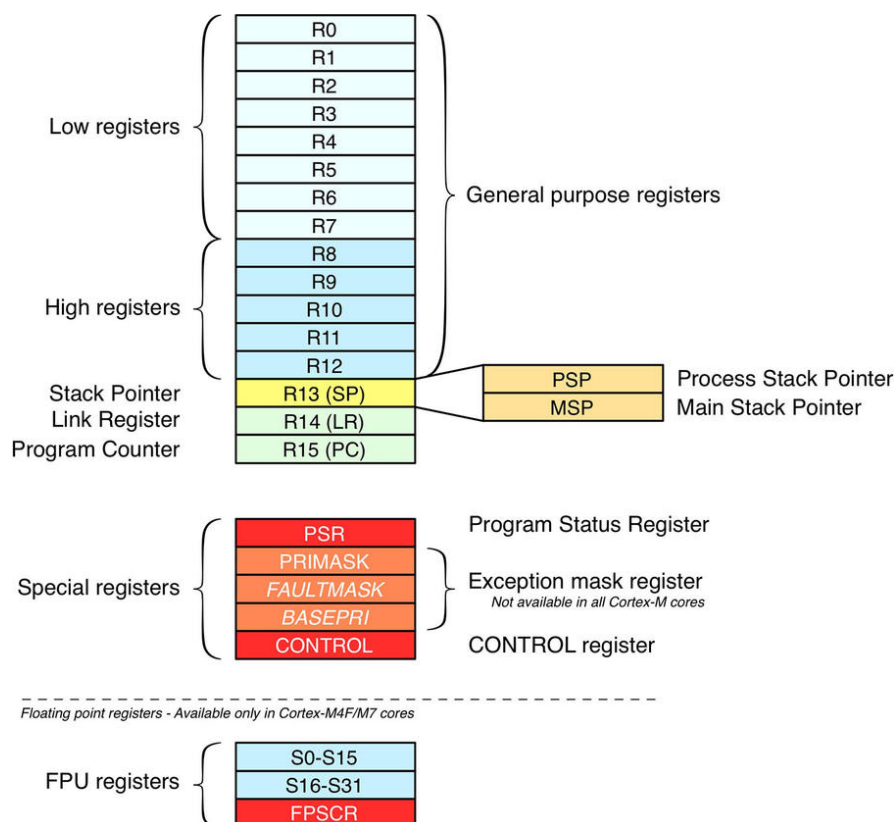

ARM Cortex-M registers

- `R0` ~ `R12` are general-purpose registers, and can be used as operands for ARM instructions. Some general-purpose registers, however, can be used by the compiler as registers with special functions.

- `R13` is the *Stack Pointer (SP)* register, which is also said to be banked. This means that the register content changes according to the current CPU mode(*privileged* or *unprivileged*). This function is typically used by *Real Time Operating Systems (RTOS)* to do context switching.

- `R14` is the *Link Register (LR)* register, which is a special-purpose register which holds the address to return to when a function call completes. This is more efficient than the more traditional scheme of storing return addresses on a call stack, sometimes called a machine stack. The link register does not require the writes and reads of the memory containing the stack which can save a considerable percentage of execution time with repeated calls of small subroutines.

- `R15` is the *Program Counter (PC)* register, which has the address of the next instruction to be executed from memory. Usually, the PC is incremented after fetching an instruction. However, control transfer instructions can change the sequence by placing a new value in the PC register.
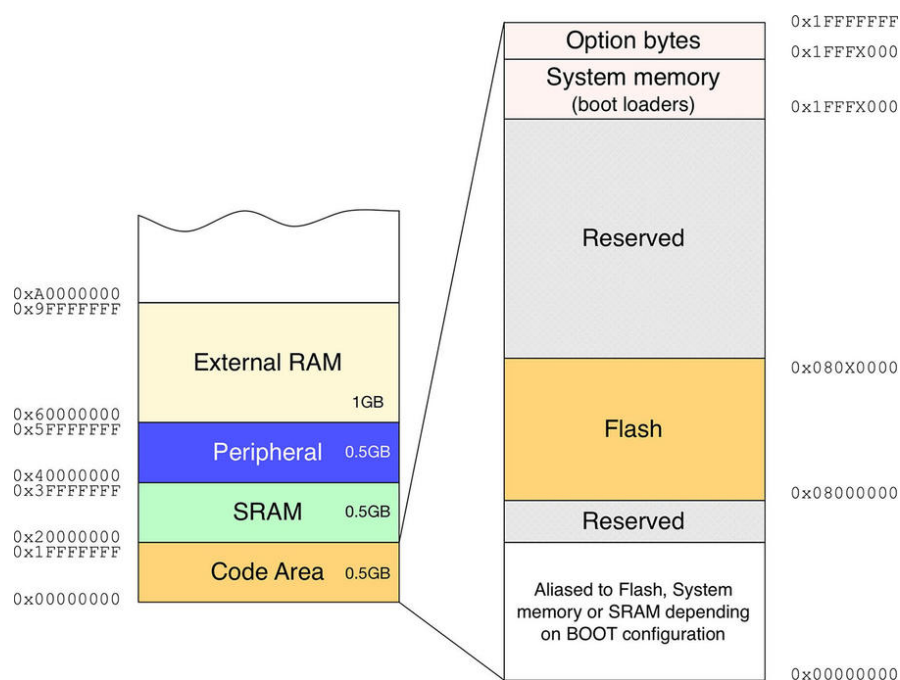
1.2. Memory Map

ARM defines a standardized memory address space common to all Cortex-M cores, which ensures code portability among different silicon manufacturer. The address space is 4GB wide, and it is organized in several sub-regions with different logical functionalities.
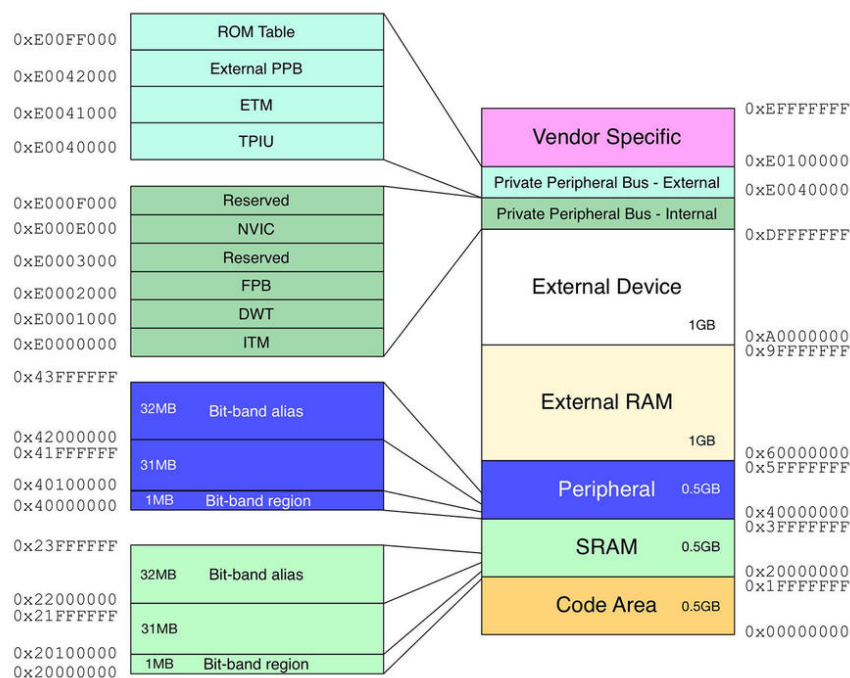
Fixed memory map for ARM cores

- The first 512MB are dedicated to code area:

  - All Cortex-M processors map the code area starting at address `0x0000 0000`. This area also includes the pointer to the beginning of the stack (usually placed in SRAM) and the system interrupt vector table.

  - An area starting at address `0x0800 0000` is bound to the internal MCU flash memory, and it is the area where program code resides. With a specific boot configuration, this area is also *aliased* from address `0x0000 0000`. This means that it is perfectly possible to refer to the content of the flash memory both starting at address `0x0800 0000` and `0x0000 0000`.

  - *System Memory* is a ROM region filled with official pre-programmed Boot Loader which can be used to load code from several peripherals, including USARTs, USB and CAN bus.

  - *Option Bytes* region contains a series of bit flags which can be used to configure several aspects of the MCU (such as flash read protection, hardware watchdog, boot mode and so on) and are related to a specific microcontroller.

- Next 512MB is mapped to Internal SRAM:

  - It starts at address `0x2000 0000` and can potentially extend to `0x3FFF FFFF`.

  - This area also can be aliased to the start up address at `0x0000 0000`.

- Other memory regions are mapped to external RAM, peripherals and the internal core registers. All Cortex processor registers are at fixed locations for all Cortex-based microcontrollers. This allows code to be more easily ported between different STM32 variants and indeed other vendors' Cortex-based microcontrollers.

Memory Map for Code Area

1.3. Bit-Banding

In embedded applications, it is quite common to work with a single bit of a word using bit masking. For example:

```c
uint8_t flags = 0;
flags |= 0x4; // set the 4-th bit
```

generates assembly code :

```
0x0a: 79fb ldrb r3, [r7, #7]
0x0c: f043 0304 orr.w r3, r3, #4
0x10: 71fb strb r3, [r7, #7]
```

Such a simple operation requires three assembly instructions (fetch, modify, save). This leads to a problem if an interruption happens between processing bit mask.

==Bit-banding is the ability to map each bit of a given area of memory to a whole word in the aliased bit-banding memory region, allowing atomic access to such bit.==
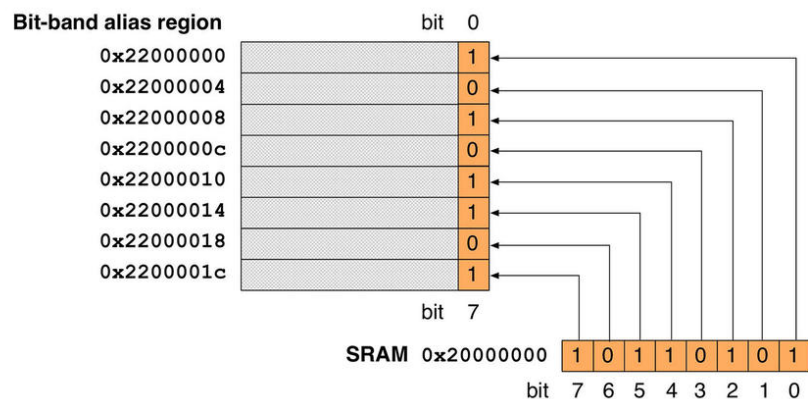
Memory Map of an address in a bit-banding region

ARM defines two bit-band regions for Cortex-M based MCUs, each one is 1MB wide and mapped to a 32Mbit bit-band alias region.

- The first one starts at `0x2000 0000` and ends at `0x200F FFFF`, and it is aliased from `0x2200 0000` to `0x23FF FFFF`. It is dedicated to the bit access of SRAM memory locations.

- Another bit-banding region starts at `0x4000 0000` and ends at `0x400F FFFF`, which is dedicated to the memory mapping of peripherals, from `0x4200 0000` to `0x43FF FFFF`.

Define two macros in C that allow to easily compute bit-band alias addresses:

```c
/* MEMORY BIT-BANDING */
// Define base address of bit-band
#define BITBAND_SRAM_BASE 0x20000000
```

```
// Define base address of alias band
#define ALIAS_SRAM_BASE 0x22000000
// Convert SRAM address to alias region
#define BITBAND_SRAM(a,b) ((ALIAS_SRAM_BASE + ((uint32_t)&(a)-
BITBAND_SRAM_BASE)*32 + (b*4)))

/* PERIPHERAL BIT-BANDING */
// Define base address of peripheral bit-band
#define BITBAND_PERI_BASE 0x40000000
// Define base address of peripheral alias band
#define ALIAS_PERI_BASE 0x42000000
// Convert PERI address to alias region
#define BITBAND_PERI(a,b) ((ALIAS_PERI_BASE + ((uint32_t)a-BITBAND_PERI_BASE)*32
+ (b*4)))
```
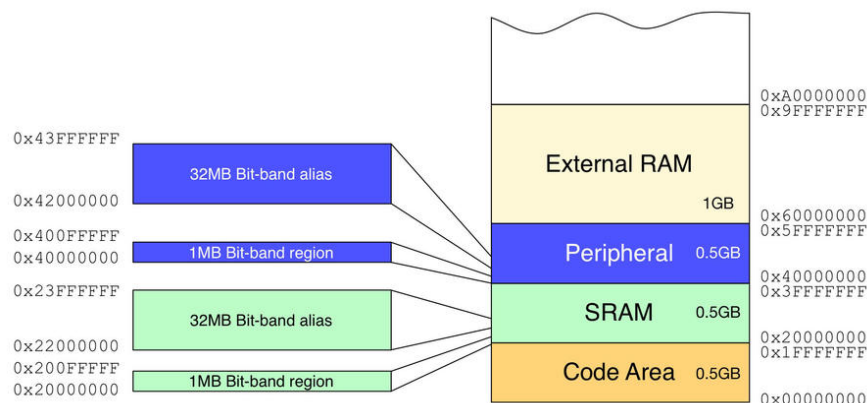
Example that quickly modifies the state of *PIN5* of the *GPIOA* port as follows:

```
#define GPIOA_PERH_ADDR 0x40020000
#define ODR_ADDR_OFF 0x14

uint32_t *GPIOA_ODR = GPIOA_PERH_ADDR + ODR_ADDR_OFF
uint32_t *GPIOA_PIN5 = BITBAND_PERI(GPIOA_ODR, 5);

*GPIOA_PIN5 = 0x1; // Turns GPIO HIGH
```


Memory Map for Bit-banding Area

1.4. *Thumb* Instruction Set

ARM Cortext-M processors provide a 32-bit instructions set, not only allows for a rich set of instructions, but also guarantees the best performance. However, memory footprint of the firmware has bigger cost. To address such issues, ARM introduced the *Thumb* 16-bit instruction set which is transparently expanded to full 32-bit ARM instructions in real time, without performance loss. Afterwards, ARM introduced the *Thumb-2* instruction set, which is a mix of 16 and 32-bit instruction sets in one operation state.

1.5. Instruction pipeline

Before an instruction is executed, the CPU has to fetch it from memory and decode it. So, it has 3 stages to complete an instruction. Modern CPUs introduce a way to parallelize these operations in order to increase their instructions throughput. The basic instruction cycle is broken up into a series of steps, as if the instructions traveled along a *pipeline.*
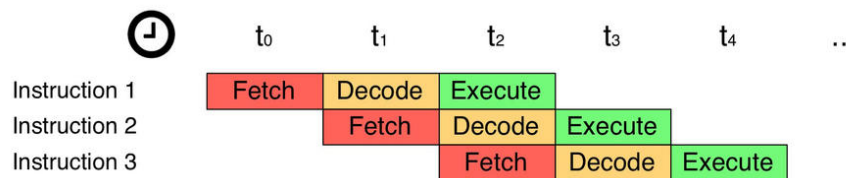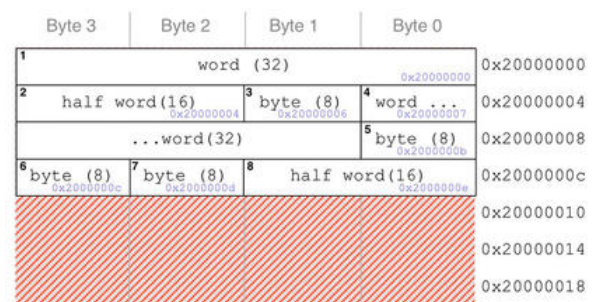

3-stage instruction pipeline

When dealing with pipelines, branching is an issue to be addressed. When branching causes the invalidation of pipeline streams, the last two instructions which have been loaded into the pipeline will be discarded.

1.6. Memory alignment
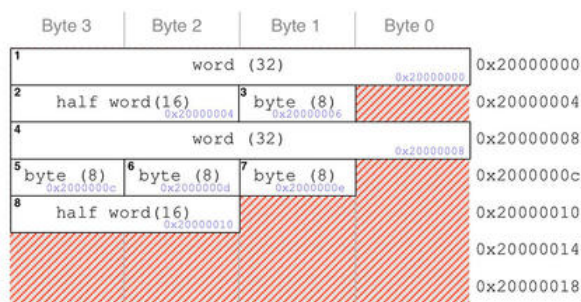
Aligned and Unaligned memory access

ARM based CPUs are traditionally capable of accessing byte (8-bit), half word (16-bit) and word (32-bit) signed and unsigned variables, without increasing the number of assembly instructions as it happens on 8-bit MCU architectures which reads byte by byte. Aligned memory access causes a waste of memory locations.

1.7. Interrupts and Exceptions

Interrupts and exceptions are asynchronous events that alter the program flow. When an exception or an interrupt occurs, the CPU suspends the execution of the current task, saves its context (that is, its stack pointer) and starts the execution of a routine designed to handle the interrupting event. This routine is called Exception Handler in case of exceptions and Interrupt Service Routine (ISR) in case of an interrupt. After the exception or interrupt has been handled,

the CPU resumes the previous execution flow, and the previous task can continue its execution. In the ARM architecture, interrupts are one type of exception.

- Interrupts are usually generated from on-chip peripherals (e.g., a timer) or external inputs (e.g. a tactile switch connected to a GPIO), and in some cases they can be triggered by software.
- Exceptions are, instead, related to software execution, and the CPU itself can be a source of exceptions.

Each exception (and hence interrupt) has a number which uniquely identifies it. Cortex-M cores has pre-defined exception table which contains the addresses of function to handle those exceptions.

Number	Exception Type	Priority	Function
1	Reset	-3	Reset
2	NMI	-2	Non-Maskable Interrupt
3	Hard Fault	-1	All faults that hang the processor
4	Memory Fault	Configurable	Memory issue
5	Bus Fault	Configurable	Data bus issue
6	Usage Fault	Configurable	Data bus issue
7 ~ 10	Reserved	-	Reserved
11	SVCall	Configurable	System service call (SVC instruction)
12	Debug	Configurable	Debug monitor (via SWD)
13	Reserved	-	Reserved
14	PendSV	Configurable	Pending request for System Service call
15	SysTick	Configurable	System Timer
16 ~ 240	IRQ	Configurable	Interrupt Request

1.8. System Timer

Cortex-M based processors can optionally provide a System Timer, also known as *SysTick* which is a 24-bit down-counting timer used to provide a system tick for Real Time Operating Systems (RTOS). It is used to generate periodic interrupts to scheduled tasks, or measure delay. When the timer reach zero, it fires an interrupt number 15, as seen the be Interrupt Table above.

1.9. Clock source and power mode

Cortex-M processors provide several levels of power management which can be set via *System Control Register* (SCR).

- Run mode: full clock speed, all using peripherals are activated

- Sleep mode: reduced clock speed, some peripherals are suspended

- Deep sleep mode: clock is stopped, need external event to wake-up

1.10. CMSIS for SW development

Cortex Microcontroller Software Interface Standard (CMSIS) is a vendor-independent hardware abstraction layer for the Cortex-M processor series and specifies debugger interfaces. The CMSIS consists of the following components:

- CMSIS-CORE: API for the Cortex-M processor core and peripherals

- CMSIS-Driver: defines generic peripheral driver interfaces for middleware making them reusable across supported devices

- CMSIS-DSP: API for process signal and data such as fixed-point, single precision floating-point

- CMSIS-RTOS API: Common API for Real-Time Operating Systems

- CMSIS-Pack: a set of collections which includes source, header, library files, documentation, flash programming algorithms, source code templates and example projects

- CMSIS-SVD: System View Description for Peripherals

- CMSIS-DAP: Debug Access Port

2. STM32 Microcontrollers

STM32 is a broad range of ARM Cortex-M microcontrollers divided in nine sub-families. Internally, each microcontroller consists of the processor core, static RAM, flash memory, debugging interface, and various peripherals.

Here are advantages of using STM32 MCUs:

- Cortex-M based MCUs with large community, free tool-chain, and many shared knowledge articles

- Pin-to-Pin compatibility for most of STM32 MCUs, which helps to change the MCU while keeping pin assignments

- 5V tolerant means it can interface with other devices which do not use 3.3V without using level shifter

- Cheap is an advantage of using STM32 MCUs with ARM based processors and supported RTOS
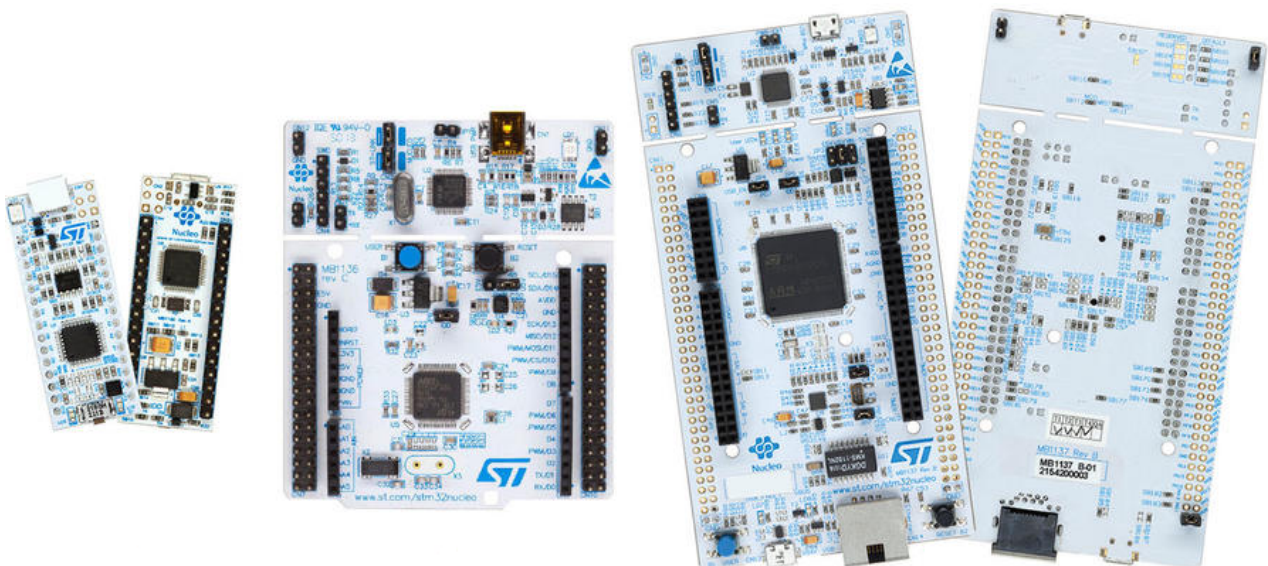
- Integrated boot loader is shipped with internal ROM which allows to reprogram the internal flash memory using some communication peripherals
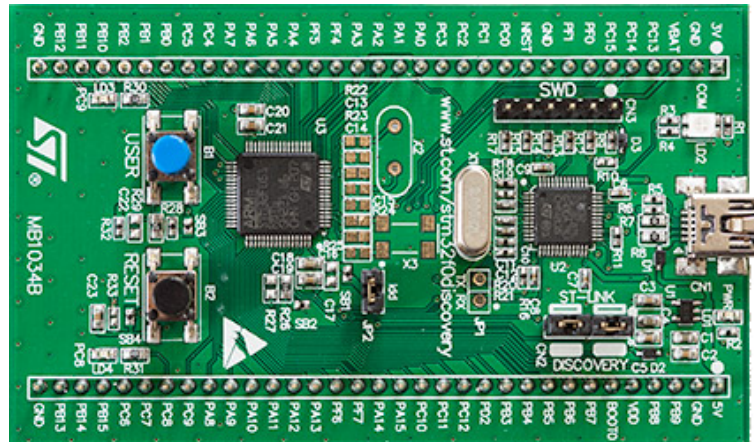
STM32 F051 Discovery Board

STM32 L0538 Discovery Board

STM32 Nucleo family boards

2.1. Cortex-M comparison

A table from ARM.

Feature	Cortex-M0	Cortex-M0+	Cortex-M3	Cortex-M4	Cortex-M33	Cortex-M7
ISA	Armv6-M	Armv6-M	Armv7-M	Armv7-M	Armv8-M Mainline	Armv7-M
	Thumb, Thumb-2					
Pipeline stages	3	2	3	3	3	6
Memory Protection Unit	No	Yes	Yes	Yes	Yes	Yes
Maximum MPU regions	0	8	8	8	16	16
Trace (ETM or MTB)	No	MTB	ETMv3	ETMv3	MTB and/or ETMv4	ETMv4
DSP	No	No	No	Yes	Yes	Yes
Floating point hardware	No	No	No	Yes	Yes	Yes
Bus protocol	AHB Lite	AHB Lite	AHB Lite, APB	AHB Lite, APB	AHB5	AXI4, AHB Lite, APB, TCM
Maximum # external interrupts	32	32	240	240	480	240
CMSIS Support	Yes					

2.2. STM32 MCUs comparison

Compare the core, frequency and memory

Type	Family	Core	Max Frequency	Flash
High Performance	STM32H7	Cortex-M7 - Corte -M4	480 MHz - 240 MHz	1 to 2 Mbytes
	STM32F7	Cortex-M7	216 MHz	256 Kbytes to 2 Mbytes
	STM32F4	Cortex-M4	180 MHz	64 Kbytes to 2 Mbytes
	STM32F2	Cortex-M3	120 MHz	128 Kbytes to 1 Mbyte
Mainstream	STM32G4	Cortex-M4	170 MHz	32 to 512 Kbytes
	STM32F3	Cortex-M4	72 MHz	16 to 512 Kbytes

Type	Family	Core	Max Frequency	Flash
	STM32F1	Cortex-M3	72 MHz	16 Kbytes to 1 Mbyte
	STM32G0	Cortex-M0+	64 MHz	16 to 512 Kbytes
	STM32F0	Cortex-M0	48 MHz	16 to 256 Kbytes
Ultra-low-power	STM32L5	Cortex-M33	110 MHz	256 to 512 Kbytes
	STM32L4+	Cortex-M4	120 MHz	512 Kbytes to 2 Mbytes
	STM32L4	Cortex-M4	80 MHz	64 Kbytes to 1 Mbyte
	STM32L1	Cortex-M3	32 MHz	32 to 512 Kbytes
	STM32L0	Cortex-M0+	32 MHz	8 to 192 Kbytes
Wireless	STM32WB	Cortex-M4 - Cortex-M0+	64 MHz - 32 MHz	256 Kbytes to 1 Mbyte
	STM32WL	Cortex-M4	48 MHz	64 Kbytes to 256 Kbytes

Comments