**Lesson# 05: Preprocessor and the "volatile" keyword in C**

* We use preporocessors use to define the names of registers instead of cryptic numbers
* When defining macros(registers)you can use other macros to define it
* For example in code we are going to use the base address of GPIOF register and add the offset and GPIOF base address to create a new register’s address
* Everything boils down to, to change any real world value you need to change the value of address for that you must know the register address
* But vendors like TEXAS instrument already provide macros for stellaris board ina separate file which are .h file
* The file name is lm4f120h5qr.h
* If we see the macros that are being defined in the file and that we defined are significantly different, lets understand this
  + The macros in main.c use unsigned int & Header file use unsigned long but practically in term size they have no difference
  + The actual difference is the **volatile** qualifier/keyword. This keyword use to tell the compiler that a variable’s value may be changed by something OUTSIDE THE CONTROL PROGRAM (eg: Hardware, another thread or interrupt).
  + This prevents compiler from optimizing variable’s access by assuming that the pointer pointed to the location won’t be change unless it explicitly use in the code
  + Actually compiler utilizes the variables to optimize the code that can be observed when you turn on HIGH level optimization
* These volatile keywords are really useful for the I/O Registers
* When I put code into high optimization mode it basically eliminates the affect of counter variable which only introduce the delays doing nothing in the code
* Here you would make the counter variable volatile
* Now find the interested registers with the address

**Lesson #06: Bit-wise operators in C**

* You want the Blue led to remain ON meanwhile make the red led keeps on blinking, that is really simple put the Blue led write line before the while loop
* There is a problem when you set red led inside the while loop, you clear all other LEDs since all LEDs lives inside a single register, thus you need to have a method for setting and clearing the individual bits without disturbing the other bits inside a single register and this is where the **bitwise C operator** comes in
* Lets learn all the bitwise operations on two regsiters with n number of bit:
  + | operator performs comparison on each bit and apply logical OR table method, if at 0th position a register has 1 and then output will be 1 just like OR table, that is what being followed here, the screenshot show loading value as well as also tells **Data Processing instruction from ARM ISA being used here is ORRS**A screenshot of a computer

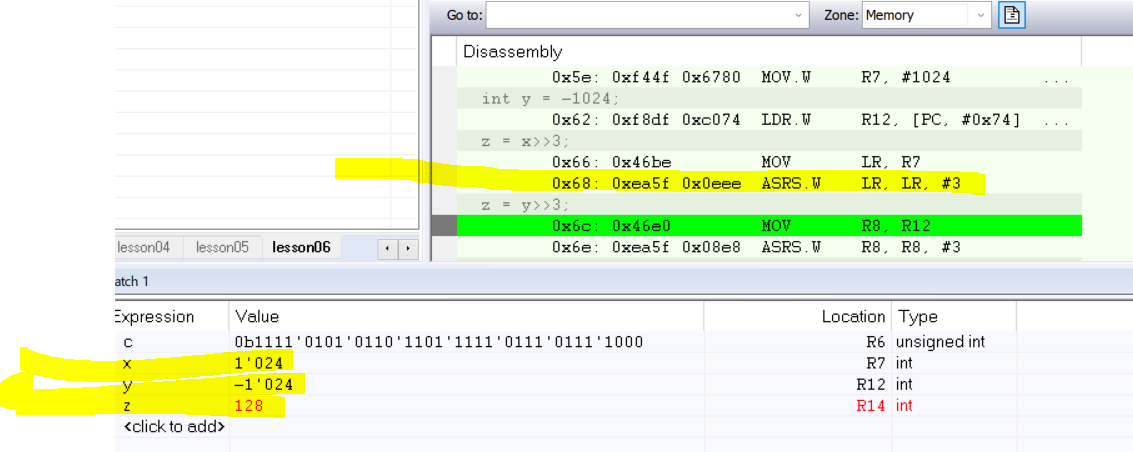
    Description automatically generated
  + Exact same method in Logical AND, this time following AND truth table and **Data Processing Instruction from ARM is ANDS** here, also you would need to recognize that the resultant OR saved in R2 register, this time the R2 value doesn’t get disturbed and new AND operation value being loaded to R3 register even if we use same c variable, means the value being override on c and store in another register.A screenshot of a computer

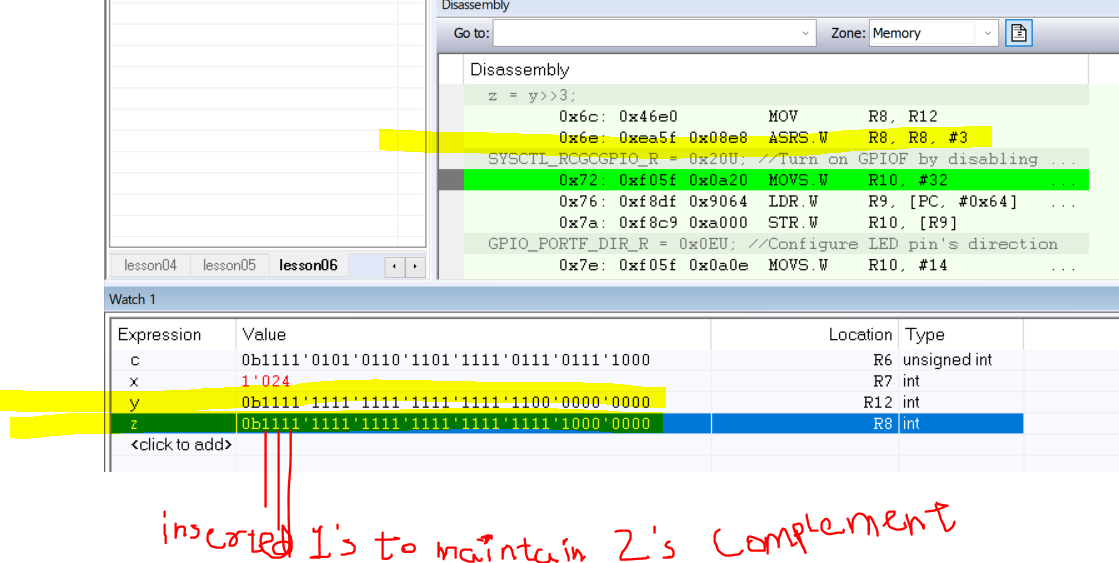
    Description automatically generated
  + In the same way just like AND new register stores the value, EORS single instruction do thatA screenshot of a computer

    Description automatically generated
  + Not operation being performed ona single register whose value being inverted, local NOT uses MVNS which stands for Move NegativeA screenshot of a computer

    Description automatically generated
  + Right shift operation shift the N number of bits specified in code into the MSB side which is left side, which is also equivalent to dividing the integer number of a register by 2 if 1 bit is shifted, if 2 bit is shifted then divide by 2^2 the register at which shift operations being performed, again let understand that this is the way to verify the shift operation. The **LSRS** command which I the shift rotate instruction command in ISA here used, **all the bits of register b are being shifted by 1 to the right**A screenshot of a computer

    Description automatically generated
  + In the same way left shift operation which shifts all the bits to the left and put 0s in LSBs. But you need to remain vigilant as the MSB’s might fall off. You can also verify it by multiplying the register’s integer value with 2^n, where n is the amplitude by which the left shift operation being performed. Here it uses **LSLS instruction to shift the bits** A screenshot of a computer

    Description automatically generated
  + Now the right shift of signed numbers, they behave differently, to preserve the negative sign value, instead of pushing the 0s from the left to shift the bits to right, the 1 will be pushed here, we can verify this by multiplying 2^n but the result would show the same value but -ve, also you would notice that this time instead of LSRS, for **SIGNED INTEGERS** shift operation machine code uses ASRS **which arithmetic shift** instruction which can be visible in the following pictures as well as you can verify division by 2^3 and -ve sign and Signed Integers use different Opcode and the shifting of 1 instead of 0 to the MSB’s bits
    - Integer Value of SIGNED\_POSTIVE\_NUMBER\_RIGHT\_SHIFT
    - Integer value of SIGNED\_NEGATIVE\_NUMBER\_RIGHT\_SHIFTA screenshot of a computer

      Description automatically generated
    - Binary value of SIGNED\_NEGATIVE\_NUMBER\_RIGHT\_SHIFT
* Define the macros of LED colors as preprocessors which is beneficial for higher order bits and saved a lot of time, instead of writing down the hexadecimal value you can just use shift operations as we know **>>n** corresponding to 2^n divides and multiply in case of **<<n.**
* **Since you want to increasing offset you use << LEFT SHIFT OPERATION**
* Means making data led to <<1==0x02 means that you obtain the Red Color
* Means making data led to <<2==0x04 means that you obtain the Blue Color
* Means making data led to <<1==0x08 means that you obtain the Green Color
* Use **bit set idioms** to turn on/off the specific bits in the register but this can only be performed on the register who has **READ/WRITE OPERATIONS**
* Making optimization high also created a short and simplified code

**Lesson#07: Arrays and Pointer Arithmetic**

* In previous lesson we know that we can write to specific bits using bitwise operator, this is done in a way that, the program first read the GPIOF register with LDR instruction, next it uses the bitwise OR/AND/EOR, then finally it writes the modified value with the help of STR instruction on the register
* Now we introducing a new concept: We want to have a unique address of each bit in a single register that we want to use/execute.
* Every hardware use its unique way to do this, we learn how stellaris Boards allows you to do this
* This technique will be useful when we are dealing with interrupt which changes the state of hardware abruptly
* When an interrupt occurs a special hardware changes the value of Program Counter so that a processor suddenly start executing a different piece of code that code called a Interrupt Service Routine ISR and that code is quite short
* When ISR ends the processor start back the execution from the same spot
* The interesting case when the ISR changes some GPIO pins, if the interrupt comes between the READ MODIFIED CYCLE WHEN UPDATING THE A SPECIFC BIT AS EXPLAINED ON ABOVE POINT, before the **changed values write back to the GPIO by ISR will get lost** this is because the main code will use the previous values of GPIO Register before the interrupt happen. This is the **inherent problem with READ-MODIFIED-WRITE Cycle.**
* The designers of the board avoid this cycle and replace it with the specific bit operation

A screenshot of a computer

Description automatically generated

* Each bit (P0,P1,…..P7) are connected to dedicated data line and dedicated address line
* To change the any bit, that bit’s address line must be 1
* For example to change 3 bits P1,P2,P3 you need to write 1 to its address lines

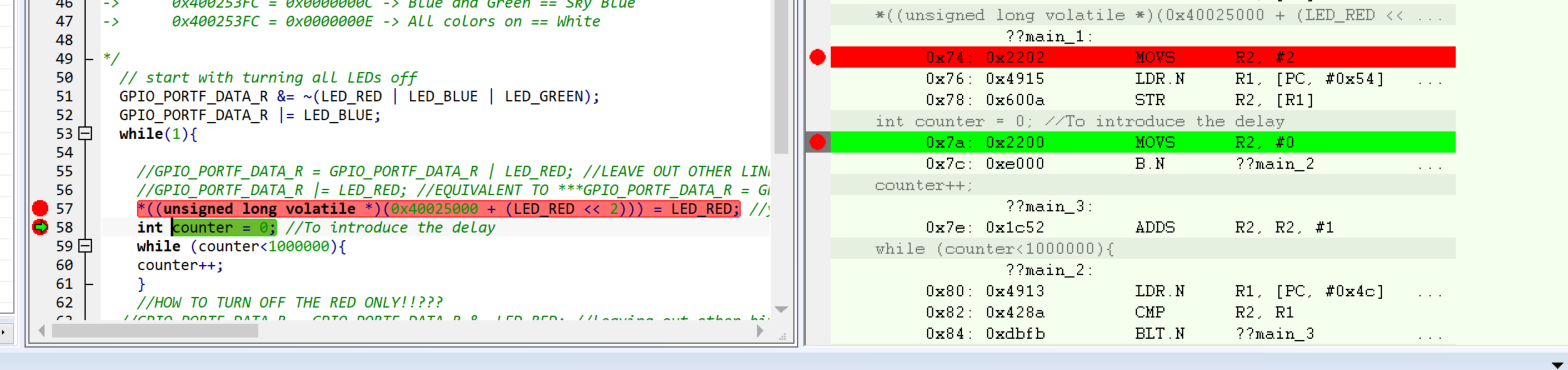
A diagram of a circuit board

Description automatically generated

* What data to write is done on data wires of the specific bits. P1=1(RED LED ON), P2=0(BLUE LED OFF), P3=1(GREEN LED ON). All this happens in a single write operation.
* This hardware design requires many registers with a unique addresses
* Each bit has its own unique address
* In previous classes to write any data to GPIOF registers we have been using 0x400253FC, using previous approach enabled all the pins on the register.
* How we access and write the values on specific bit of a register
* One way is to hardcode the addres directly, the approach we used to in preprocessors.

**\*((unsigned long volatile \*)(0x40025000 + (LED\_RED << 2))) = LED\_RED; //you have just change the specific bit without using a bitwise operator**

* The following is the machine code of turning on RED LED where it complies with **READ-MODFIED-CYCLE** which can be potential reason for creating bugs when dealing with Interrupts, this cycle can be identified LDR-ORR-STR are execute in a sequence, the **LDR-ORR-STR** == **READ-MODFIED-CYCLE** A screenshot of a computer

  Description automatically generated
* Now I show the machine code without bit-wise operator, and on that you can clearly see how machine code get simplified and robust to bugs when dealing with Interrupts, here you cant find **READ-MODFIED-CYCLE == LDR-ORR-STR**
* We need to improve this code by using an array! What is Array? A group of variables of the same type occupying consecutive memory locations such as 256 identical data register
* How to define the array

**int array[number\_of\_elements] = {element#01, element#02, …..}**

**int array[5] = {2, 1, 6,5,1}**

* How to access the array

**int c=array[1]+2; //accessing 2nd element of an array**

* C language treats arrays as pointers, as pointer also points to the location of a variable so does an arrays, that why we can dereference the array as well in the same exact way

**int c=\*(counter+1) +2; //accessing 2nd element of an array**

**int c=\*(counter+3) +2; //accessing 4th element of an array**

* The pointers and arrays can go hand-in-hand so we can apply this on our preprocessors which are predefined pointers
* So the following 3 lines of code produce the exact same machine code:

**\*((unsigned long volatile \*)(0x40025000 + (LED\_RED << 2))) = LED\_RED; //ADDRESS ARTH**

**GPIO\_PORTF\_DATA\_R[RED\_LED] = RED\_LED; //ARRAY ARTH**

**\*(GPIO\_PORTF\_DATA\_R + RED\_LED) = RED\_LED; //POINTER ARTH**

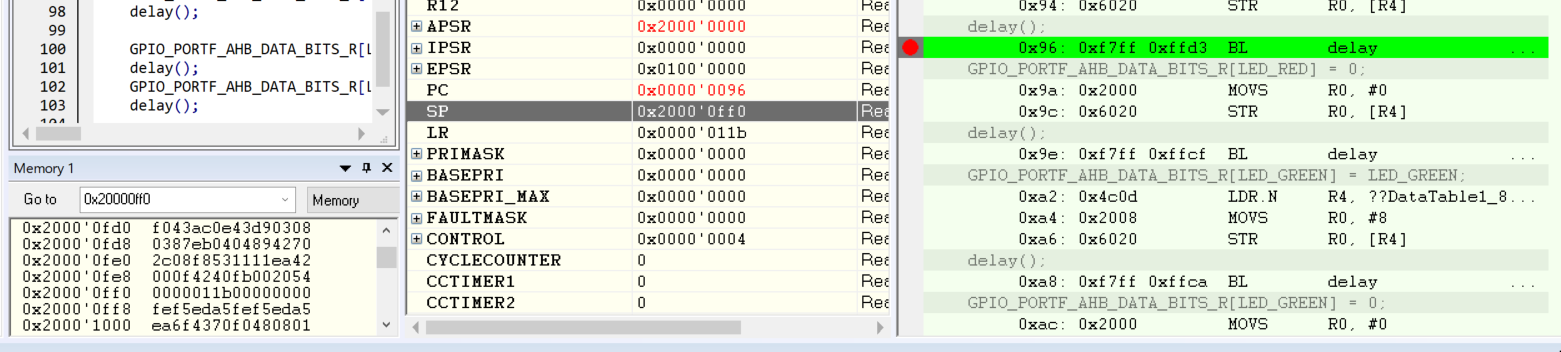
* The very important difference between **pointer artihmatic and address arithmetic** is that
* In pointer artihmatic you don’t need to scale the offset ( <<2 ) by the size of the element because this is done automatically for you. Why????
* Lets Array indexing technique to clear the red led!!

**GPIO\_PORTF\_DATA\_R[RED\_LED] =0; //writes 0 to led red bit position**

* So this is the fast and interrupt-safe program for manipulating the GPIO pins
* My development board is not supporting the array operation on GPIO\_PORTF\_DATA\_R, compiler says that **expression must have pointer-to-object type but it has type "unsigned int",** this is due to multiple reasons one important one might be is different board.
* There is one more thing, just like other ARM board stellaris board contains the AHB and ABP, one exception is that the AHB normally use in memory component section in ARM processors here the GPIO utilizes both AHB and ABP and obviously AHB is much faster than the ABP, ABP uses because of backward compatibility.
* For that you need to go to the datasheet and find how to convert from default ABD to AHB, find : GPIO High-Performance Bus Control (**GPIOHBCTL**) which is on page#258.
* To find in preprocessor file just type **GPIOHBCTL** in lm4f120h5qr.h
* You copy the register name and set bit 5 in it since GPIOF is also present at bit 5 hereA table with numbers and a yellow box

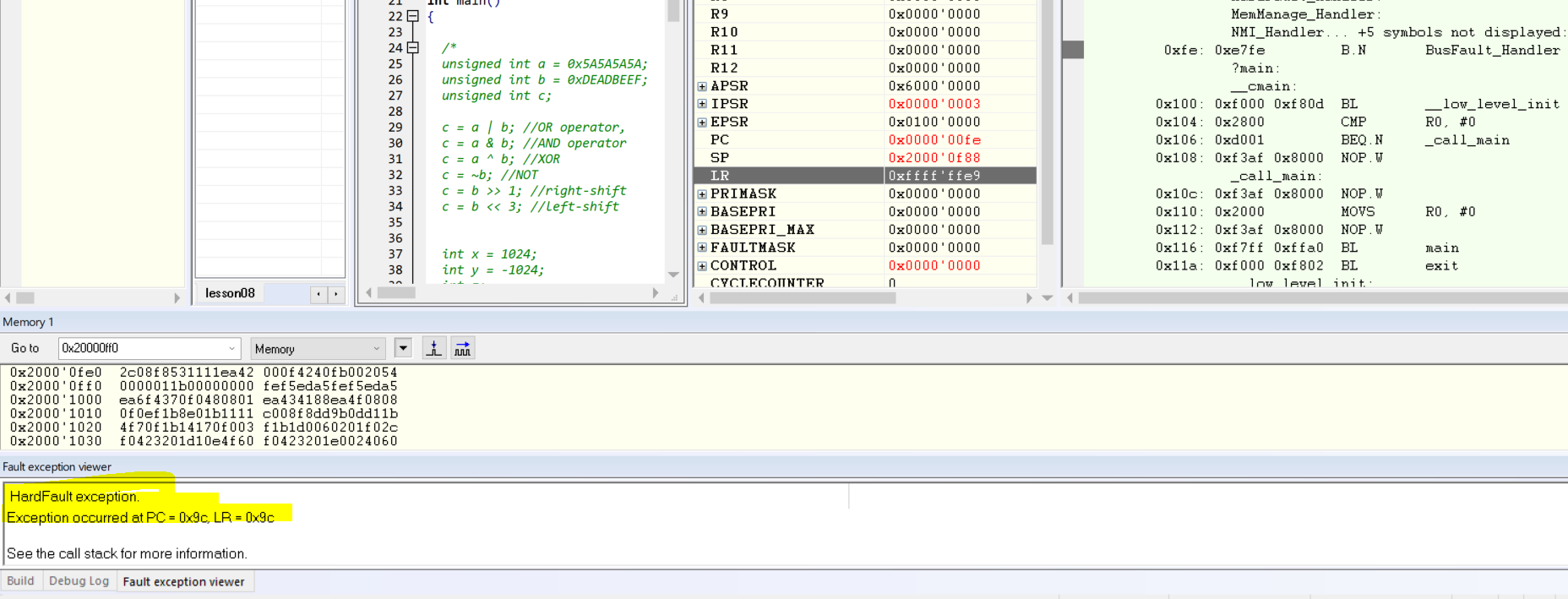
  Description automatically generated
* For other register operation you can find registers with \_AHB suffixes, you just need to insert this suffix into the and after **GPIOHBCTL**

**Lesson #08: Functions in C and the call stack**

* We will discus today how a C stack enables calling function that call other function
* Call stack is the key to understand function, interrupts, context, switch, and RTOS
* A function in C known as procedure, subroutine, or sub-program is a reusable code that can be executed from different point in a program
* The **return type, the name, the list of arguments** are the identity of the function
* Define then initialize the function
* One important thing, whenever you run code, set optimization to low, because if controller is in high optimization it will inline the function means that it will eliminate the function
* When you work with function always checks the **require prototypes** inside C/C++ Compiler
* Function in c has 3 steps:
  + Defining the prototype outside int main()
  + Defining the function outside int main()
  + Calling the function inside int main()
* We only create function for delay loop and testing it
* How processor actually calls the delay function?
* It is seen that BL instruction which is ARM control flow register saves the address into R14 register which is also called the link register LR
* LR Register remembers the place in the code to return the execution after the function complete its execution
* BL instruction is 4 bytes long and other instruction are 2 bytes
* Instruction set of ARM CORTEX M processor which is called THUMB-2 mostly consist of 2 byte instruction and sometimes 4 bytes
* Function:
  + Adjust SP (Stack Pointer) register, SP is the hardware implementation of C, called stack mechanism.
  + A C stack is an array of RAM that can grow and shrink from 1 end only, the end is called the top of stack and **SP register contain this top address**
  + In the ARM stack grows from higher to lower addresses and shrinks from lower to higher, in other processor this process is opposite
  + You can imagine stack as the stack of dishes you can access only the top dish
  + Subtracting 4 from the stack means grows the stack by 4 and create space for local variable counter at the top of the stack “jagah banao plate rakhne ki”
  + Add another break point at the end of stack and analyze the machine code
  + When function is ending, The first thing compiler do exact opposite on stack that were performed at the entry of the function, in our case stack is shrunk by 4 bytes so exact opposite will happen.
  + The time when function being called check the BL instruction also the SP register which holds the 0x20000FF0 which is the top of stack****
  + Lets see after function execution: highlighted text shows that this is the no of times the loop runs**A screenshot of a computer

    Description automatically generated**
  + The actual return from return function of the function is the BX command, which stands for Branch and Exchange, this instruction set the Program Counter to the value specified by the location of the BX instruction A screenshot of a computer

    Description automatically generated
  + ARM normally use 2 types of instruction set, ARM and THUMB2, ARM instruction set is 32 bit, meanwhile the THUMB2 is 16 bit which is more compact which allows for more efficient use of memory and often leads to faster execution. **BX instruction** determines which instruction set to use. When executing subroutine or function or procedure, the processor stores the return address in the Link Register(LR).
  + Upon completion of subroutine, the processor need to jumps back the **instruction** that called the subroutine, this is where BX instruction comes into play.
  + The BX instruction transfers control back to the address stored in the LR but also checks the least significant bit (LSB) of the LR
    - I**f the LSB is 1:** The processor switches to the Thumb instruction set and continues execution.
    - **If the LSB is 0:** The processor switches to the ARM instruction set and continues execution.
  + In Cortex-M, since there is no ARM instruction set to switch to, the LSB in the LR (which typically indicates whether to use ARM or Thumb) is ignored in terms of switching instruction sets. The processor remains in the Thumb-2 mode, making the behavior of the BX instruction in this context a historical artifact rather than a functional feature.
  + Even tough to check if the LSB changes to 0 how CORTEX-M behaves, we playfully enters changes to 0 to see what happensA screenshot of a computer

    Description automatically generated
  + We put c on place of b, exception occurred
  + When you added call of another function into the int main() function, the int main() function stops acting as leaf function and had to something special to preserve its own return address.
  + When you add a call to another function inside int main() (or any other function), int main() is no longer a leaf function. This means it has to deal with the potential overwriting of the return address stored in the LR
  + prevent losing the return address, int main() must save the current value of the LR before the BL instruction overwrites it. The best place to save this value is on the stack because the stack is designed to store temporary data such as return addresses and local variables
  + As we know that the return address is stored in LR registered but this time this LR is already filled with the help of BL instruction, so any function that executes BL must somehow save the previous value of the LR so it can return to the right place.
  + The question is, where is the best place to save the LR, the PUSH Opcode saves the specified list of registers on the stack and automatically grows the stack.
* Stack is utilized for 2 purposes;
  + Holds local variables of the function that have been called
  + 2nd it stores the return addresses
* Function args allow you to specidify the initial values of the local variables
* For example you want your function to take different time delays with the help of variables
* In debugging we also find out that the parameters are passed to the R0 register
* Address of counter variable is at the top of the stack because its address is same as the SP registerA screenshot of a computer

  Description automatically generated

**Lesson #09: Modules, Recursion, ARM Application Procedure Call Standard (AAPCS)**

* With the functions, you can create functions in other files
* Paste delay code into a newly created file with the name delay.c into the directory where main.c is present
* Even when you create a file in cwd you need to add into the project by RC on the project and click on add files then add the file, this is the standard process you need to follow when making dealing with IAR toolset
* In the delay.c file you need to declare and define the function, which also make us say that “the prototype/declerartion and definition must exist on the same file”.
* But above point violating the DRY principle for this reason will place the prototype in a different file called delay.h in the current working directly, that delay.h will need to be include in all other files where its being used by the #include “delay.h”, this command will be repeated where function is define which is delay.c and also where fcuntion being called which is main.c
* In this way we would avoid the repitation of prototype code
* Many times other headerfiles include other headerfiles which could lead to a situation where **header file can be included more than once**.
* To see yourself how the protection is being made you need to look at the premade preprocessor header file and on the top of the file you can see

**#ifndef \_\_LM4F120H5QR\_H\_\_**

**#define \_\_LM4F120H5QR\_H\_\_**

…

…

…

…

**#endif**

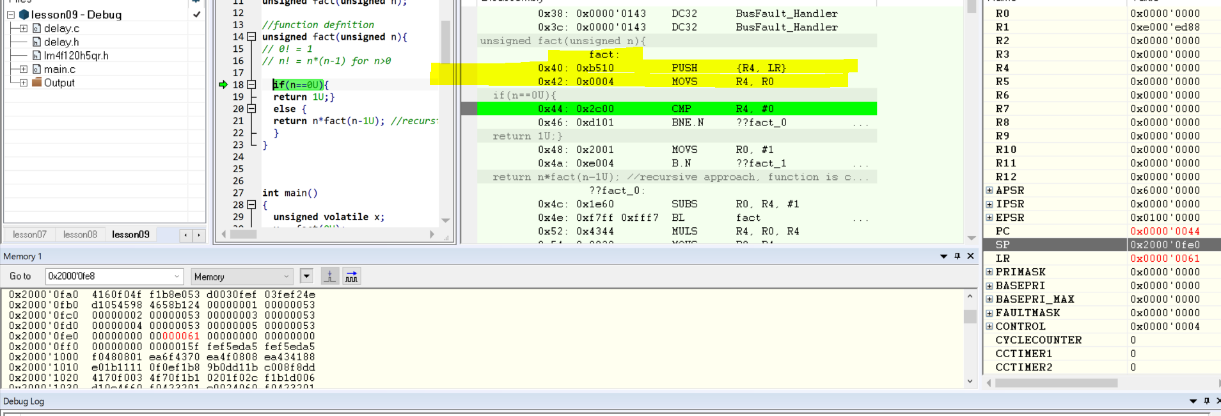
* This protects this file to be included with other header files, the above line checks whether the file being defined as macro or not, if not then the bofy of the file will be skipped, above code in the next line we have defined it
* We will apply this technique on delay.h header file
* This approach of placing functionalities into different files orginate structures which helps the compiler in better calculation. -MODULAR APPROACH
* How it helped? Because then compiler would only execute the changed files.
* In C we have learned, to build programs on separately combined files.
* Now we explored more abilities of functions, which is the ability of a function to return
* We are creating a function that calculates the n-factorial
* The way we planned, first create a prtotype and calling up mechanism
* In calling up mechanism we will define a vriable the will be passed as an argument, in our case it would be volatile to prevent the compiler for optimizing it.
* Now you need to define the range of values being passed to the function as argument, such in our case we want it to be 0-n, and also we save this calling into a function

**x = fact(0U);**

**expression\_var = 2U + 3U \* fact(5U); //calling function into the expression**

**(void)fact(5U)//you don’t care about return value, so then cast with void**

* Compiler doesn’t know in which file the function you called is reside, you need to investigate by yourself like a detective
* There are two ways in which you can write mathematical algorithms in codes
  + Recursive-for this exercise we will use recursive definition
  + Iterative
* Lets debug, the first thing we wanted to observe is to above the stack, whose address is stored in SP, so copy the stack pointer’s value into memory view and see above the stack
* Stepping into the code:
  + The call of your function consist of 2 instructions, first the argument value moves into R0 value and then the function is called with BRANCH and Link **BL**A computer screen with text

    Description automatically generated
  + **Function Definition:**
    - Push the registers **R4** and **LR** to the stack
    - Since we know that BL when called subroutine then it stores the return address into LR, so that the program can return to a right place
    - Meanwhile BX is used to return from a subroutine, it branches to the address stored in LR
    - Here you should also know why LR should be saved, because factorial is not a leaf function, so it preserved LR and its useful approach because when it calls itself
    - Why saving R4 register? As we know the first arguments is passed into R0 register, but inside factorial, R0 is reused to **return** the and also as the **argument** for **nested factorial call**. So the compiler moves R0 to R4
    - At the return statement, function moves the return value into the R0 register which proves our previous point
    - The last machine code instruction of the function is very unique! As we know any stack operation need to be reversed at the end of function, before the function returns, so the function pops the registers which it pushed initially
    - But the content of original LR popped directly into the PC registerwhoch causes the return to be exact the position
    - Another thing to notice that the value on the stack that was popped into PC is actually an odd number 0x49, but the return address the PC has become is an even number which is 0x48, why the LSB is specially handled in the BX instruction, this instruction is used to return from a leaf function?
    - Here you can also see **POP to PC instruction is also an special case and behaves like the BX instruction**
    - Here you can also see the function returns the value into R0, which is then stored in the top of the stack where the x variable lives A screenshot of a computer

      Description automatically generated
    - Now in the next call where inside a function, the function calls itself, the **next call to factorial** happens before the previous **call returns** and pops the **register** from the **stack,** so it **nests** on the top of the **stack** allocated by the first call.
    - After the recursive call, the value that returned in R0 is MUL by the original value by argument n that stored in R4, the stored again stored in R0 to be returned from a function A screenshot of a computer

      Description automatically generated
    - In the main function you can see how expression is evaluated, where factorial value is taken from R0.
    - The 3rd call which pass arg 5U shows 6 level of recursive calls, as we go down to those levels we can observe that the stack is building up, now this time recursive calling stops, now all these nested calls will return one at a time and we can clearly see how stack being unwind
    - Finally we can see the result it produced in R0 is 0x078 ==120 decimal == 5! A screenshot of a computer

      Description automatically generated
* The last topic of the video is, function calling convention, there must be some agreement between the caller of the function and the function being called, example: both side must have an understanding that the return address will be provided to the function into the LR register, also both parties must agree that the first argument will be passed in R0, and the return value must also been return in R0 as well.
* Like these agreements there also many other agreements which forms whole formal contract called **ARM APPLICATION PROCEEDURE CALL STANDARD (AAPCS)**
* You can find this document online by searching “AAPCS”
* This is very useful when you learn about handling the interrupts on the ARM processors
* The salient features of **AAPCS:**
  + R0-R3 nad R12 is used for passing the argument and the return values
  + On the other hand, function must preserve 8 REGISTERS R4-R11
    - That doesn’t mean function cannot use those 8 register, if it does use those register, if it does, the function code must save the, on the stack and restore before the returning. Example can be recent R4 register which saved on the stack
    - This behaviour allows the function of caller to use the preserved registers for the values that must survive a function call
    - Remember that the value stored in R4 exactly survived the recursive call to factorial because it was needed in the final multiplication
* The recursive implementation here is just for a demo of a deep call sequence so you can watch and understand how the stack grows and shrink, in-fact any such deep call sequence must be avoided because it eats away RAM for the Stack
* A much better implementation must be ITERATIVE VERSION OR LOOKUP TABLE
* This concludes this lecture about “modules, function that return values, recursive function and AAPCS”
* **Stack Pointer (SP):** The stack pointer holds the address of the top of the stack, which is a region in memory where the program stores data temporarily (like function return addresses, local variables, etc.).
* When a function is called, **the first argument** is usually moved into the **R0 register**. This is a convention in ARM architecture.
* function call is made using the **Branch and Link (BL)** instruction. The BL instruction does two things:
  + It branches to the address of the function.
  + It saves the return address (where the program should go back after the function finishes) into the **Link Register (LR)**.
* function starts executing, it first saves some important registers onto the stack. This includes:
  + **R4 register:** Often used to store variables that need to persist across function calls.
  + **LR register:** Since LR holds the return address, it’s saved onto the stack to prevent it from being overwritten in case the function calls another function (especially in recursive functions).
* The function does its work. For example, in a factorial function:
  + The argument in R0 might be copied to R4 because R0 will be reused.
  + The function might call itself (recursion), requiring the current state to be saved so it can resume correctly after the recursive call.
* After the function completes its work, it needs to return to the point where it was called. To do this:
  + The return value is placed back into the R0 register.
  + The function must restore the saved state by **popping** the saved registers (R4, LR) back from the stack.
* The function uses the **BX instruction** to return, which branches to the address stored in LR. The LR register might store an odd address (like 0x49), but due to the ARM architecture’s handling of the LSB (Least Significant Bit), the program counter (PC) will adjust it to the correct address.
* **Recursive Function Example:**
  + **Initial Call:** The first call to the function pushes the return address and R4 onto the stack.
  + **Recursive Call:** Each recursive call adds another layer to the stack, saving more return addresses and variables.
  + **Return Process:** As the function completes each recursive call, the stack unwinds, meaning each saved state is popped off the stack and the return address is restored, allowing the function to return to the correct place.
  + **Result:** Finally, the result of the factorial (like 120 for 5!) is stored in R0 and returned to the main program.
* **Iterative Methods:** These use loops instead of recursion, saving stack space.
* **Lookup Tables:** Precompute values (like factorials) and store them in a table for quick lookup, avoiding repeated calculations.

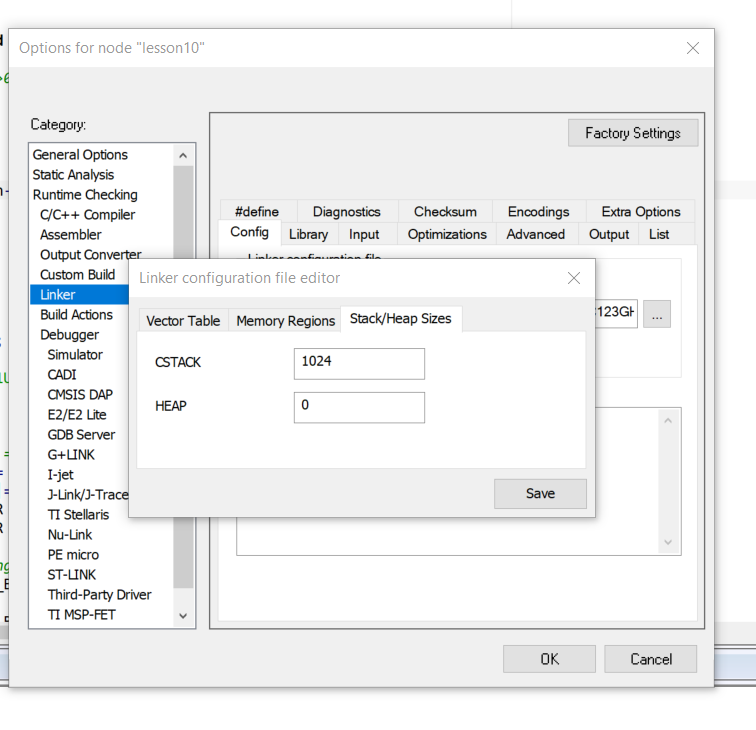
**Lesson# 10: Stack Overflow and Other Pitfalls of Functions**

* You will learn today about pointer arguments, returning pointer values
* To installl New version of IAR software uninstall the previous version
* We will hack the function to stress the stack to the breaking point
* With the help of foo[n] array we are growing our stack exponentially, to see the affect:
  + Before stack grow:**A screenshot of a computer

    Description automatically generated**
  + After stack grow:**A screenshot of a computer program

    Description automatically generated**
* One good question might be what and where the values are coming from onto the foo variable
* The stuff coming from a previous uses of RAM, this particular data looks like a flash memory image, most likely left by flash loader that programmed your code into the flash ROM, but the thing to focus is the content of this stack is garbage for you
* Indicating that any automatic variable doesn’t have any specific value but garbage values
* Call stack is like a stack of dishes that are all dirty so before using the these stack of dishes we need to wash them\
* As of goal this exercise was to break the stack, till now we are able to explode it, but to break it further we increase the size of the foo[n] increasing n
* Open call stack window which tells you are in which function currently
* As we know stack grows upwards, so stack pointer finally reaches to the top of the RAM memory starting address which 0x20000000**A screenshot of a computer

  Description automatically generated**
* Lets run more from this point and see what happens
* Finally the stack pointer goes below the valid RAM, means more lower address than 0x20000000, if we look at instruction code the program hangs around **Bus\_fault\_handler,** this require some explanation
* Bus fault handler is not your code, instead it is the so called **Exception Handler** provided in the IAR startup code which is linked with your main program by the **linker**
* The bus fault exception is a hardware mechanism implemented in the CPU to handle the situation when the CPU forced to access **non existence memory.** IAR software uses bus fault exception handler as well as other exceptions
* But instead of program goes to Exception Handler you can provide something else to your code, for example resetting ‘reset your PC’
* We will learn how to define our own exception handler in our own startup code
* **Today we have experienced our very own stack overflow**
* Now you have to develop a habit to checking your stack pointer when you find your program hanging inside hardware exception
* Stack overflow can fail in some other ways, such as only corrupt some data but not run out of memory which can be much harder to detect and diagnose
* How can we change the stack size ????
  + Step 01: open option and go to linker, tick override default, click on edit**A screenshot of a computer

    Description automatically generated**
  + Step 02: click on stack/heap sizes option, turns out that the default stack size is of 2 kb mentioned on CSTAK, since we can use decimal so we specified in the CSTACK and type 1024, meaning turning the stack size to 1kb****
  + Step03: The heap is the region of a RAM for a dynamic memory allocation with the standard function, malloc and free, this is very useful in general purpose computing but in real time embedded programming, the heap causes more harm than any good and **you should not use it.** In this case set the heap size to 0 and click save button****
  + Step 04: Now you need to choose the location of your IAR Linker script project.icf, save this file in your cwd of the project
* Lets currput stack in another way by passing 7U value in function call and set foo[6]
* Place the break point where you call the function specially when you pass 7U value in function call and debug
* The changes:
  + the difference here is the the subtraction command **SUB** which subtracts SP by 0x18 which only make some room on the stack but no cycles are wasted to clean the space, that is why the foo[] array contains the garbage ????
  + The **ADD** instruction put the address of the foo[] array which happens to be the current top of the stack into R1
  + **STR** instruction writes the value n from R0 to the index n which is also R0
  + Now watch carefully the affect of STR instruction because the crime happens right there, the last index of foo is 5, so the index 7 goes **2 locations** beyond the end of **foo,** this location happen to save LR register which is now corrupted and the function will not be able to return correctly
  + What was the crime? You indexed an array out of bounds and corrupted the stack, the C language allows because C doesn’t check array indexes and trusts that you **know what you are doing.**
  + The real story infolds due the crime is the 1000’s of clock cycles before the system finally fails, obviously you wont do single step for each cycle instead put break points strategically:
    - The first strategic location is the return from foo, why? Because yyou know that the problem is with the return address, when you hit this breakpoint, all the recursive calls to factorial are nested on the stack, this is the maximum use of the stack, you can verify that this is not the stack overflow problem, now when you continue to run from here you gradually unwind or reduced the stack, and each nested call return
    - We keep unwinding it and reach to the last isntrcution of the stack POP R4 performs final return, note that the return address that is about to be stored into the PC is 0x07, this is the corrupted value which I have carefully planned to be **odd** because if it is even the pop instruction would fail right here and CPU will go to the **exception handler** which will end the story, if you wonder why every return address must be odd on CORTEX-M you should visit lesson 8 where we tell that CORTEX-M doesn’t have more than 1 instruction Set
    - By careful crime creation, the POP isntrcution succeeds and the Program counter forced to 6
    - But what happens next is surprising! Disassembly view is misleading, the problem is that these low memory locations are used for the so-called **exception** and **interrupt vector table** which means a bunch of 32 bit memory addresses but surprisingly CPU executes this data as the legit 16 bit instruction, whereas it must take 2 disassembly steps per each 32 bit data value, by some coincidence the main function immediately follow ths vector table in the FLASH ROM so now the CPU start to execute the ral instructions
    - The first one is to PUSH to the satck but the stack already has the previously pushed registers from main, because remember the main has never really returned, when you continue from here you end up hitting the break point which we placed on ADD isntrcution, remove it and continue, now each time you hit the continue button, you execute the whole cycle of recursive calls, corrupting the stack and re-entering the main() through backdoor of executing the vector table, remember the stack slowly growing, because the main() does not actually return so it does not POP its stack from the stack frame from the stack, finally remove the last break-point and let the program run free, from here you should know that you eventually overflow it, and the CPU enters the BUS Fault exception handler
    - I hope you understand here is that the runaway program corrupting its state sometimes for thousands of CPU cycles, this tends to be very hard to reproduce and debug**A screenshot of a computer

      Description automatically generated**
* Now we will shifts the gears bit and talk about function arguments, including pointer arguments and returing pointer values from the functions
* Lets modify the delay function iter into, now it checks if its greater than 0 than loop keeps running keeps decrementing the iter value, meaning loop runs backwards, this shows that the function arguemments are just like local variables which you can modify the only difference here, the arguments are initialize by the caller, and local variables must be initialized inside the function
* Make the delay loop volatile to save from compiler optimization, you must make the same changes to prototype of function and definition of function, also at the calling side instead of passing the constact pass the variable which has the required constant into the cuntion of int volatile part
* Set the break point first call of the delay and immediately after the call, at the first brake point verify the variable x must have value 1 million, but when program executes and the function return, the value 1 million is still remain in x, the function has already decremented to 0 in the loop
* The conclusion of this experiment is show that the c passes function arguments by value meaning that only argument’s value is copied to the internal internal variable to initialize it, but internally function uses **the copy rather than the original argument** this means that the function will **never change the original argument,** but sometimes you need to change the original argument, the classical example is the swap operation which exchange the values of its arguments of x and y.
* The first attempt might be to write swap function here, for that we pass the paramters as location of x, y to the function and use pointer variables there to access those locations and using the dereferencing technique to swap the values of the location, that is how much the pointer is power is, you now you have successfully swapped the x and y values
* Now we will talk about the persistent warning that we are having the return statement is unreachable specially after the return statement.
* This is appearing because compiler sees that there is a while(1) loop before the return, however the return type from main must be int, because this is required by the C standards, at the same time standard also requires, every function with **non-void** return type always explicitly return this type, so it is impossible to satisfy the standard and avoid the IAR warning at the same time.
* So far we have opted for standard compliance and portability because other compilers such GNU GCC will report if the return statement is missing
* So we don’t get bothered by return 0 statement, and we can remove it
* Now you want to make the swap function to remember the x, y arguments to remember the x,y pair of arguments in the original order and return it as an array
* Change the code by introducing an array and swap them and produce them in reverse order.
* Ignore the warning with return statement but lets ignore it and see why it is wrong, lets use new swap function in your code in your while loop, the general idea is to keep swapping the delay times for LED ON and OFF time to make the blinking pattern mor interesting
* Set the break points at the return and check the stack and verify all the variables that are present into the stack, when you step out of swap, the stack contains only x and y, and tmp array is still recognizeable in the RAW memory view, now it is above the stack pointer thus it is no longer shown in the CSTACK view.
* Now set the break point as 2nd call of delay and run, when u stop there notice that the argument in R0 is 0, instead of expected 500,000.
* A quick look into the raw memory shows why! In the mean time, the first call to delay used the stack and destroyed the previous value. So now you see why returning a pointer to a local variable is always a bad idea, because such pointers always fall above the stack after the function returns, the more technical term is all local variables go out of scope when the function returns, so it longer even exists, and cannot be accessed.
* The remedy: instead of using local variables on the stack, use local variables that not on the stack, in C the ***static*** keyword used in front of local variables tells the compiler that to allocate the variable outside the stack, so it outlives any call to the function and therefore can be accessed even after the function returns, after this change the calculation produce no warnings
* Now tmp is longer on the stack but on the regular memory right at the beginning of the stack
* Now the second call to delay, receives the correct argument of 500,000