**Lesson# 16: Interrupts Part-1: What are interrupts, and how they work**

|  |  |  |
| --- | --- | --- |
| **Time** | **Capture** | **Create** |
| **4.06** | Analogy and need of interrupt | The code of delay() we are using is putting our controller in a busy-wait polling state, which for a longer period of time blocks other operations and doing 1 single operation of checking whether the iter<counter  It is like you keep looking at clock throughout the night to wake up at 6AM in morning  Just like this analogy, interrupts in hardware, interrupts the hardware though interfaces option of hardware and in response to it hardware respond  To hear the alarm clock you need a hearing system, just like that hardware needed **for the interrupt to happen to the hardware, interrupt is like an IC** |
| **5.29** | What is interrupt consist of and where it is? | USING MICROCONTROLLER DATASHEET: An alarm peripheral sits next to CPU block and connected via interrupt is SysTick, this peripheral operated on same CPU clock  SysTick have 3 registers STCTRL, STRELOAD and STCURRENT  -STCURRENT is a 24 bit down counter with getting down by 1 on every CPU clock cycle. When reached 0 it generate an Alarm via interrupt to the CPU, controlled by SysTick VAL, using this register you can set the time alarm should be rung  -STRELOAD: on time domain, when alarm is generated the STRELOUD resets and reload the interrupt counter |
| **6.49** | How interrupts work and it works in which clock cycles | How CPU listens to the interrupt coming from interrupt line?  CPU samples when the Asynchronous interrupt lines when rising edges of clock arrives  The hardware that samples enter into Preemption stage when interrupt line is high it samples it, when the interrupt line is high the CPU enters into Interrupt entry, this can happen with interrupt entry instruction which makes the CPU enter into interrupt which consist of 12 CPU clock cycles meanwhile the instructions like MOV, ADD are consist only on 1 clock cycle  Interrupt is independently works |
| **8.07** | How would you relate the calling of function to interrupt, dealing with brain-dead delay function | Between the pauses of interrupt the braindead delay() function communicates to it  Make changes in code by introducing XOR operator which toggles the LED when its on or OFF |
| **8.13** | How XOR table works for the code we have written | When the previous state turns to 0 and no pulse arrive remain 0, turn 0 when pulse arrive become 1, turns 1 if pulse is not arrive remain 1, if turns to 1 and pulse is arrive, turns to 1 |
| **9.07** | How code works with the changes you have made | LED is blinking after 1 second |
| **9.50** | How to find the interrupt registers from datasheet to the library which uses to your code | SysTick registers are mapped into **core\_cm4.h** file which imported into tm4c\_cmsis header file  One more thing is that, the file uses different register names then it mentions in the datasheet, but since we are using 3 registers then it is really easy to find out |
| **10.15** | How SysTick CTRL registers employed | Use this register to set its 2,1,0 bits to enable **Clock-Source, Interrupt-Enable and Counter-Enable** respectively by using bitwise << operator |
| **10.32** | How SysTick VAL registers employed | Since it is a clear-write register then it doesn’t matter what your write to it and we write 0 which will be clear |
| **11.29** | How to use SysTick LOAD depending on clock and how it being made for the interval | On this register you will determine the intervals, to set the interval of LED blink to half a second you need to know the value clock which drives the Interrupt in HZ, and we have 16MHZ clock frequency  This value is introduced in the main.c code as the constant macro  Inside the code we divide the clock size by 2 and -1  **Why -1, since systick counts from 0, if you didn’t subtract 1 you would count one clock too many????** |
| **11.51** | How to verify that your incrementing counter remains under 3 bytes which is of SysTick | Since the counter is based on 24 bits==3bytes, you need to check the expression you are creating, its result must and should remain inside 24 bits by turning its value to hex, if 6 digits of hex are obtained then you are inside 3 bytes, as **nibble** ==4bits is dedicated to 1 digit, then 4\*6 = 24 bits |
| **12.15** | How your code layout is working accessing the functions | When your systick imterrupt is enable, this will interrupt the CPU based on the result of the expression you set in SysTick VAL  RN the interrupt occurs after half a second which would call the systick interrupt handler which we install inside our vector table, that was related to another bsp.c file which has its definition, the definition of SysTick |
| **13.25** | How to deal with changes when you place the board specific codes of LED RED…. Into bsp.c, how you you deal with existing delay.h and make a new file that related to bsp.c, obviously we need bsp.h file | Place the LED toggling code inside the bsp.c’s SysTick\_Handler() definition  Throws an error because it doesn’t have any board specific definition we have set including clock frequency, this can be done via placing those board specific definition for LEDs from main.c file to delay.h, change the name of delay.h->bsp.h  Also change the security of the file  Also include bsp.h into main.c and remove delay.h |
| **13.57** | Why did we remove delay() function now and left while loop running | We need the CPU to remain active to service the interrupts |
| **14.46** | How SysTick is connected to the Interrupt line of CPU, what made the interrupt line enable then, means what function being used to enable the interrupt? | PRIMASK bit must be cleared in order the interrupt to reach to the CPU, this is placed between CPU and SysTick as the gate keeper  You need to call IAR intrinsic function called \_\_enable\_interrupt() == \_\_iar\_builtin\_enable\_interrupt(); this function will clear the primask bit |
| **15.24** | How the final code behaves on debugger | When you run you see it is running inside empty while loop  When you set breakpoint at bsp.c you will see its running around the XOR command to toggle the red led  The instruction set it generates 4 or 5 instruction opcodes |

**Stages of Execution:**

**Stage #01**

**A screenshot of a computer

Description automatically generated**

**Stage #02**

**A screenshot of a computer

Description automatically generated**

**Stage #02**

**A screenshot of a computer

Description automatically generated**

**Lesson #17: interrupts Part-2: How most CPUs (e.g. MSP430) handle interrupts?**

|  |  |  |
| --- | --- | --- |
| **Time** | **Capture** | **Create** |
| **1.49** | Use different file than tm4c\_cmsis because the other file named "TM4C123GH6PM.h" is more compatible to CMSIS standards |  |
| **3.36** | Call SysTick handler function as a regular function, before a enabling the interrupt by calling \_\_iar\_builtin\_enable\_interrupt() abd after it | Before it turns out to be regular C function and RED LED BLINKS |
| **7.31** | How ISR made | When we debug the code we found out that the SysTick interrupt after enabling the SysTick handler call through a different process called premption, but we use SysTick handler as a regular function here in CORTEX-M processor which is a unique feature of them, because normally processor require sepciall entry code through a return from interrupt instruction, they cant be regular C function  **When code through MSP430 processor debugger which is not a CORTEX M processor**  \_\_interrupt() keyword suggest the compiler that this is no ordinary C function, but rather an interrupt handler called **ISR( Interrupt Service Routine)**  The keword is \_\_interrupt() and #pragma vector = …. Is specific to board not on IAR software |
| **11.30** | For MSP430 you need to find a way to expire the timer at will by putting fake values in your code using, for which you need to know how this timer works | The fake values we opted inside TA0R are just below the TACCR0 value which when hit cause the alarm, **the next clock cycle will cause these values to match which will trigger the time0 interrupt**  When debug the code and inject the code with **TA0R with 0XF422,** and place the breakpoints after BIS instruction, after that the timer expires and interrupt will be high, the breakpoint is placed on BIC instruction which only run when Interrupt Cycle not interrupt, and the other on XOR which only occure if interrupt is occurred |
| **12.30** | The program returns to BIC instruction from the RETI instruction on Interrupt Execution | Triggering an interrupt you want, exactly at the instruction you is invaluable lesson as you can execute part of code is not regularly occurs |
| **15.40** | How an interrupt knows where to go to? | When we break at the same location again, we observe that the SP pushes the following 4 bytes into the stack. Referring to the datasheet confirms that this occurs when an interrupt is triggered, involving both the PC and SP.  The SR register value is saved, and the SR becomes 0 upon interrupt entry. This action also clears the GIE, which disables further interrupts to the CPU.  Upon an interrupt call, the SR and PC values are saved on the stack. During that, the RETI instruction **reverts** **the old values of SR (which controls other interrupt pins of the PC) and PC (return address) are restored.**  The RETI instruction does the opposite of INTERRUPT by restoring the original SP and PC values before the interrupt occurred. Since the PC had a return value, the program resumes from the BIC instruction where it was previously interrupted. |
| **17.15** | ISR can differ from regular C instruction in 1 more important way, the ISR must save more CPU registers than a regular function | To see this we modify code in bsp.c and create another function called **LED\_toggle** which called by ISR  Nomal Function: This **Timer0\_Function()** function also called also from main.c which defined on bsp.c, this Timer0\_Function() called **LED\_toggle()** function which toggles the led  Set the defined prototypes of **LED\_toggle()** and **Timer0\_Function()** in bsp.h  **Timer0\_Function VS Timer0\_Handler()** |
| **18.15** | What is the difference betwwwen the two function, did you achive to sabe more CPU registers when using ISR ? | The instructions generated fpr ISR function which is **\_\_interrupt void Timer0\_Handler() by pushing R12->R15 register on stack and then popping them as well,** the generated instruction from ISR did use multiple instruction than a simple **Timer0\_Function()**  Thus we can see that ISR function use more regitsers than a normal register |
| **19.00** | Why does they use so many regitsers? | Because the function call within ISR can damage/clobber registers R12 through R15 so they have to be preserved otherwise an **interrupt PREEMEPTION** would have a side effect of **clobbering** registers  Interrupt can prempt asynchronously any 2 instructions  So the compiler cannot tolerate clobbering registers  On the other hand, a regular function call is synchronous because the compiler is doing it via CALL instruction or sometime BR instruction  In any case the compiler prepared that the certain CPU registers will be potentially clobbered at this paryicular point in the code |

**Lesson#18: Interrupts Part-3: How interrupts work on ARM Cortex-M?**

|  |  |  |
| --- | --- | --- |
| **Time** | **Capture** | **Create** |
| **1.00** | MCP430 handles interrupts in a typical way like embedded CPU’s does |  |
| **2.00** | What unique was in MSP430 intructions of the functions | Normal Functions: The return instruction is ret, also the instruction consist of only 1 line, no need to save any registers  Interrupt Functions: The return consist of reti, also the instructions are using multiple instructions, need to save multiple registers, to save registers from clobbering by PREEMPTION |
| **2.52** | How ARM CORTEX-M uses regular functions as interrupt handlers, means create same instruction set as it was created by MSP430 | The method od experimentally using the interrupt by externally placing such value that, when next clock cycle run the interrupt being called  Unfortunately you cannot use this method here because STCTRL is WRITE-CLEAR register, means that writing any value on it clears the register without triggering an interrupt |
| **4.40** | How to get a direct control of timer on ARM CORTEX-M | Inside SysTick, the INTCTRL’s 26th bit set the SysTick interrupt to the pending state  Debug the code   1. Set breakpoint at top of while(1) loop main.c and run the code 2. On disassembly code move the breakpoint to LDR.N instruction 3. Set another breakpoint at SysTick handler function bsp.c 4. On register panel, select System Control Block 5. Expand ICSR section 6. Go to the PENDSET =1 7. Copy the SP register value and paste on memory view 8. By the way: CORTEX-M provides an interrupt pending bit for every interrupt source, so you can trigger every interrupt source, you can trigger every interrupt in the system by this method, most of these pending bits reside inside the Nested Vctored Interrupt Controller (NVIC)   You are stop at MOVS instruction, by writing to PENDSTSET register, you make the interrupt line high on next clock cycle  You also placed two breakpoints for either condition  One is next to LDR.N instruction, this breakpoint would hit if the interrupt would not preempt after the MOVS instruction  The other breakpoint is placed inside interrupt inside SysTick handler which hit only when the interrupt would fire after MOVS instruction, thus preempting the normal program flow at this exact point |
| **5.01** | What is the result of running the code? Does interrupt fired ? | You cannot single step from this point where your program has stopped, because it disables the checking interrupt after each instruction, instead u need to let the program freely run  Result: When I ran, the SysTick Interrupt has fired  At this point you have a verified method to trigger your interrupt at machine instruction of your choosing |
| **5.29** | Why we are setting FPU Point to None? | FPU is a complex peripheral for speeding up floating point computation, but unfortunately it adds the complications to the interrupt processing which you don’t want to deal with this at this time |
|  |  | The result before we run from the break point was  0x2000003F0  After we ran SysTick  0x2000'03d0  Since each word is based on 4 bits == represent single digit of hex  0x2000003F0 - 0x2000'03d0 = 0X20 == 0010 00 = 32 decimal == 32 bits == 4 bytes == 8 word or nibble  So the new value of SP pointer moves up 8 words above now we need to understand why and how |
| **8.22** | Why Simple C can be used for interrupt? | First we verify the stack register from datasheet where first entry is IRQ Top Of Stack and last entry xPSR  The previous 8 words grows like that, utilizing R0, R1, R2, R3, R12, LR,PC, xPSR register, meanwhile preserving  Back in Lesson#09 you learned about **AAPCS - ARM Application Proceedure Call Standard** which stated that which registers must be preserved by a function call  As we can see Interrupt call is complimenting and following AAPCS, **means CORTEX-M interrupt entry compliments the AAPCS** that is why a regular C function can be used as an interrupt handler  One point to remember: A calling convention could be different from compiler to compiler |
| **10.50** | Why the invalid value returns from a return address on LR register? | When you now single step the function return using standard instructions, but the LR register gets the the unusual value which is **0xffff'fff9** which is an ARM convention of returning the address ==indication of returing of address  When this value 0xffff'fff9 loaded into the PC the cortex M hardware treats this as a return from interrupt  When function is returned, all the registers get back the values before of the function call  Standard return from a function works also as a special return from an interrupt because the **LR loaded with a special value upon the interrupt entry**  The MSP430 achived by a special instruction Interrupt Return IRET, ARM CORTEX-M achieves by using special-data, which is the content of LR Register  ARM provides several variants of interrupt returns which are all summarized in the datasheet 2.5.7.2 Exception Return  Each variant is correspond to some mode, for example,   * Handler mode is a distinct processor state when it handles an exception such as an interrupt or a fault * Thread Mode is when it execute regular code such as your while(1) loop * Floating Point states means FPU is activated and that interrupts use the FPU stack frame as opposed to the regular stack frame * MSP-Main Stack Pointer * PSP-Process Stack pointer |
| **11.27** | Why these distinction being made? | The datasheet make these distinction because your ARM CPU has 2 SP, SP\_main and SP\_Process but only 1 of them is visible as SP, depending on the internal state of the CPU, this concept is called the register banking and is also the ARM Special |
|  | What is the purpose of optional aligner word on interrupt stack?? | The purpose of this optional stack entry is to align the Interrupt Stack Entry frame at an address that is multiple of 8 bytes  The reason hardware needs SP to bealigned is to perform highly optimized block transfers of registers to and from the stack  Interrupt Entry and Exit take 12 clock cycles each which is very fast as compares to when you think of push and pop from the stack |
| **13.40** | When the Stack Alignment is necessary? To understand this we interntionally misalign SP register in the debugger | Just repeat the process from 4.40  But this time I intentionally fill the values above the stack to be deadbeef  Finally you misalign the SP by subtracting 4 bytes from the SP  Now when you run, on memory view 8th register in the INTERRUPT STACK HAS BEEN PUSHED ON THE STACK, but one stack entry has been skipped which is the aligner word  Upon interrupt return whole 9 word stack frame is removed, because the SP value had went back to the original misaligned value  In practice this stack misalignment should not happen, because the 8 byte stack alignment is a requirement of the AAPCS and compilers make sure that the stack is always align  Stack Alignemnt topic will be more important when we talk about RTOS |
| **15.27** | What is the impact of FPU Alignment ? | Re-enabled the FPUv4 single precision  When execution enters to the function, the SP drops to 264 Bytes word which is 4 times bigger than the regular exception  LR return 0xffff'ffe9 instead of returning 0xffff'fff9  The 0xffff'ffe9 correspond to mode MSP  **The real difference between using and not using FPU is that you need to use the stack size bigger if you would use FPU** |