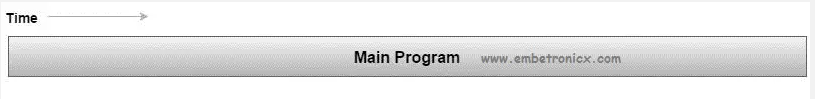
# Embedded Topics

## Interrupt Topics

**What is an Interrupt in OS?**interrupt means stop the continuous progress of (an activity or process)  
  
**What is an Interrupt in OS (Microcontroller)?**same here also. It will stop the process which is running currently and then execute the process in ISR. So it is a signal which may be distributed (or) alter the sequence of execution of the processor. In other words, an interrupt is a signal to the processor emitted by hardware (or) software indicating an event that needs immediate attention.  
  
**what is ISR?**ISR means interrupt service Routine. Whenever an interrupt occurs, the controller completes the execution of the current instruction and starts the execution of an Interrupt Service Routine (ISR) or interrupt handler.  
  
**What is Pooling?**The pooling method simply uses a code section that checks a particular flag (or) Flags, to know that status of operations. The pooling method is always part of the main code and not part of an ISR. The microcontroller keeps checking the status of other devices.  
Pooling is a simpler method to execute and is recommended for the early stages of design where the working of a peripheral is to be verified. It doesn’t involve any priority. Also, the code segment will always execute with a fixed time and in a fixed sequence.  
It is easy to debug and has no effect on the execution of other sections of code. There are no big memory management issues. There is no issues with the stack.  
  
**Interrupt vs Pooling?  
Interrupt:** Interrupt is like a **Shop Keeper.** If one needs a service (or) product, he goes to him of his needs. In case of interrupts, when flags (or) signals are received they notify the controller that they need to be serviced.  
**Pooling**: The pooling method is like a **salesperson**. The salesman goes from door to door while requesting to buy a product (or) service. Similarly, the controller keeps monitoring the flags (or) signals one by one for all devices and provides service to whichever component that needs its service.  
  
**Program execution without Interrupts?**so it only executes the main function & some functions which are called in the main function.  


**Program execution with interrupts?**In this method, it will execute the main function. When an interrupt signal comes, it stops the main function execution & just executes the Current instruction & suddenly it will go to the ISR function and executes that. After execution of ISR & again it will come to the main function.  
A diagram of a company

Description automatically generated  
  
**Types of Interrupts?**Although interrupts have the highest priority than other signals, there are many types of interrupts, but the basic type of interrupts are

1. Hardware Interrupts
2. Software Interrupts

**Hardware Interrupts**:  
If the interrupt is coming from hardware (or) external devices. It is called Hardware interrupt. For Example, from the keyboard we will press the key to do some action this pressing of a key on the keyboard will generate a signal which is given to the processor to do action, such interrupts are called H/W interrupts.  
  
**Software Interrupts**:  
The interrupts which are caused by the software instructions are called S/w interrupts.

Other differentiation is as follows:

1. Maskable Interrupts
2. Non - Maskable Interrupts

**Maskable Interrupts**:  
An interrupt that is disabled (or) ignored by the instructions of CPU are called Maskable interrupts. The interrupts are either edge triggered (or) Level triggered. If interrupt response time is little late also no harm happens, at those case Maskable interrupts are used.  
Example: system input/output (keyboard, mouse) …etc.

**non-Maskable interrupts**:  
An interrupt that can’t be disabled (or) ignored by the instructions of CPU are called non-Maskable interrupts. A non-maskable interrupt is used when response time is critical (or) interrupt never be disabled.  
  
Differences between Maskable & Non Maskable interrupts

|  |  |
| --- | --- |
| **Maskable interrupts** | **Non Maskable interrupts** |
| Maskable interrupt is the Hardware interrupt that can be disabled (or) ignored by the instructions of the CPU | Non Maskable interrupt is the Hardware interrupt that can’t be disabled (or) ignored by the instructions of the CPU |
| When maskable interrupt occur, it can be handled after executing of current instruction. | When non-maskable interrupts occurs, the current instruction and status are stored in stack for CPU to handle the interrupt. |
| Maskable interrupts helps to handle low priority tasks. | Non maskable interrupts helps to handle high priority tasks such as Watchdog timer. |
| Maskable interrupts used to interface peripheral device. (keyboard, input, output) | Non maskable interrupts are used for emergency purpose e.g.: system failure, smoke detector … etc. |
| Response time is high | Response time is low |
| Interrupt can be masked (or ) made pending | Interrput can’t be masked (or) can’t pending |

**Interrupt Latency**:  
When an interrupt occurs, the service of the interrupt by executing the ISR may not start immediately by context switching. The time interval between the occurrence of interrupt and the start of execution of the ISR is called interrupt latency.

## Watchdog Timer

TBD – To Be Done

## What happens when press reset button in the microcontroller?

As you are an embedded engineer, have you though anytime, about what happens when you press the reset button (reset sequence in ARM cortex M4 (or) boot sequence in ARM cortex M4) ? how it is calling the main() function in your application automatically?   
  
**Reset Sequence in ARM Cortex-M4**?  
Before starting this we must know about the microcontroller memory architecture. Please find the below image to know about the memory map of ARM Cortex-M4  
A close-up of a chart

Description automatically generated  
  
Here, we have a **code** region, where we are going to write our final binary. That memory is starting from 0x00000000 to 0x1fffffff. Now we will forget about other regions. We will take only the SRAM and Code region. We know that the code region has the final output of our program (.hex (or) .bin etc). SRAM will be having stack, heap, global RW variables and static variables .. etc.  
  
**Memory layout of the program**?  
Now we have to know how final output of our program is stored in **code region**. The below image will explain how our program is stored in the code region.  
A diagram of a data processing process

Description automatically generated  
  
As you can see form the above image, the final output of our program will be ordered in this way by using a linker  
  
**vector table of ARM Cortex\_M4**:  
That vector table will contain all the locations of the execution and interrupt ISR.  
A close-up of a document

Description automatically generated  
  
So, the 0x00000000 address contains the initial stack pointer value. Then 0x00000004 has the address of the reset handler. Once you press the reset button (or) power up the controller, the below things will happen.  
  
**What happens when you press the reset button in ARM Cortex-M4**?

1. PC (Program Counter) will be loaded with 0x00000000. So will start from the address 0x00000000
2. Since the address has an initial Stack pointer value. It will be fetched to the MSP (Main Stack Pointer ) so, that value will be starting from stack.
3. Then PC will be loaded with Reset Handler’s address.  
   **Note**: These above 3 steps are done by the hardware (this architecture-specific)
4. After that, the reset handler will perform the below operations.  
   Initialize the system  
   Copy the initialized global variables, static variable (.data) to SRAM.  
   Copy the un-initialized data (.bss) to SRAM and initialize to 0  
   it calls main()

This is how your main() is getting called while power-up (or) press the reset button. You may understand clearly if you see he below execution.  
  
**Reset Sequence in ARM Cortex-M4 – Example execution**  
I am using IAR IDE & STM32. Each controller has a specific startup file that has all the ISR’s address (vector table). In my case, I have startup\_stm3214xx.s file. I am going to run through a debugger (J-link). Using that we will step by step. I have attached the some startup files.  
A screenshot of a computer

Description automatically generated  
  
The above code is not a full code. I just provided some lines for your understanding. If you see the above code, the vector tables first place contains Stack. This is the initial **SP**. Then after that, they have passed the **Reset\_Handler**’s address.  
  
Now I am start debugging

1. Refer to the below image, the cursor points to **Register\_Handler** & **SP** also points to 0x20000000. And other registers are initialized to zero.  
   A screenshot of a computer

   Description automatically generated
2. Then using the BLX command, it is branching to SystemInit function. There it will initialize the system.  
   once its done with the system initialize, it will load the \_\_iar\_program\_start to Ro register.  
   A screenshot of a computer

   Description automatically generated
3. Since we are using IAR it using \_\_iar\_program\_start function. This is IAR specific function  
   You can not find the source of the \_\_iar\_program\_start. But you can see those kinds of stuff in the disassembly window of IAR.  
   Therefore , you can see that it is going into \_\_iar\_init\_vfp, which initialize the hardware floating point unit (FPU) checkout the below image.  
   A screenshot of a computer

   Description automatically generated
4. Then it will jump to main() function.  
   A screenshot of a computer

   Description automatically generated
5. In main(), it calls, \_\_low\_level\_init. This function is intended to perform a customized initialization of you H/W that either must occur very early on (or) can speed up the startup process. For example, if you plan to increase the CPU clock speed. It’s better to do it sooner. So that the reset of the startup code will execute faster. Check out below image.  
   A screenshot of a computer

   Description automatically generated  
     
   as you can see by the test of R0 register, \_\_low\_level\_init returns a value that determines whether to go straight to calling main (or) perform a data initialization. It returns a non-zero value. So the function \_\_iar\_date\_init3 is called.  
   A screenshot of a computer program

   Description automatically generated  
     
   \_\_iar\_data\_init3 will copy data to SRAM (.data). And also copy the .bss data to SRAM and initialize it 0. With the help of \_\_iar\_zero\_init3  
   A screenshot of a computer

   Description automatically generated  
   After that, it calls \_call\_main function. In that \_call\_main function. We are calling the original main() function.   
   A screenshot of a computer

   Description automatically generated

**Summary**:

1. When you press the reset button, it will copy the Stack pointer to MSP (Main Stack Pointer) from the location of 0x00000000
2. Then it moves to Reset\_handler
3. In the Reset\_handler, it will initialize the hardware (system), then copy the initialized data (initialized global & static variable) to SRAM.
4. Then copy the unininitialzed data to SRAM and initialize to zero
5. Finally it calls our main() function.

NOTE: this process applicable when you do not have a bootloader. If you have a bootloader, then it will execute in different way

**What happened when you have a boot loader**?  
When your project has a boot loader, then you will be having 2 binaries. One is Bootloader image and another one is an application. The bootloader will be placed in the 0x00000000 with its vector table. The application will be placed in another area of the flash memory with it’s vector table.

1. When you press the reset button, it will start from the bootloader. So it will copy the Stack pointer to MSP (main stack pointer) from the location of 0x00000000.
2. Then it moves to the bootloader reset handler
3. The bootloader will do some operations based on your need like check for firmware version, upgrade the firmware, etc
4. Once it has done with its operation, it will jump to the application’s vector table
5. Then it will initialize the MSP (Main Stack Pointer) again
6. Now we have to tell the controller to use **the application’s vector table** instead of using the **bootloaders vector table**. That will be done using the **Vector Table Offset Register**(VTOR)
7. Then it goes to the application’s reset handler. There it will copy and initiate the data segments to the RAM (Global, Static variables)
8. Finally, It moves to the main function of the application.

If you want to learn about the bootloader, then we have provided a separate bootloader series for you.  
<https://embetronicx.com/bootloader-tutorials/>

# Arm-Assembly

## Introduction to ARM Assembly basics

ARM is most widely used processor around us compared to intel. ARM is available in mobiles, TV, STB, routers, IOT.  
  
<https://azeria-labs.com/writing-arm-assembly-part-1/>  
https://twiserandom.com/arm/arm-assembly-language-tutorial/index.html

## ARM – Instruction Format

Example:

1. add r0, r1, r2 #performs r1+r2 & stores the result in r0
2. adds r0, r1, r2 # performs r1+r2 & stores the result in r0, additionally if r1+r2 overflows (or) result of addition is zero (or) if some other conditions occur, a flag bit, which corresponds to the condition which has occurred is set.
3. addsEQ r0, r1, r2 #If the zero condition flag is set, r1 is added to r2, and the result is stored in r0. Additionally a flag is set, for example if a carry (or) overflow (or) other conditions occur

### The Format

Mnemonic{s}{Condition} Rd, Rn, <Operand2>  
**Mnemonic**: is the name of instruction, for example add instruction.  
**{}** : What is between the curly brackets is optional, for example add instruction can optionally be suffixed, with an ‘s’ (or) condition as in ‘EQ’ (or) both.  
This being said, when a Mnemonic, which is the name of an instruction,   
 is suffixed by an ‘**s**’, condition flags are **set**.  
 as in ‘**EQ**’, the condition flags are **checked**.  
**Rd** : is the destination register.  
**Rn** : is any register  
**<Operand2>** : can be any one of any register (Rm), (or) an immediate value as “#2” (or) any register Rm is shifted by an immediate value like ‘#2’ (or) by another register.  
  
Example:

1. add R0, R1, R2 #adds the value in the register R1 to R2 & stores the result to R0 = R1+R2
2. add R0, R1, #1 #adds the value in R1 to the immediate value #1, R0 = R1 +1
3. add R0, R1, R2, LSL #1 #performs the logical shift left on R2, by 1, the result is added to R1; R0= R1 + R2<<1
4. add R0, R1, R2, LSR R3 #performs the logical shift right on the content of R2 by the value in R3, adds the result to R1 and stores to R1; R0 = R1 + R2>>R3

**NOTE/Concern about Immediate values**:  
they written as #1 in decimal (or) hexadecimal #0xFFFFFFFF (or) in binary 0b1  
An ARM32 instruction is 32 bits, **12 bits** out of these 32 bits can be reserved **to represent an immediate** value.

1. These 12 bits are also partitioned into 4 bits, which represent a right rotation.  
    & 8 bits, which represent a value, which range is b/w **0** to **FF**
2. The assembler is in charge, of converting the immediate values that you provide, into these **12 bits**, by rotating to the right the lower 8 bit value, using the rotation bits multiplied by 2 which should give a result equal to the immediate value, other wise an error is generated.
3. Example:  
   mov r0, #0x83FFFF1F # This will cause an error, which states something like invalid operand value, and that an immediate value, must be creatable by rotating the 8 bit number right. Using rotation bits.  
     
   mov r0, #0x80000000 # 0x80000000 can be obtained for example by rotating the binary value 10, to the right by 2 as such the assembler will not complain, and the value 0x80000000 is stored into the register r0.

### The condition Flag

So what are these Condition flags? Basically there are N, Z, C & V and these are just bits, which belong to and set on the current program status register.  
A white rectangular sign with purple text

Description automatically generated  
  
Instruction of the format, Mnmonic {s] as in adds can affect some, (or) all of the condition flags.  
  
**The C flag**:  
C, has the name of “**carry condition flag**”, its value is affected when the result can’t fit into 32-bit range.  
 Example:

1. mov r0, #0xffffffff # move the value 0xffffffff into the register r0  
   mov r1, #0x1 # move the value 0x1 into r1  
   adds r2, r0, r1 # add r0 and r1 & store result into r2 ( addition of 11111111111111111111111111111111 + 00000000000000000000000000000001 which results into 100000000000000000000000000000000 which is 0, and the carry flag is set to 1 )
2. mov r0, #0x2  
   mov r1, #0x1  
   subs r2, r0, r1 # r2 = r0-r1 (ie; 2-1) (00000000000000000000000000000010 – 11111111111111111111111111111111 which is 2’s complement subtraction, 2+(-1), result is 100000000000000000000000000000001 so end up with carry flag being set to 1
3. **Shift Operation**:  
   mov r0, #0b100   
   mov r1, #1  
   lsrs r2, r0, r1 #lsrs, logical shift right, r0 contains (100) in binary ie; in decimal it is 4), right shifted by 1 and stores the result in r2. So the last shifted out bit is ‘0’. Now carry bit is ‘0’
4. mov r0, #0b1   
   mov r1, #1  
   rors r2, r0, r1 # rotate right, Now carry bit is ‘1’
5. **Inline shift**:  
   mov r0, #0x80000000  
   movs r1, r0, LSL #1 # r0 is shifted by 1 and the result is stored into r1 will have zero value, since the last bit shifted out is ‘1’. Then the carry flag is set ‘1’
6. **Compare**:  
   mov r0, #1  
   mov r1, #2  
   cmp r0, r1 #compare 2 registers r0 &r1, (performs r0-r1) (ie; 1+(-2) ) and the carry flag is set to ‘0’

**The V flag**:  
v, has the name overflow flag, this is only related to 2’s complement addition & subtraction  
Basically, in case of **addition**, when adding 2 positive numbers, and the result is negative , this flag is set, adding 2 negative numbers, and the result is positive, this bit is set.  
  
Incase of, subtraction, when subtracting a negative number form a positive number, and the result is negative, this bit is set & when subtracting a positive number, from a negative number, and the result is positive this bit is set.  
  
So, this is related to truncation happening, which causes wrapping.  
Example:

1. Examples of adding positive numbers to positive numbers, and negative numbers to negative numbers, that causes an overflow to happen.  
     
   mov r0, #0x40000000  
   adds r1, r0, #0x40000000   
   # This operation actually amounts to the addition of 01000000000000000000000000000000, 01000000000000000000000000000000 which results in: 0000000000000000000000000000000 So adding two positive numbers, resulted in a negative number, this is overflow, so the overflow bit is set to 1.Additionally, and since the result fits into 32 bits, the carry flag is set to 0.
2. mov r0, #0x80000000  
   mov r1, #0x80000000  
   adds r2, r1, r0  
   # The binary representation of # 0x80000000, is 10000000000000000000000000000000 In two's complement 32 bit, this is a negative number. 0x80000000 is being added to itself as in 10000000000000000000000000000000 + 10000000000000000000000000000000 and the result is 100000000000000000000000000000000 Adding two negative numbers, resulted in a positive one, because of truncation, this is overflow, and the overflow bit is set to 1. Additionally the carry bit is set to 1, since the result does not fit into 32 bits.
3. Examples of subtracting negative numbers, from positive ones, and positive numbers from negative ones, that lead to overflow.   
   mov r0, #0x0  
   mov r1, #0x80000000  
   subs r2, r0, r1  
     
   # r1 is 0x80000000, which is a negative number, and is being subtracted from r0 which is 0, which is a positive number, of if you prefer a non negative one. When using two's complement subtraction, this is as performing the addition, of r0, with the two's complement of r1, which amounts to: 00000000000000000000000000000000 + 10000000000000000000000000000000 and which results into 10000000000000000000000000000000 Subtracting a negative number from a positive number, must yield a positive number, the result is negative, as such this is overflow, and the overflow bit is set to 1. Additionally the carry flag is set to 0, because the result does fit 32 bits.

**The N Flag**:  
N has the name of “**negative condition flag**” it is set to the most significant bit of result.  
So, if the most significant bit is ‘1’ the N flag is set to ‘1’,  
Otherwise the MSB is ‘0’, then the flag is set to ‘0’.  
Example:  
mov r0, #0xC0000000  
mov r1, #0xC0000000  
adds r2, r0, r1 #Adding 0xC000000 with itself, amounts (ie; 11000000000000000000000000000000 + 11000000000000000000000000000000) which results in 110000000000000000000000000000000. The MSB being set to ‘1’means the negative flag is set to ‘1’. Since the result is larger than 32 bits, it means that the carry flag is set to ‘1’, 2 negative numbers are added the result is negative number, So overflow doesn’t apply, and the overflow flag is set to ‘0’  
  
**The Z flag:**Z has the name of “**Zero Condition flag**”, if the result of instruction is zero, then it set to ‘1’, and not zero it is set to ‘0’  
Example:  
mov r0, #0x1  
mov r1, #0x1  
subs r2, r0, r1 # subtracting 0x1 from 0x1,amounts to adding :00000000000000000000000000000001 + 11111111111111111111111111111111 which results into:100000000000000000000000000000000 the result being **zero**, the **Zero flag is set to 1** , additionally the **Carry flag is set to 1,** since the result doesn’t **fit into 32 bits**. **The negative flag is set to 0**, since **the most significant bit is 0**.**The overflow bit is set to zero**, since we are subtracting two positive numbers, so overflow does not apply

### The Conditions

EQ, NE:  
**EQ**: means that an instruction will proceed, if and only if the ‘z’ flag is set to 1  
Example:

1. mov r0, #0x1  
   cmp r0, #0x1  
   addEQ r1, r0, #0x1  
     
   #0x1 is moved into r0 & cmp will perform (r0 – 0x1) (0x1 – 0x1)which results into zero flag is set to 1. add having EQ condition will execute, only if the Z-flag is set to 1, which is, hence in this case add does execute r1 will have a value of 0x2.

**NE**: means, only perform the instruction, if the Z flag bit is set to ‘0’  
**CS**: means, that the operation is to be performed, if and only if, the carry flag is set to 1  
**CC**: means, that the operation is to be performed, if and only if, the carry flag is set to 0  
**VS**: when the overflow flag is set to 1,  
**VC**: when the overflow flag is set to 0  
**MI**: stands for minus, MI is executed if the N- flag (negative flag) is set to 1  
**PL**: stands for plus, PL is executed if the N-flag is set to 0  
**GE**: stands for greater (or) equal, and this condition applies when N=V  
**LT**: stands for Less than, and this condition applies when N!=V  
**GT**: means greater than, this condition applies if and only if Z=0 & N=V  
**LE**: stands for less than (or) equal, this condition is applicable if and only if z=1 (or) N!=V  
**HI**: means unsigned Higher, and condition is met only if C=1 & Z=0  
**LS**: means unsinged lower (or) same, and the condition is applicable if and only if , C=0 (or) Z=1  
**HS**: means unsigned higher (or) same, this condition is applicable if and only if c=1  
**LO**: stands for unsigned lower and this condition applies only if c=0  
**AL**: stands for always which means regardless of the condition flags, set into the CPSR register, instruction is executed  
Example: movAL r0, #0x00000000 # AL means always execute the instruction.

### When are condition flags cleared?

The condition flags, are N,Z,C,V. some instruction affect all the condition flags, such as the ‘add’ instruction, whereas others, do affect only few, such as AND, (or)

## Instruction set cheat sheet

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated  
  
A screenshot of a computer

Description automatically generated  
  
A screenshot of a computer

Description automatically generated  
  
A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated  
  
Example:

1. for(int i=0; i<5; i++)  
    // some statements:  
     
   mov r0, #0  
   mov r1, #5  
   for\_loop:  
    cmp r0, r5  
    beq finish  
    #some statements  
    add r0, r0, #1  
    b for\_loop  
   finish:  
    #some statements
2. assembly of do { } while(x>=0)  
   mov r0, #5  
   do\_while:  
    #some statements to execute  
    subs r0, r0, #1  
    bPL do\_while
3. Example of if -else  
   cmp r1, r2  
   bNE else  
    # some statements to execute, if r1 and r2 are equal  
    b after\_if\_else  
   else:  
    #some statements to execute  
    #if r1 and r2 not equal  
   after\_if\_else:  
   # instructions in the if -else block have executes, continue the sequential flow
4. Example of how to use the ‘bl’ & ‘bx’ instructions  
   str lr, [sp, #-4]! #decrement the stack pointer by 4 and push the link register content, where now the stack pointer, points into memory  
     
   bl do\_something #branching is done using ‘bl’ instruction. 1st the link register will be assigned the value of PC+1 so it is going to point to the following add instruction. After that, branching to the do\_something block is performed  
     
   add r0, r0, r1  
   ldr lr, [sp], #4 # load the content pointed by the SP register into the link register & after that increment the stack pointer by 4  
     
   b done # branch to the done section  
   do\_something:  
    mov r0, #0  
    mov r1, #1  
    bx lr # branch back to where the program was executing, since the link register contains the value of the program counter before branching plus one  
    done:

A screenshot of a computer

Description automatically generated

# X86 – Assembly

## X86

Part1 : <https://lnkd.in/eNeM5iF>  
Part2 :<https://lnkd.in/duV7zybf>  
Part3 :<https://lnkd.in/e7WeYY7>