Embedded Systems Bare-Metal Programming Ground Up (STM 32)

Session 1

Notes:

→ The reference manual is a reference manual.

It has all the information about all the registers in the microcontroller chip, and it gives an explanation for each of the bits in all the registers.

→ Datasheet gives summary about microcontroller as well.

→ User manual describes how things like LED pins and push button are connected to the chip on board.

Goal 1: toggle LED

mindset: where is led connected? Port: Pin:

(create address from scratch)

→ " If you see that chip on top of the board, you have so many pins coming out of the four sides of the of the chip.

These pins are grouped into ports.

So there is a port a, port b, port c, etc. So each port has a number of pins, therefore you hear things like PA1, this means port A pin one. PB2 mean in Port B, pin 2 or PA2, port A pin 2, PD14, meaning Port D pin 14.

So whenever you want to locate a pin, you have to know it's port on its pin number. "

→ for STM32f411CEU6, USER LED is connected to PC13.

meaning: Port: C Pin: 13

→ GPIO is peripheral

→ UL = unsigned long

→ explain what buses are, why do the peripherals need buses?

The reason this is just because every part of the microcontroller, every peripheral, every module needs access to the clock needs to be clocked.

The buses carry the clock to the various part of the microcontroller and the various buses.

two buses: AHB - Advanced high performance bus *

APB - Advanced Peripheral bus

through the AHP bus, we can access a particular peripheral, which less clock cycles, meaning AHB buses are faster than APB.

→ ANALOGY: Each port/GPIO consider as HOUSE ADDRESS, each house has registers consider as ROOM in the HOUSE ADDRESS.

→ " Right, so we spoke about the buses being, you know, transporters of clock to the various peripherals.

Now there is another module which is in charge of enabling each bus to transport a clock.

This module is known as the **RCC module** over here and RCC stands for resetting clock control.

So RCC resetting clock control and this the address range, it's also connected to the AHB1 bus. "

→ next step, to locate the registers we need within these peripherals. (meaning, we have identified the HOME ADDRESSES, next, we have to identify the ROOM NO.s)

RCC → enable clock for AHB1 → AHB1 EN Register → IO port A clock enable → set bit to 1 → gives GPIOAEN

GPIOA base address

Each GPIO have atleast 2 registers: 1. direction register (i/p or o/p)

2. data register (take i/p or send o/p)

aim: to find these regsiters from the range of addresses provided for GPIOA

IN STM32, DIRECTION REGISTER ALSO CALLED AS MODE REGISTER.

GPIOA_BASE_ADDRESS → add offset to find direction register → find bits in mode register corresponding to led pin →set to output mode (here 01)

GPIOA_BASE_ADDRESS → add offset to find output data register → find bit corresponding to led pin → set to 1 to turn on

IMPORTANT NOTE:

TYPECAST REGISTERS TO VOLATILE ENDPOINT AND THEN DEREFERENCE THE POINTER

like so,

#define gpioa_od_r (*(**volatile unsigned int** *)(gpioa_base + od_r_offset))

THIS is because:
It's done so the compiler directly reads/writes the **actual hardware register at that fixed memory address** without optimizing it away, because the register's value can change anytime.

NEW TERM: "Friendly Programming"

Eg. rcc ahb1en reg =0b 0000 0000 0000 1111 0110 0000 0000 0000

rcc ahb1en reg | = gpioA en → gives →0b 0000 0000 0000 1111 0110 0000 0000 0001

instead of if:

rcc ahb1en reg = gpioA en → gives →0b 0000 0000 0000 0000 0000 0000 0000 0001
-preserving the previous bits while modifying the reqd bit.

- 3 steps:

/* 1. enable clock access to gpioa */

/* 2. set pa5 as output pin */

/* 3. set pa5 high */

Toggle (^) : xor operation

a ^= b → a = a XOR b

0 XOR 1 = 1
1 XOR 1 = 0

→ ANOTHER Method: using TYPEDEF STRUCTS

where each struct has the registers we need and the rest are removed, but to maintaong the address consistency we use DUMMY[] arrays.

this is so that we dont need to define the registers every single time.

use pointers to accces members of the struct like so,

#define RCC ((RCC_Typedef*) RCC_base)

RCC→ahb1enr |=gpioaen

*(The only difference between this code and previous code is how we initialised the registers of the peripheral and gpio)

Session 2

NOTES:

1. we start off with basic knowledge of GPIO one must know of:

- PA1 = Port A Pin 1
- GPIO - General purpose Input Output

  SPIO - Special or alternate function of the pin.

  we configure the gpio to perform a special purpose or alternate function. we usually do this when we are trying to configure the peripherals such as the UART, PWM.
- Registers: Direction Register - set as i/p or o/p (in STM32 known as MODE REG)

  Data register - write to pin or read from pin
- buses: AHB and APB
- Clock Sources

  on chip Rc Oscillator - least precise

  externall crystal = more precise

  Phase locked loop (PLL) - u can program the clock

2. Developing GPIO output driver

   in this lesson, we set up our chip headers file and directly configured our registers using the header file instead of define the base addresses and offset everytime, significantly decreasing the code size.
3. using BSSR (Bit set Reset Register)

   in this lesson we saw how we can replace the output data register with the bit set reset register (BSSR) to toggle or turn the LED on and off.
4. Developing Gpio input driver

   in this lesson we saw, how to configure a Push Button, to read the data from its input data register (IDR) and use it to turn the LED off when its on by default.

```
int main(void)
{

    /*Enable clock access to GPIOA and GPIOC*/
    RCC→AHB1ENR |=GPIOAEN;
    RCC→AHB1ENR |=GPIOCEN;

    /*Set PA5 as output pin*/
    GPIOA→MODER |=(1U<<10);
    GPIOA→MODER &=~(1U<<11);

    /*Set PC13 as input pin*/
    GPIOC→MODER &=~(1U<<26);
    GPIOC→MODER &=~(1U<<27);
```

```
    while(1)
    {

        /*Check if BTN is pressed*/
        if(GPIOC->IDR & BTN_PIN)
        {
            /*Turn on led*/
        GPIOA->BSRR = LED_PIN;
        }
        else{
            /*Turn off led*/
        GPIOA->BSRR = (1U<<21);
        }

    }
}
```

Session 3

Notes:

1. Overview of UART

- difference between serial and parallel communication

  serial - transmits 1 bit at a time

  parallel - transmits 8 bits at the same time

  uart - uses serial communication method

- serial data communications - 2 methods -

  (i) Synchronous (clock is transmitted with the data)

  (ii) Asynchronous (No clock is transmitted. Tx and Rx agree on the speed for the data transmission (baudrate))

- UART - Universal Asynchronous Receiver Transmitter

  USART - Universal Synchronous Asynchronous Receiver Transmitter

  (usually what is described in reference manual)

- Transmission modes: Duplex, Simplex, half duplex, full duplex
- Protocol summary: In asynchronous transmission, each byte is packed between start and stop bits.

  Start bit - always 1 bit, always value 0

  Stop bit - either 1 or 2 bit, always value 1

- Configuration Parameters:

  -baudrate this is the connection speed and it is expressed in bits per second

  -stop bits. The stop bits is a parameter that we would have to configure because there are two options. One or two bits. we need no configure

  the start bit because it is required and its value is always zero.

  - parity.

  We would have to indicate the parity mode we want, whether we want odd parity or even parity and parity is use for error checking.

2nd Video: Getting into Coding

- we’re going to nominate USART2 as it is directly connected to the USB of the debugger ST-Link v2 that we’re using. We’re going to communicate between the MCU and the computer.
- Else if u use any other usart then we’ll need a usart to usb converter like an FTDI chip.
- USART2 → connected to APB1 bus → check APB1 register to find where USART2 is connected →bit 17
- enable GPIO to ALternate function mode → find in datasheet

Table 9. Alternate function mapping

Port		AF00	AF01	AF02	AF03	AF04	AF05	AF06	AF07	AF08	AF09	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2C1/I2C2/ I2C3	SPI1/I2S1S PI2/ I2S2/SPI3/ I2S3	SPI2/I2S2/ SPI3/ I2S3/SPI4/ I2S4/SPI5/ I2S5	SPI3/I2S3/ USART1/ USART2	USART6	I2C2/ I2C3	OTG1_FS		SDIO			
Port A	PA0	-	TIM2_CH1/ TIM2_ETR	TIM5_CH1	-	-	-	-	USART2_ CTS	-	-	-	-	-	-	-	EVENT OUT
	PA1	-	TIM2_CH2	TIM5_CH2	-	-	SPI4_MOSI /I2S4_SD	-	USART2_ RTS	-	-	-	-	-	-	-	EVENT OUT
	PA2	-	TIM2_CH3	TIM5_CH3	TIM9_CH1	-	I2S2_CKIN	-	USART2_ TX	-	-	-	-	-	-	-	EVENT OUT
	PA3	-	TIM2_CH4	TIM5_CH4	TIM9_CH2	-	I2S2_MCK	-	USART2_ RX	-	-	-	-	-	-	-	EVENT OUT
	PA4	-	-	-	-	-	SPI1_NSS/I 2S1_WS	SPI3_NSS/I2 S3_WS	USART2_ CK	-	-	-	-	-	-	-	EVENT OUT
	PA5	-	TIM2_CH1/ TIM2_ETR	-	-	-	SPI1_SCK/I 2S1_CK	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA6	-	TIM1_BKIN	TIM3_CH1	-	-	SPI1_MISO	I2S2_MCK	-	-	-	-	-	SDIO_ CMD	-	-	EVENT OUT
	PA7	-	TIM1_CH1N	TIM3_CH2	-	-	SPI1_MOSI /I2S1_SD	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA8	MCO_1	TIM1_CH1	-	-	I2C3_ SCL	-	-	USART1_ CK	-	-	USB_FS_ SOF	-	SDIO_ D1	-	-	EVENT OUT
	PA9	-	TIM1_CH2	-	-	I2C3_ SMBA	-	-	USART1_ TX	-	-	USB_FS_ VBUS	-	SDIO_ D2	-	-	EVENT OUT
	PA10	-	TIM1_CH3	-	-	-	-	SPI5_MOSI/I 2S5_SD	USART1_ RX	-	-	USB_FS_ ID	-	-	-	-	EVENT OUT
	PA11	-	TIM1_CH4	-	-	-	-	SPI4_MISO	USART1_ CTS	USART6_ TX	-	USB_FS_ DM	-	-	-	-	EVENT OUT
	PA12	-	TIM1_ETR	-	-	-	-	SPI5_MISO	USART1_ RTS	USART6_ RX	-	USB_FS_ DP	-	-	-	-	EVENT OUT
	PA13	JTMS-SWDIO	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA14	JTCK-SWCLK	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA15	JTDI	TIM2_CH1/ TIM2_ETR	-	-	-	SPI1_NSS/I 2S1_WS	SPI3_NSS/I2 S3_WS	USART1_ TX	-	-	-	-	-	-	-	EVENT OUT

- to configure the baudrate:

  bus freq and baudrate that we want as argument

  Bd_rate = ((PeriphClk +BaudRate)/2U)/BaudRate);

→ system freq by default = 16MHz

→ in the clock tree, the system clock is taken and it is divided by a value and then what is derived after this division is used as clock for the buses. In default, if we've not configured the clock tree, the divider for the buses is 1. So this means that our peripheral clock for APB1 has same freq as system.

→ uart module config registers → CR - control registers

SR - Status registers

-

#include "uart.h"

#define GPIOAEN (1U<<0)

#define UART2EN (1U<<17)

#define CR1_TE (1U<<3)

#define CR1_RE (1U<<2)

#define CR1_UE (1U<<13)

#define SR_TXE (1U<<7)

#define SR_RXNE (1U<<5)

#define SYSFREQ 16000000

#define APB1_CLK SYSFREQ

#define UART_BAUDRATE 115200

static void uart_set_baudrate(**USART_TypeDef** *USARTx,**uint32_t** PeriphClk, **uint32_t** BaudRate);

static uint16_t compute_uart_bd(**uint32_t** PeriphClk, **uint32_t** BaudRate);

void uart2_tx_init(**void**);

void uart2_write(**int** ch);

int __io_putchar(**int** ch){

```
uart2_write(ch);
return ch;
}
void uart2_rxtx_init(void){
/***Configure uart gpio pin****/
/**1.Enable clock access to gpioa***/
RCC->AHB1ENR|=GPIOAEN;
/***2.Set PA2 mode to AF mode ****/
GPIOA->MODER&=~(1U<<4);
GPIOA->MODER|=(1U<<5);
/**set pa3 to af mode**/
GPIOA->MODER&=~(1U<<6);
GPIOA->MODER|=(1U<<7);
/***2.Set PA2 AF type to UART_TX(AF07)****/
/**GPIO AFLR - ref manual - Pin2 -bit 8 to 11**/
GPIOA->AFR[0]|=(1U<<8);
GPIOA->AFR[0]|=(1U<<9);
GPIOA->AFR[0]|=(1U<<10);
GPIOA->AFR[0]&=~(1U<<11);
/**set pa3 af to uart_rx**/
GPIOA->AFR[0]|=(1U<<12);
GPIOA->AFR[0]|=(1U<<13);
GPIOA->AFR[0]|=(1U<<14);
GPIOA->AFR[0]&=~(1U<<15);
/*****Configure UART module ********/
/**Enable clock access to uart2****/
RCC->APB1ENR|=UART2EN;
/***configure baudrate***/
uart_set_baudrate(USART2,APB1_CLK,UART_BAUDRATE);
/***configure the transfer direction***/
USART2->CR1 = (CR1_RE|CR1_TE); //Not using | operate deliberately, clearing all bits except bit 3
/***enable the UART module***/
USART2->CR1|= CR1_UE;
}
void uart2_tx_init(void){
/***Configure uart gpio pin****/
/**1.Enable clock access to gpioa***/
RCC->AHB1ENR|=GPIOAEN;
/***2.Set PA2 mode to AF mode ****/
```

```
GPIOA→MODER&=~(1U<<4);
GPIOA→MODER|=(1U<<5);
/***2.Set PA2 AF type to UART_TX(AF07)****/
/**GPIO AFLR - ref manual - Pin2 -bit 8 to 11**/
GPIOA→AFR[0]|=(1U<<8);
GPIOA→AFR[0]|=(1U<<9);
GPIOA→AFR[0]|=(1U<<10);
GPIOA→AFR[0]&=~(1U<<11);
/*****Configure UART module ********/
/**Enable clock access to uart2****/
RCC→APB1ENR|=UART2EN;
/***configure baudrate***/
uart_set_baudrate(USART2,APB1_CLK,UART_BAUDRATE);
/***configure the transfer direction***/
USART2→CR1 = CR1_TE; //Not using | operate deliberately, clearing all bits except bit 3
/***enable the UART module***/
USART2→CR1|= CR1_UE;
}
char uart2_read(void){
/**make sure receive data register is not empty*/
while (!(USART2→SR & SR_RXNE)){}
return USART2→DR;
}
void uart2_write(int ch){
/***MAke sure TX register is empty b4 putting data in it***/
while (!(USART2→SR & SR_TXE)){}
/**Write to transmit data register***/
USART2→DR=(ch & 0xFF);
}
static void uart_set_baudrate(USART_TypeDef *USARTx,uint32_t PeriphClk, uint32_t BaudRate){
USARTx→BRR=compute_uart_bd(PeriphClk,BaudRate);
}
static uint16_t compute_uart_bd(uint32_t PeriphClk, uint32_t BaudRate){
return ((PeriphClk +(BaudRate/2U))/BaudRate);
}
```

Session 4

ADC

A physical quantity is converted to electrical signals using a device called transducer, and the transducer simply converts a physical quantity to either voltage or current .Transducers

used to generate electrical output are referred to as sensors, sensors for temperatures, velocity,

pressure, light and many other natural physical quantities produce an output that is voltage or sometimes current.

Therefore, we need an analog to digital converter to translate the analog signals to digital numbers

so that the microcontroller can read and process these numbers.

concept of ADC resolution.

An ADC has N- bits resolution where and can be eight, 10, 12, 16 or even 24.

Higher resolution ADC provide a smaller step size where step size is the smallest change that can be detected by an ADC.

And to compute the step size we've got to take the reference voltage into account.

So assuming the reference voltage is 5V, which we simply do five divided by 256 and this gives us

19 and this gives us 19.53mV. A 10-bits ADC has a 1024

step size and the smallest change this 10 bit ADC can detect is 4.88mV

Session 6: Timers

NOTES:

TIMER Uses:

-counting events

-creating delays

-measuring time between event

TIMER vs. COUNTER

clock source is internal → a timer.

When a clock source external such as something fit to the CPU → a counter.

STM32 TIMER USES

-to generate a time base.

-to measure the frequency of an external event. This is known as input capture mode.

-to control an output waveform or to indicate when a period of time has elapsed. This is known as the output compare mode.

-mode that allows the timer to be started in response to a stimulus and to generate a pulse with a programmable length after a programmable delay.

It's called one pulse mode.

TIMER REGISTERS

-TIMx_CNT

Shows the current counter value. Size could be 32-bit or 16-bit depending on timer module.

-TIMX_ARR (Auto Reload Register)

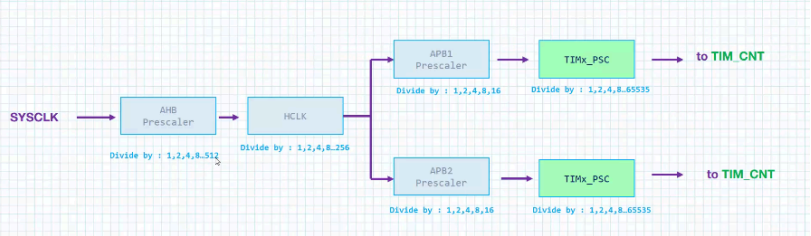
Timer raises a flag and the counter restarts automatically when counter value reaches the value in the auto reload register. The counter is an up counter by default but can also be configured to be a down counter.

TIM2→ARR = 10000;

-TIMx-PSC (Prescaler Register)

The prescaler slows down the counting speed of the timer by dividing the input clk of the timer.

EG: TIM2→PSC=1600-1;



time prescaler determines how fast the time counter increases or decreases.

SIMILAR thing is done is STM32CUBEMx.

CONTROL REGISTER 1 (CR1)

Used to enable or disable the timer.

STATUS REGISTER (SR)

Checking, setting and clearing the flags of the timer.

TIM2→SR & 1; //CHECKING THE REGISTER UPDATE INTERRUPT FLAG (UIF)

TIM2→SR &=~1; //CLEARING THE REGISTER UIF

CAPTURE/COMPARE REGISTER (CCR1,CCR2,CCR3,CCR4)

One ccr for each of the 4 channels

CAPTURE COMPARE MODE REGISTER 1 (CCMR1)

configuring capture/compare functionality for CH1 and CH2

CAPTURE COMPARE MODE REGISTER 2 (CCMR2)

configuring capture/compare functionality for CH3 and CH4

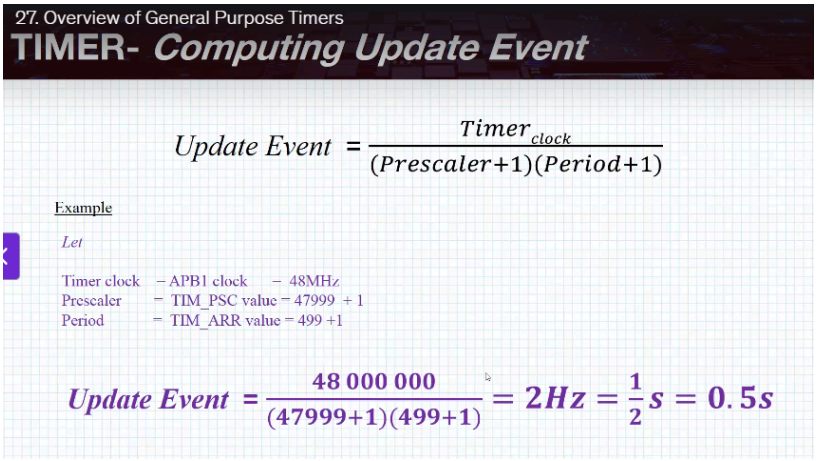
CAPTURE/COMPARE ENABLE REGISTER (CCER)

used to enable any of the timer channels either as input capture or output compare

TERMS:

UPDATE EVENT: when timeout occurs or how long it takes for flag to be raised

FORMULA :



PERIOD: Value loaded into auto reload register.

UP COUNTER: Counts from zero to set value.

DOWN COUNTER: Counts from set value to zero down.

DEVELOPING THE GENERAL PURPOSE TIMER DRIVER

-TYPES: general purpose timers, advanced timers, basic timers.

these are timers provided by the STmicroelectronics.

To keep it simple, the psc is how fast the timer shall work and ARR how long should it count.

DEVELOPING THE TIMER OUTPUT COMPARE DRIVER

-allows us to toggle the GPIO directly when a particular period elapsed.

- allows the timer to directly toggle a hardware pin when a particular period elapses.

DEVELOPING THE TIMER INPUT CAPTURE DRIVER

What to do if I want to set input capture for falling edge?

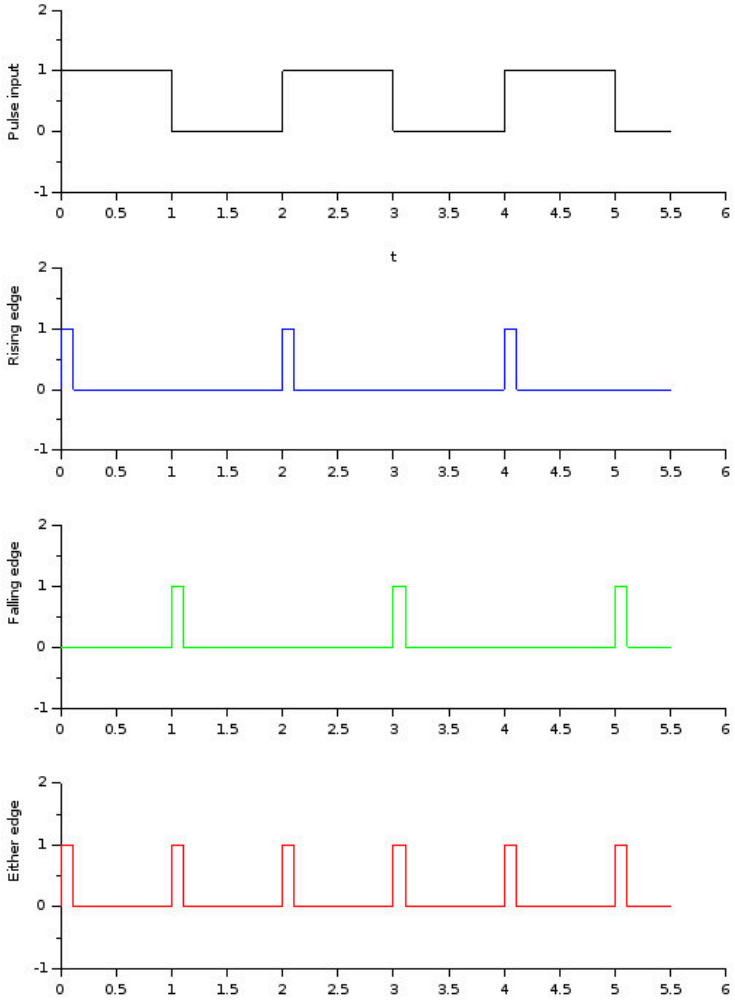
in CCER register, set CCxP to 1 will give falling edge

Why ARR register value is not mentioned in input capture function, if so how CCR captures the value?

the ARR values is used to determine how long the timer shall counts.

Since we are using it in input capture mode, it is recommended to use maximum value offered by the timer.

Why timestamp variable is updated by 2000? It must be updated by 1000 as timing set is 1 sec.


Pulse input
Rising edge
Falling edge
Either edge
t
2
1
0
-1
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6

Session 7: Interrupts

NOTES:

INTERRUPT MODE: when a module needs to be serviced, it's notifies the CPU by sending an interrupt signal. When the CPU receives the signal, the CPU interrupts whatever it is doing and services that module and a module here could be the ADC, the GPIO that UART, et cetera.

POLLING METHOD: The CPU continuously monitors the status of the given module. And when a particular status condition is met, the CPU then services the module.

INTERRUPT SERVICE ROUTINE (ISR) OR INTERRUPT HANDLER:

the function that gets executed when an interrupt occurs

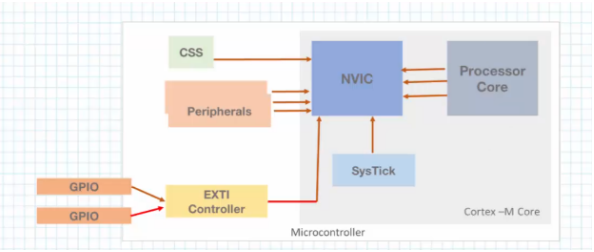
NESTED VECTOR INTERRUPT CONTROLLER (NVIC)

The NVIC is a dedicated hardware inside the cortex microcontroller it is responsible for handling interrupts.

-interrupt from the process or the process over here are called exceptions

-Interrupt from outside the processor core known as hardware Exceptions or interrupted request often written as IRQ.

VECTOR TABLE: contains the addresses of all the interrupt handlers and the exception handlers such that when an interrupt triggered the process, the processor simply goes to the victor table and finds the address of the ISR, then executes that.



EXTI LINES (External Interrupt Lines):

All GPIO pins are connected to EXTI lines which is connected to NVIC.

Interrupts : *External Interrupt (EXTI) lines*

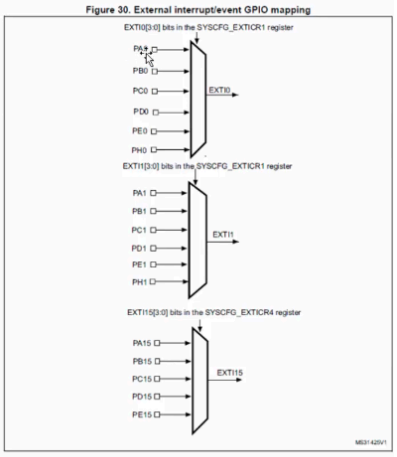
- GPIO pins are connected to EXTI lines
- It possible to enable interrupt for any GPIO pin
- Multiple pins share the same EXTI line

- **Pin 0 of every Port** *is connected* ***EXTI0_IRQ***
- **Pin 1 of every Port** *is connected* ***EXTI1_IRQ***
- **Pin 2 of every Port** *is connected* ***EXTI2_IRQ***
- **Pin 3 of every Port** *is connected* ***EXTI3_IRQ***

...

This means we cannot have PB0 and PA0 as input interrupt pins at the same time since they are connected to the same multiplexer i.e. EXTI0

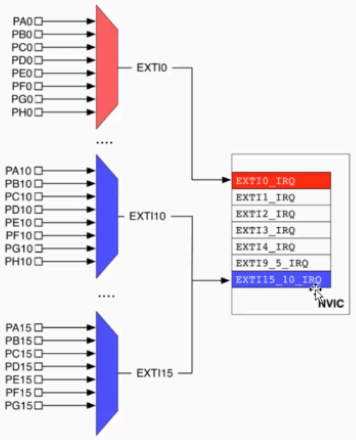
Same for PC4 and PB4 at the same time, etc.



Interrupts : *External Interrupt (EXTI) lines*

- Pins 10 to 15 share the same IRQ inside the NVIC and therefore are serviced by the same Interrupt Service Routine (ISR)

- Application code must be able to find which pin from 10 to 15 generated the interrupt.



INTERRUPT STATES:

DISABLED: default state

ENABLED: interrupt is enabled

PENDING: interrupt is waiting to be serviced

ACTIVE: interrupt is currently being serviced

PRIORITIY:

-priority allows us to set which interrutp should execute first.

-they also allow us to set which interrupt can interrupt which.

-lower the number = higher the priority

-2 types of priority in stm32f4: “fixed” and “settable”

-priority of each interrupt is defined using one of the INTERRUPT PRIORITY REGISTERS (IPR)

-each IRQ uses 8 bits inside a single IPR register
-one IPR = 4 diff IRQ

-total 60 IPR registers → therfore, 4×60 =240 IRQ

-8 bits to configure priority → 2^8= 255 priority levels

however we only use upper 4 bits to configure priority = 2^4 =16 priority level

- ***IPRn*** = *IRQ(4n+3), IRQ(4n+2),IRQ(4n +1) and IRQ(4n)*

first one→used in bare metal programming

2nd one→used in HAL programming

Developing the GPIO Interrupt Driver

-EXTI is part of the SYSCONFIG module → hence must enable clock access to it as well.

SYSCFG connect to APB2 bus

NOTE:

PA0 default state =**floating**

Floating =**random HIGH/LOW transitions**

Falling edges appear → EXTI0 fires → serial monitor prints without you touching anything.

enable internal pull up resistor by adding these lines in **pa0_exti_init():**

```
/* enable pull-up for PA0 */
GPIOA→PUPDR &= ~(1U << 0);   // clear PUPDR0
GPIOA→PUPDR &= ~(1U << 1);
GPIOA→PUPDR |=  (1U << 0);   // PU = 01
```

SYSTICK ONLY AVAILABLE IN CORTEX M4 USER GUIDE!!

TICKINT = enables SysTick exception request.

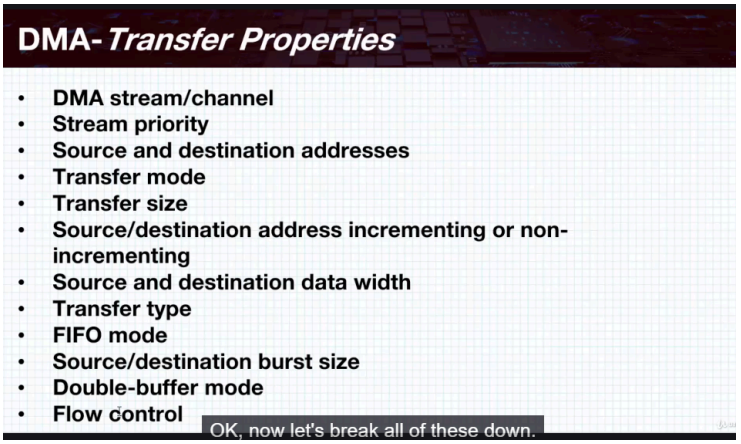
Session 8: DMA

NOTES:

-DMA allows data transfers to take place in the background without the intervention of the cortex and processor.

-During this operation, the main processor can execute other tasks, and it is only interrupted when a whole data block is available for processing.

So because of this, large amount of data can be transferred with no major impact on the system performance.

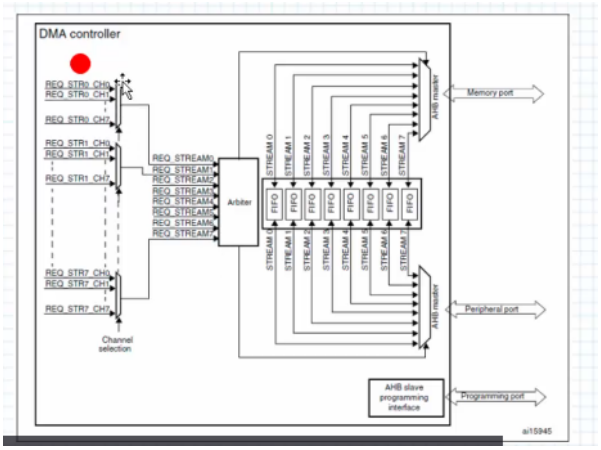


DMA -2 ports

1. MEMORY PORT
2. PERIPHERAL PORT

2 DMA modules - each module has 8 DIFFERENT STREAMS.

-each stream is dedicated to managing memory access request from one or more peripherals.



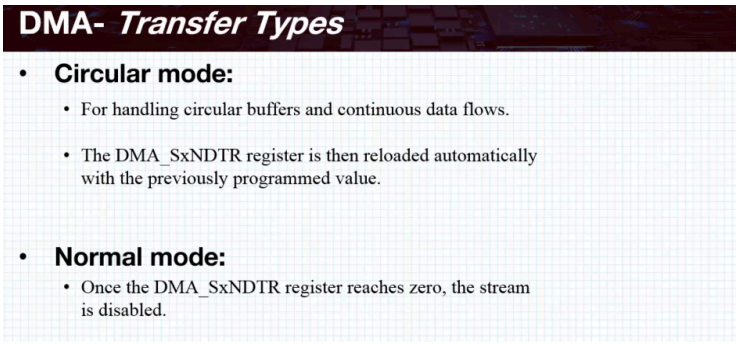
-CHANNEL

each stream has 8 channels. at a time only one channel can be active in a single stream!

-ARBITER

arbiter is for handling the priority between the DMA streams.

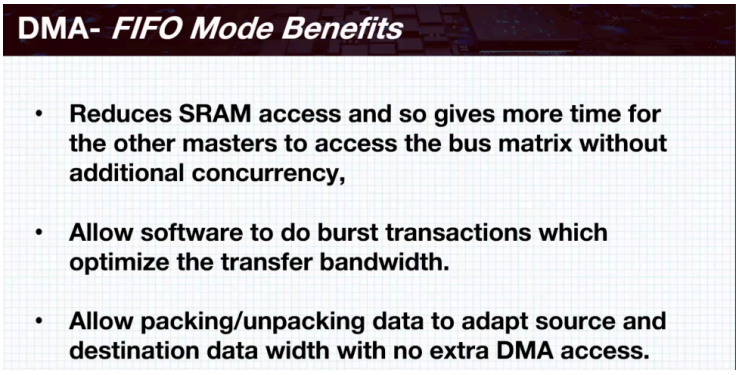
-there are 4 priority levels. if two or more DMA streams have the same software priority level, then the hardware priority is used and in the hardware priority, stream zero has priority over stream one, stream one has priority over stream two, stream two has priority over stream three and so on and so forth.



circular mode - used for continuously storing sensor values like ADC...

FIFO MODULE:

1. The FIFO is used to store the data coming from the source before transmitting to the destination for a short period of time.
2. If the DMA FIFO is enabled , We can decide to have data packing and unpacking, or we can decide to use the DMA in burst mode.
3. The Configure DMA FIFO threshold defines the DMA memory port request time so we can configure threshold for the FIFO such that we say if the FIFO is half full, tell us if it is quarter full tell us if it is three fourth full tell us.



note: only dma2 module allows memory to memory data transfer.

DEVELOPING DMA DRIVER FOR UART:

Table 27. DMA1 request mapping (STM32F411xC/E)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX	I2C1_TX	SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX	-	SPI3_TX
Channel 1	I2C1_RX	I2C3_RX	-	-	-	I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1	-	I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4	-	-	-	-	-	USART2_RX	USART2_TX	-
Channel 5	-	-	TIM3_CH4 TIM3_UP	-	TIM3_CH1 TIM3_TRIG	TIM3_CH2	-	TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	I2C3_TX	TIM5_UP	USART2_RX
Channel 7	-	-	I2C2_RX	I2C2_RX	-	-	-	I2C2_TX

stream 6 channel 4 = usart2_tx in DMA1

Session 9: I2C

NOTES:

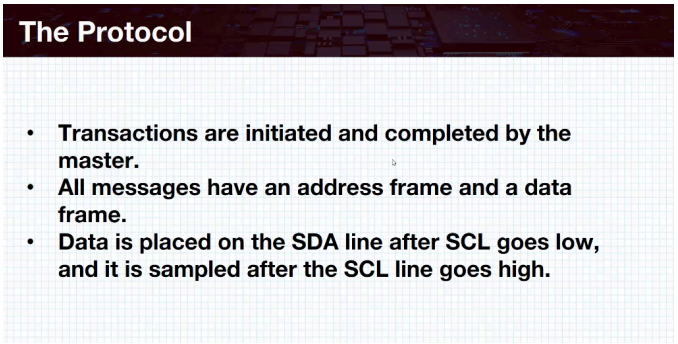
I2C - Inter Integrated Circuit

SCL -Serial clock line -synchronizing data transfer between the master and the slave

SDA - Serial Data - the data line

OPERATION MODES:

1. Master Transmitter
2. Master Receiver
3. Slave Transmitter
4. Slave Receiver



-Start and stop conditions are always generated by the master.

I2C Duty Cycle:

specifies the ratio between t(low) and t(high) of the I2C SCL line

by choosing the appropriate duty cycle, we can prescale the peripheral clock to achieve the desiredI2C speed.

TO NOTE:

For configuring i2c clock to standard mode:

As per the datasheet CCR[11:0] info you can see that the calculation has been done for 8MHz.

Sm mode or SMBus:

Thigh = CCR * TPCLK1

Tlow = CCR * TPCLK1

For instance: in Sm mode, to generate a 100 kHz SCL frequency:If FREQ = 08, TPCLK1 = 125 ns so CCR must be programmed with 0x28

*(0x28 ⇔ 40d x 125 ns = 5000 ns.)*Here,FREQ = 16MHz, hence TPCLK1 = 1/16MHz = 0.0000000625 (62.5ns)THigh = 5000ns (based on the 8MHz example)now need to find **CCR = THigh/TPCLK1 = 5000ns/62ns = 80**

STM32F4 has only one DATA REGISTER for both transmit TXE and receiving RXE data. In F7, however, has 2 data registers - transmit data register and receiver data register.

BTF Flag → Byte transfer finished → in I2C_SR1.

Session 10: SPI

NOTES:

SPI - Serial Peripheral Interface

SPI is a synchronous and full-duplex communication protocol between a master and several slaves.

full duplex, we mean the communication goes both ways.

Four Pins in total:

SDIN - MOSI Master out Slave in - used to send data from the master to the slave

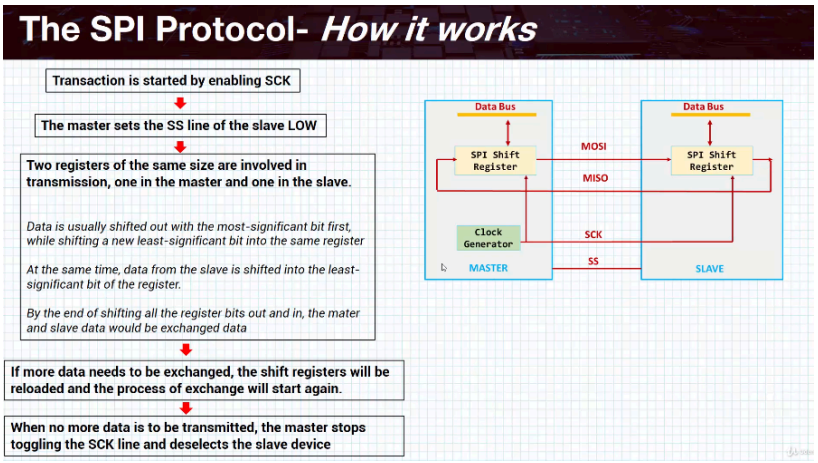
SDO - MISO Master in Slave out - used to send data from the slave device to the master

SCLK - SCK (Synchronize data transfer between two chips)

CE - SS - Slave Select (used to initiate and terminate data transfer) - used to select a particular slave in the SPI multiple slaves configuration.

Modes - Single Slave, Multiple Slaves

SPI slaves have no addresses.



During Transmission:

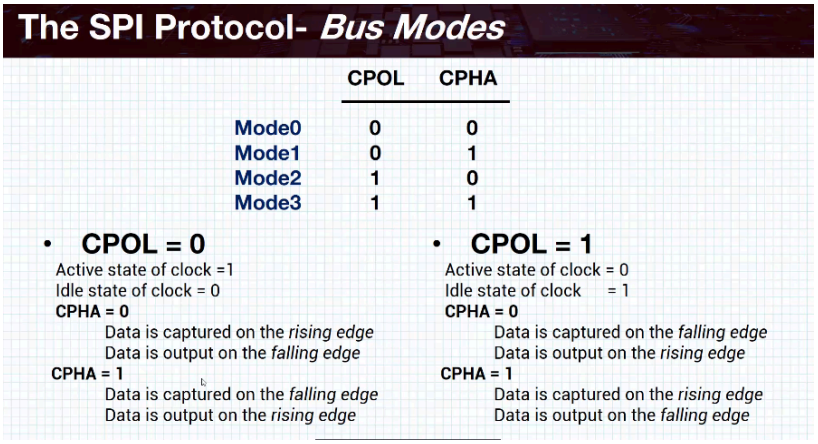
SCK is toggled continuously, data is sent on rising edge of SCK.

SS is set LOW.

CLOCK PHASE AND CLOCK POLARITY

Master and Slave device must agree on CPHA AND CPOL.

Combns of CPOL AND CPHA are referred to as SPI bus modes.



CPOL=0 Means sampling on the first edge

CPOL=1 Means sampling on the second edge

NSS

2 types:

1. NSS Software mode: SS line driven internally by firmware.
2. NSS Hardware mode: dedicated GPIO pin is used to drive SS line.

   2 types:

   i. NSS output enabled: used when the device operates in master mode.

   ii. NSS output disabled: allows multi master capability for devices operating in master mode.

TI mode

So if we decide to operate in TI mode, then NSS Hardware must be used. Over here,

The clock phase and clock polarity are forced to conform with the Texas instrument protocol requirement in

this mode NSS signal pulses at the end of every transmitted, every transmitted byte.

def spi1_config(){

}

⇒ entirely configured using CONTROL REGISTER 1.

Some Theory

RISC Design Philosophy

ARM as a company

So ARM is a company based in Cambridge U.K. ARM is a design firm. They do not manufacture any hardware or silicon.

All they do is they come up with the designs of the architecture and then they sell the license out

to companies such as Apple, Samsung, Huawei, Qualcomm, Texas Instruments etc.

RISC - Reduced Instruction Set

CISC - Complex Instruction Set

-think of instruction set as the set of instructions in a language.

-The RISC design philosophy is aimed at simple but powerful instructions that execute within a single cycle at clock speed.

-RISC focuses on reducing complexity of the instructions performed by hardware because it is easier to provide greater flexibility and intelligence in software rather than in hardware.

As a result, greater demand is placed on the compiler.

RISC philosophy is implemented with four cardinal design rules in mind.

1. There has to be a reduced number of instructions and each instruction must be executed in a single cycle.
2. To make sure every processor cycle executes an instruction, instructions are executed in parallel using pipeline

This is important because in CISC there are times the processor just runs while waiting for data or

instruction to be ready from memory.

3. RISC machines provide a large number of general purpose registers.

Any register can either be data or address. Registers are fast low cost stores that we use to do processing. CISC architecture registers have dedicated purposes therefore they are not as flexible.

Also in RISC architecture.

The processor operates on data held in the registers.

This is one of the other reasons

RISC processors are capable of one instruction per cycle.

ARM Design Philsophy

-ARM processes are specifically designed to be small to reduce power consumption and extend battery operation. Essentially for applications such as mobile phones and other embedded devices.

-ARM's core competency lies in taking RISC architecture, high code density and power efficiency and putting it in one processor.

-Code density simply means the amount of space that an executable program takes up

your memory. High code density means the program takes less space.

-In RISC systems due to its philosophy of simplying instructions,

greater complexity is placed on the compiler. The compiler has to take the simple instructions and understand the complex functions those instructions are supposed to perform and generate a simple binary file which will be downloaded onto the processors.

Therefore complexity is placed on the compiler in the RISC system.

-In CISC architecture the compiler does minimal work. A greater complexity is placed on the processor

In effect the processor must be designed complex enough to understand the complex files generated by the compiler.

So this is the fundamental difference between the CISC and RISC.

Embedded Systems with ARM Processors

Four main hardware components of embedded device:

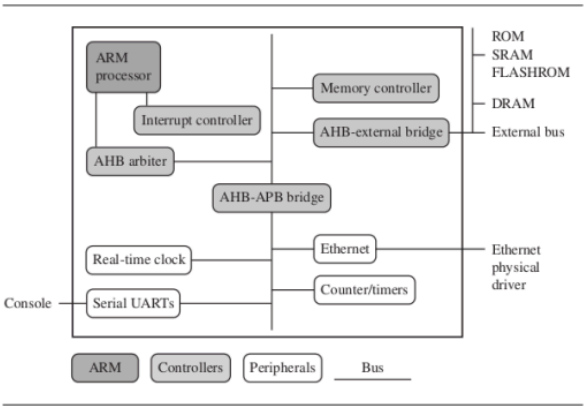


Figure 1.2 An example of an ARM-based embedded device, a microcontroller.

Bus Tech. and AMBA Bus Protocol

two classes of devices connected to the bus:

1. **bus master** is often a logical device capable of initiating data transfer with another device
2. **bus slave** is capable of only responding to a transfer request from the master.

A bus has two architecture levels:

1. first level is known as the physical level,

The physical level deals with the electrical characteristics of the bus and things like that bus width, for example 16-bit, 32-bit 8-bit etc..

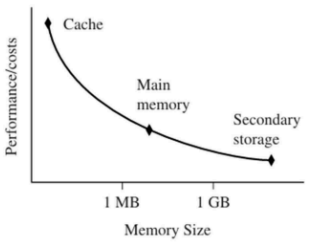
2. second level is known as the protocol level.

The bus protocol dictates the rules the bus master and the bus slave uses to communicate.

AMBA stands for Advanced Micro-controller Bus Architecture and it's the widely adopted on-chip architecture used for ARM processors.

MEMORY:

- **Memory**
  - **Hierarchy**
  - **Types**
  - **Width**



Instruction size	8-bit memory	16-bit memory	32-bit memory
ARM 32-bit	4 cycles	2 cycles	1 cycle
Thumb 16-bit	2 cycles	1 cycle	1 cycle

fastest mem : cache and located closer to processor
slowest mem : secondary storage set further away from processor

-cache provides an overall increase in performance and it comes with a loss of predictable execution time though.

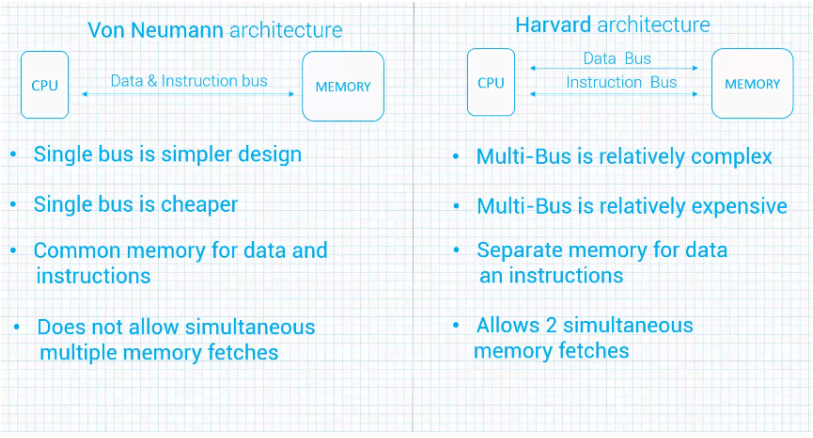
The memory controller simply connects the different types of memory we talked about to the processor. For example when the power is turned on the memory controller may be configured in hardware to allow certain memory devices to be active. These memory devices allow the initialization code to be executed.

Some memory devices must be set up in software though something like that DRAM. We first have to set up the memory timings and the refresh rates using software before it can be accessed.

The interrupt controller, it basically determines which peripheral can access

the processor at specific times.

There are two types of interrupt controllers available in ARM. The standard interrupt control and the vectored interrupt controller.



Von – Neumann Architecture vs Harvard Architecture

Von – Neumann Architecture	Harvard Architecture.
• Codes and data in the same memory	• Codes and data in separate memory
• Require less space	• Require more space
• CPU cannot access data and instructions in same time	• CPU can access data and instructions in same time
• Comparatively slower	• Comparatively faster
• Only a single memory connected to the CPU	• Data memory (RAM) and Program memory (ROM) are separately connected to the CPU
• Using only one channel/bus	• Using separate busses (two busses)

CACHE

Cache is a block of fast memory placed between the core and the main memory.

It allows for more efficient fetches from some memory types with a cache the processor core can run for majority of the time without having to wait for data from slow external memory.

TCM

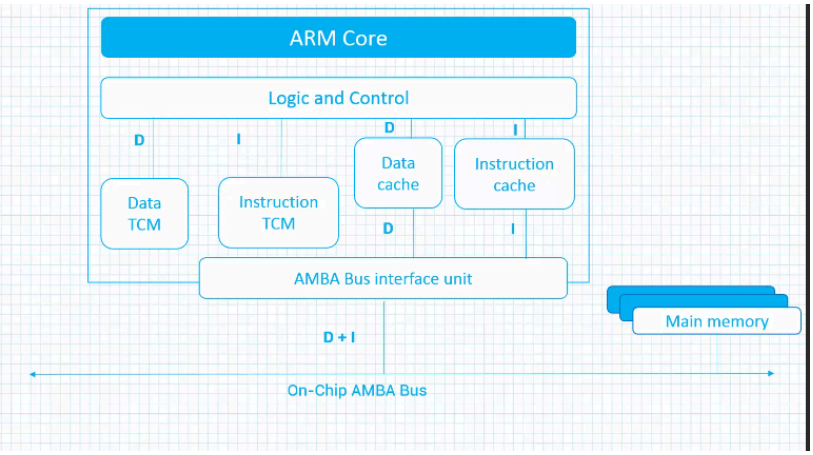
Real time systems are deterministic systems they require execution time to be predictable.

This can be achieved using the second form of fast accessible memory called the tightly coupled memory or the TCM. The TCM is a fast SRAM located close to the core and guarantees the clock cycles required to fetch instructions or data. TCMs appear as memory in address map and it can be accessed very fast.

So in effect they work like the cache.

- In fact by using the TCMs and the cache ARM processors have both extra performance boost in our predictable realtime response.

This is what the modified architecture looks like.



MEMORY MANAGEMENT:

MPU, the memory protection unit and the MPU provides a form of limited protection.

And the second is the MMU the memory management unit which provides a much more comprehensive protection.

MPU:

MPU employs a rather simple system that uses a limited number of memory regions. These regions as we shall see, are controlled by a set off special registers and each region is defined with specific access permissions.

MPU is often used for systems that require memory protection but the systems do not have very complex memory maps.

So with a system that has a simple memory map and requires the memory protection the MP you can simply be used.

MMU:

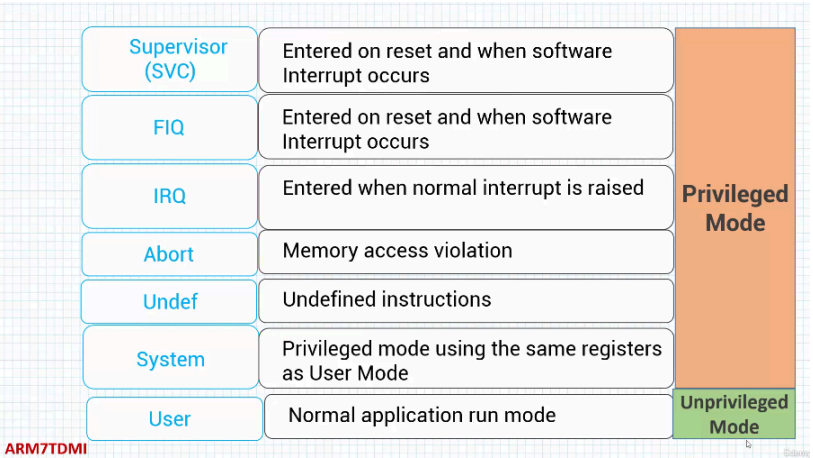
physical to virtual address conversion.

The Programmer's Model

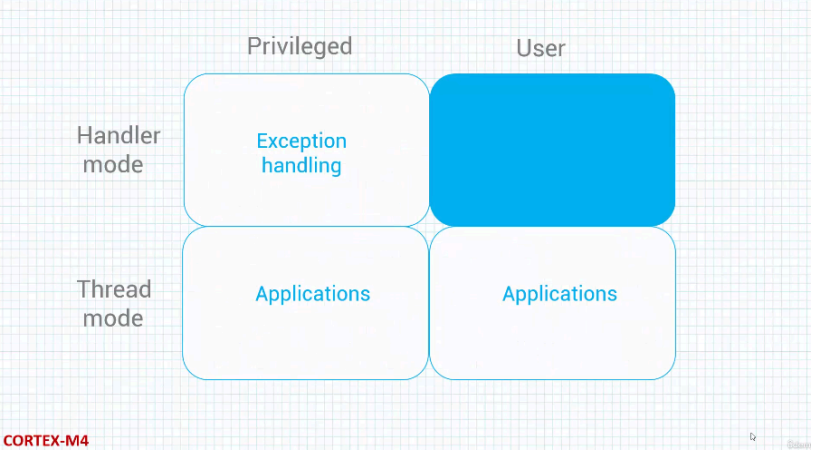
From a program perspective what is necessary is a model of the device. Something that describes not only the way the processor controlled but also the features available to you as the programmer, such us where can data be stored and what happens when the machine detects invalid instruction, where the registers are stacked etc. This is what's known as the programmer's model.

Processor modes:

for ARM7TDMI:



for Cortex M4



similarly Registers:

ARM7TDMI:

User/System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

Banked Register

The ARM7TDMI processor has a total of 37 registers and 30 of these registers are general purpose registers.

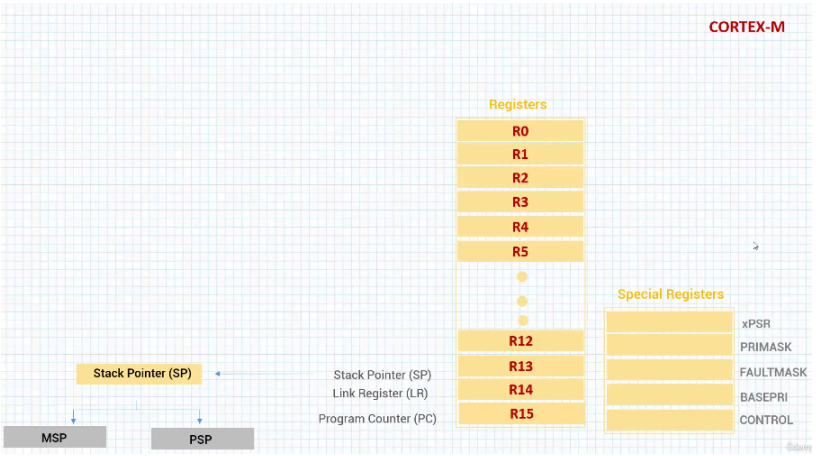
6 of them are status registers and 1 is the program counter register. The ARM7 registers are arranged in partially overlaping banks

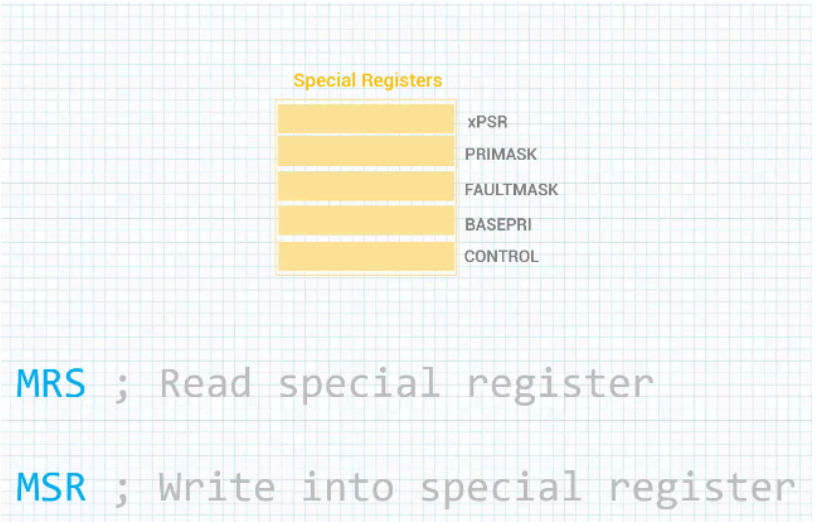
- The General Purpose registers (R0 –R12) contain data or addresses.
- R13 is known as the stack pointer (SP) and it points to the top element of the stack.
- R14 is known as the Link Register (LR) and it's used to store the return location for functions.
- R15 is known as the Program Counter (PC). It is readable and writable.
  - Read returns current instruction address plus 4

Cortex-M

The Cortex-M has 17 General Purpose Registers, 1 Status Register and

3 Interrupt Mask Registers. R0 through R12 are the general purpose registers.





xPSR=CPSR -APSR, EPSR, IPSR

CORTEX-A

The A, in Cortex-A,

stands for Application. The Cortex-A family

focuses on high end applications such as smartphones tablets, desktop computers etc. These are products

with very high processing power requirements. The Cortex-A core generally has large caches and additional

blocks for graphics and operating system support.

CORTEX-R

The R in the Cortex-R stands for real time.

And the Cortex-R

processors are designed for those applications where real time or safety constraints play a major role.

We find them in the breaking mechanisms of cars, in modems, in X-ray machines, in other medical devices

Pull-Up or Pull-Down Resistor

Why Pull-Up or Pull-Down Resistor Needed?

Digital input pins must be clearly HIGH (1) or LOW (0). Without a resistor, the pin "floats" – it has no fixed voltage. EMI from nearby wires, motors, or lights, plus tiny leakage currents, make the pin toggle randomly between 0 and 1. This causes false readings, crashes, or glitches in your circuit.

What is Pull-Up Resistor?

A pull-up resistor is used to keep a digital signal line at a stable HIGH level when no device is actively driving the line. Without a pull-up, the pin can float and randomly change between HIGH and LOW due to noise. By connecting the signal to VCC through a resistor (commonly 10 kΩ), the pull-up provides a default HIGH state, while still allowing an external device like a switch connected to ground to pull the line LOW when pressed. This ensures clean, predictable logic transitions and prevents false triggering.

What is Pull-Down Resistor?

A pull-down resistor, on the other hand, holds a signal line at a stable LOW level when nothing else is controlling it. Without it, the input may float and pick up noise, causing unpredictable readings. By connecting the signal to GND through a resistor, the pull-down provides a safe and reliable default LOW state. When an external device, such as a button connected to VCC, is activated, the voltage rises to HIGH without causing a short circuit because the resistor limits the current. Pull-down resistors are commonly used in microcontroller inputs and logic circuits to ensure stable, noise-free operation.

Applications

Common in microcontrollers like Arduino for buttons, I2C buses, or unused pins; many chips have built-in weak pull-ups (e.g., 4.7kΩ). Use pull-ups for open-drain outputs to avoid bus contention, and pull-downs for normally closed switches. Strong pulls (1-10kΩ) suit noisy environments; weak ones (4.7kΩ+) save power in battery circuits.