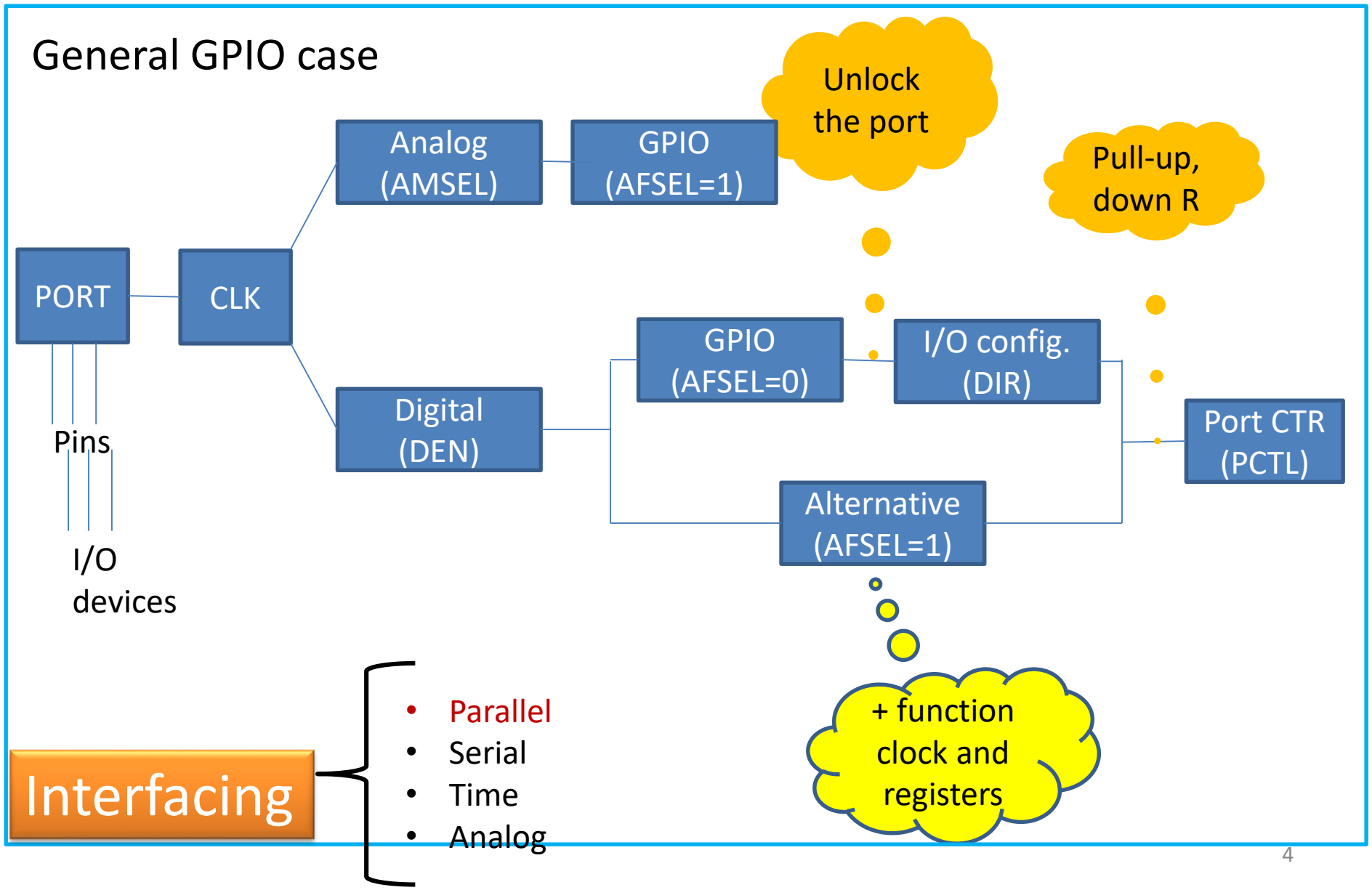
CET 241: Day 8-9

Dr. Noori KIM

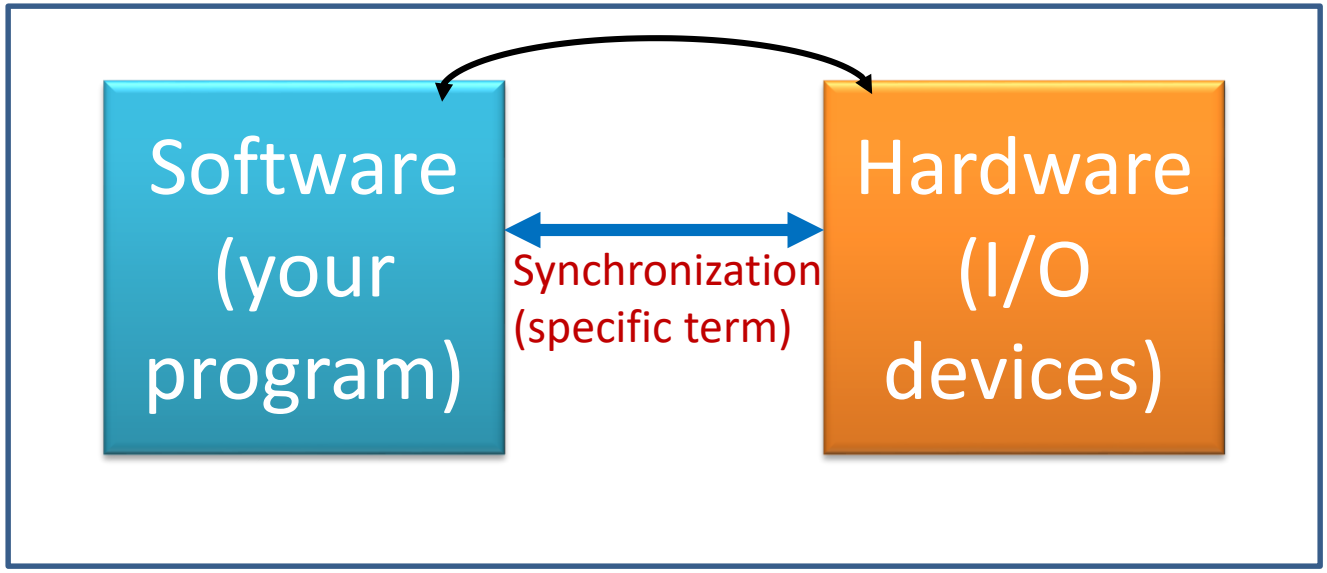
Synchronization

Dr. Noori Kim

Interfacing



- The purpose of interfacing?
  - To interact with external I/O devices
- In other words of interfacing?
  - = How the software synchronizes with the hardware to compensate speed difference

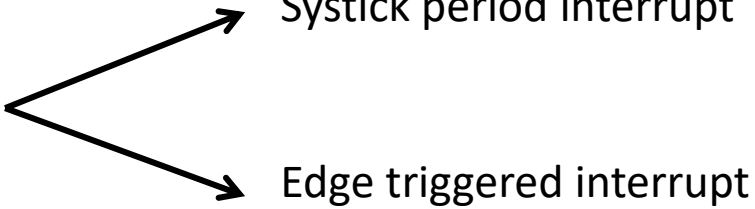


What is the Synchronization?

- When **connecting two components** together, we need a way to **Synchronize** them
  - Components include both software and hardware
- To compensate (or bridge) **speed mismatch** of communicating devices
  - A is too fast compared to B
  - Then, A is waiting for B until B's task is done

Types of Synchronization

1. Blind cycle
   - Basic idea: waiting for Δt (fixed)
2. Busy wait (polling)
   - Basic idea: checking a flag
3. Interrupt
   - Basic idea: suspending



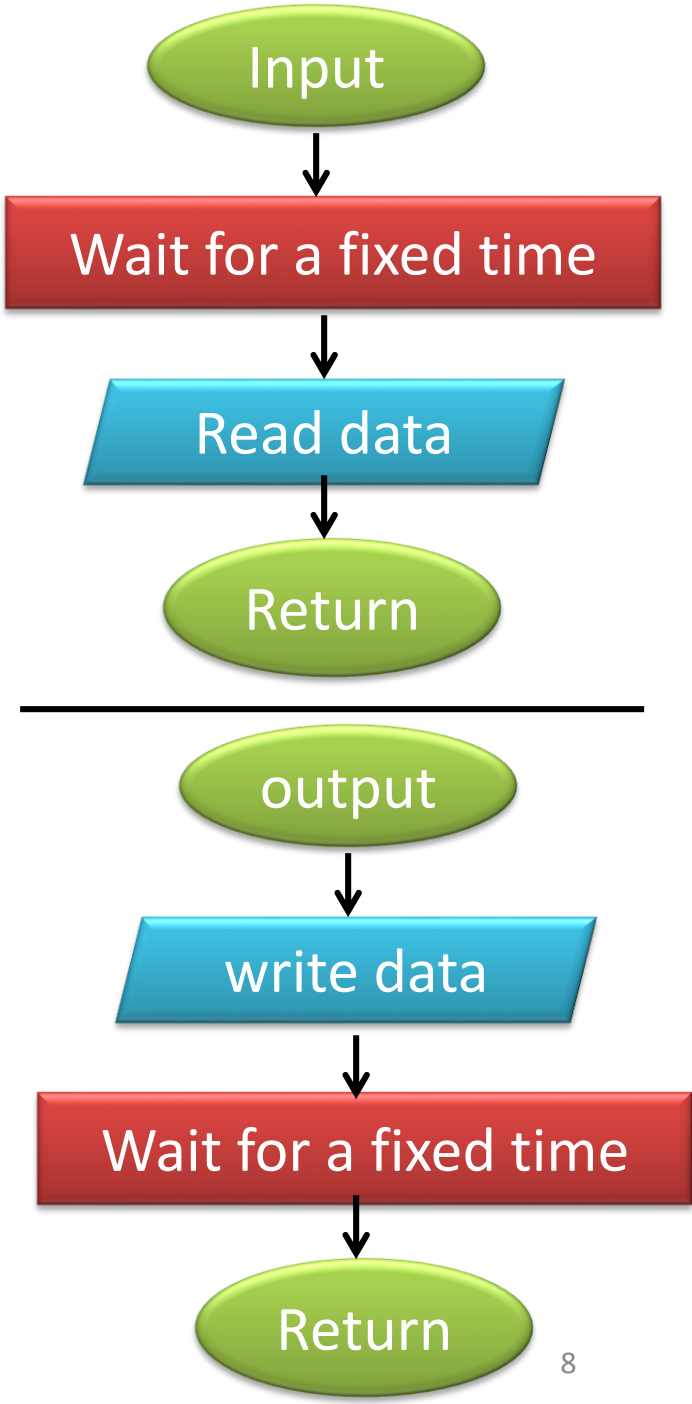
4. Periodic Polling
   - Basic idea: a type of interrupt satisfied by both a clock and I/O devices
5. Direct Memory Access (DMA)
   - Basic idea: transferring data directly to/from memory without software's acknowledgement

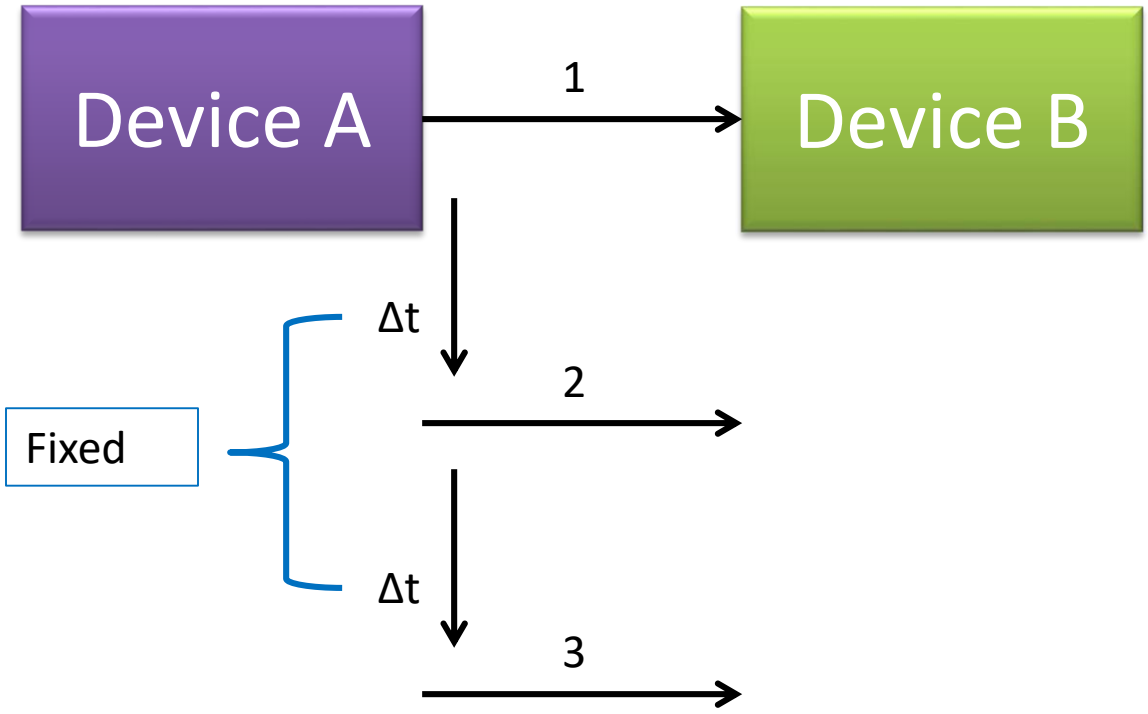
1. Blind cycle

- One of the simplest way
- Fixed delay in the software
- Assuming that I/O will complete before the fixed time
- Example in your code:

```
delay (16000);    //wait for 1ms
```

You fix the time

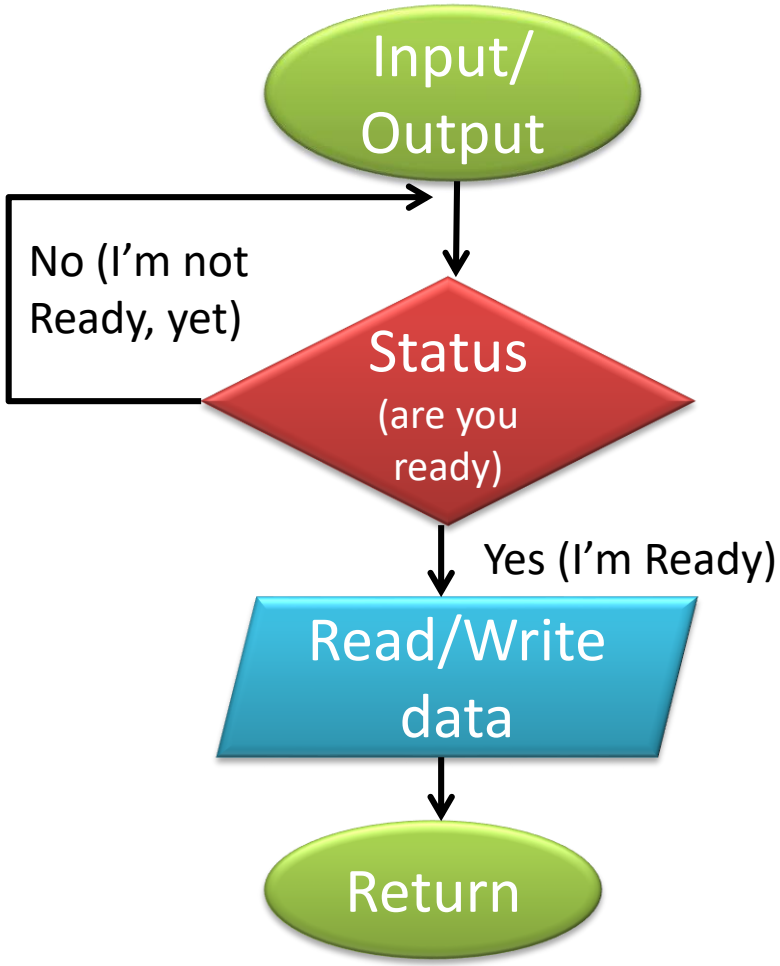


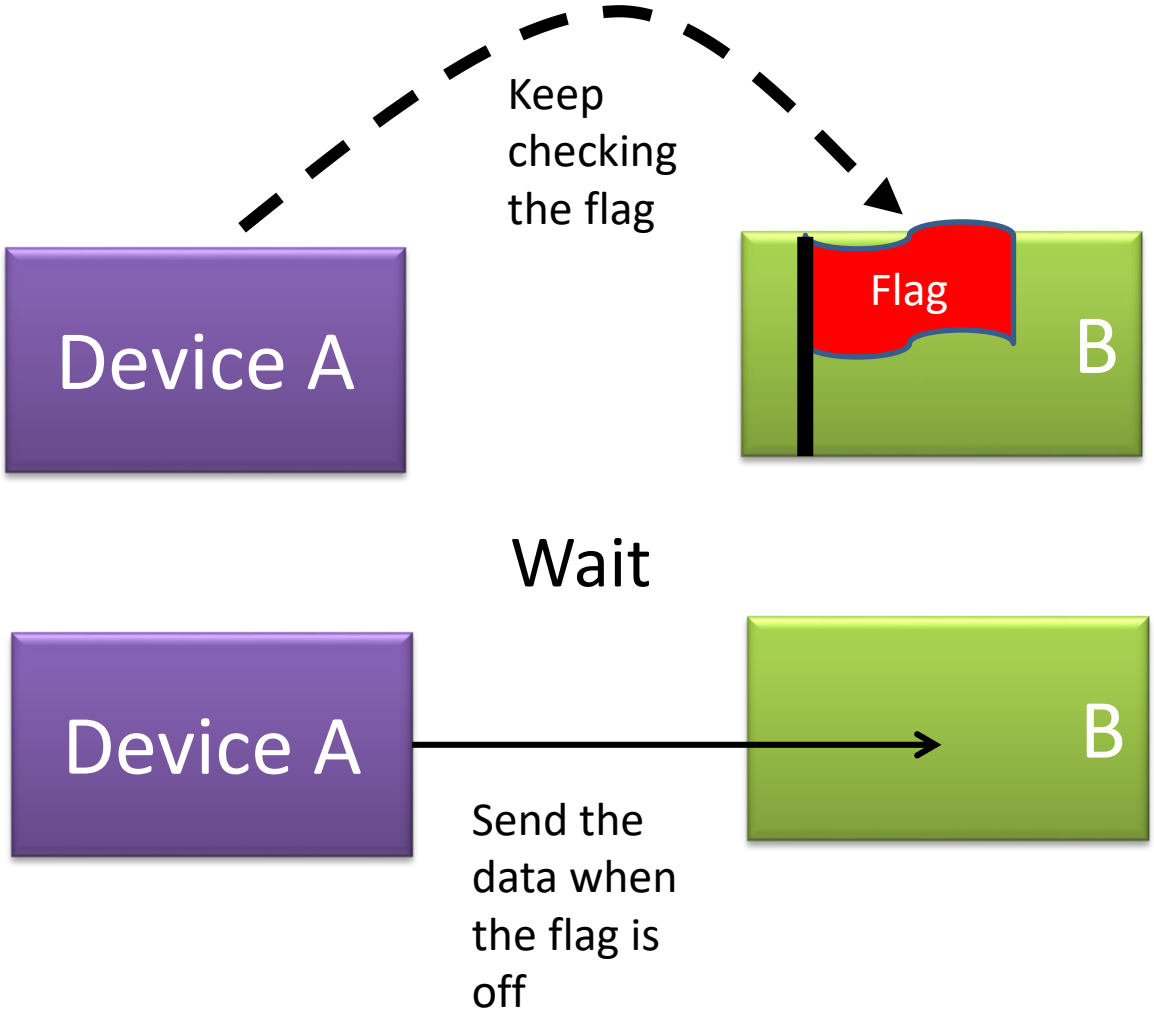


- Δt is determined by B's speed (i.e., datasheet)
- Good for a simple device
- What if B's speed is varying?

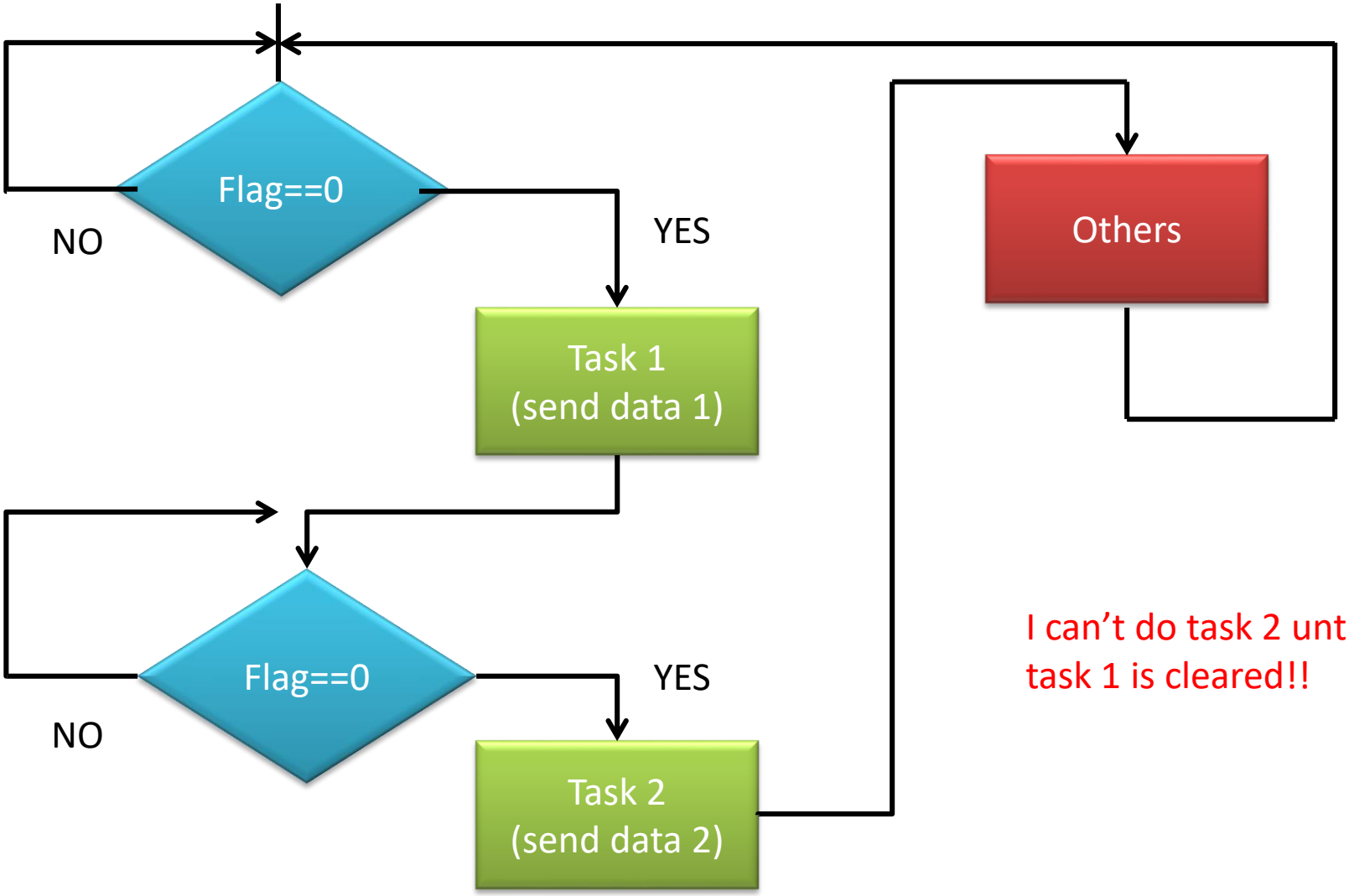
2. Busy wait (polling)

- Software **checks a status flag** in the device
  - Input device: software waits until the input device has new data, and read
  - Output device: software writes data, triggers, and waits until the device is finished.
- Not good for RT systems

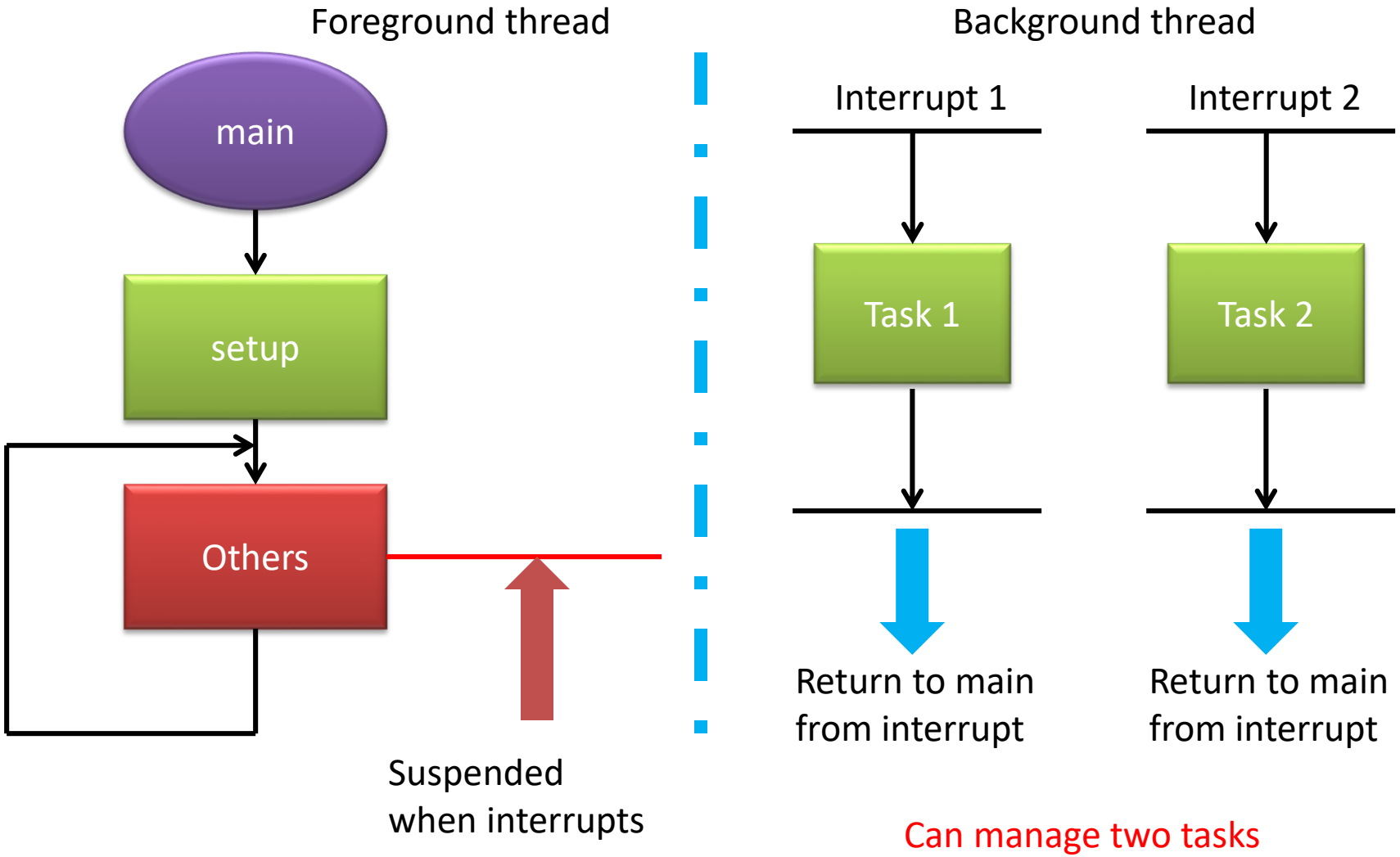



Keep
checking
the flag
Device A
Flag
B
Wait
Device A
B
Send the
data when
the flag is
off

- Scenario (2 tasks are performing)



3. Overall idea behind an interrupt



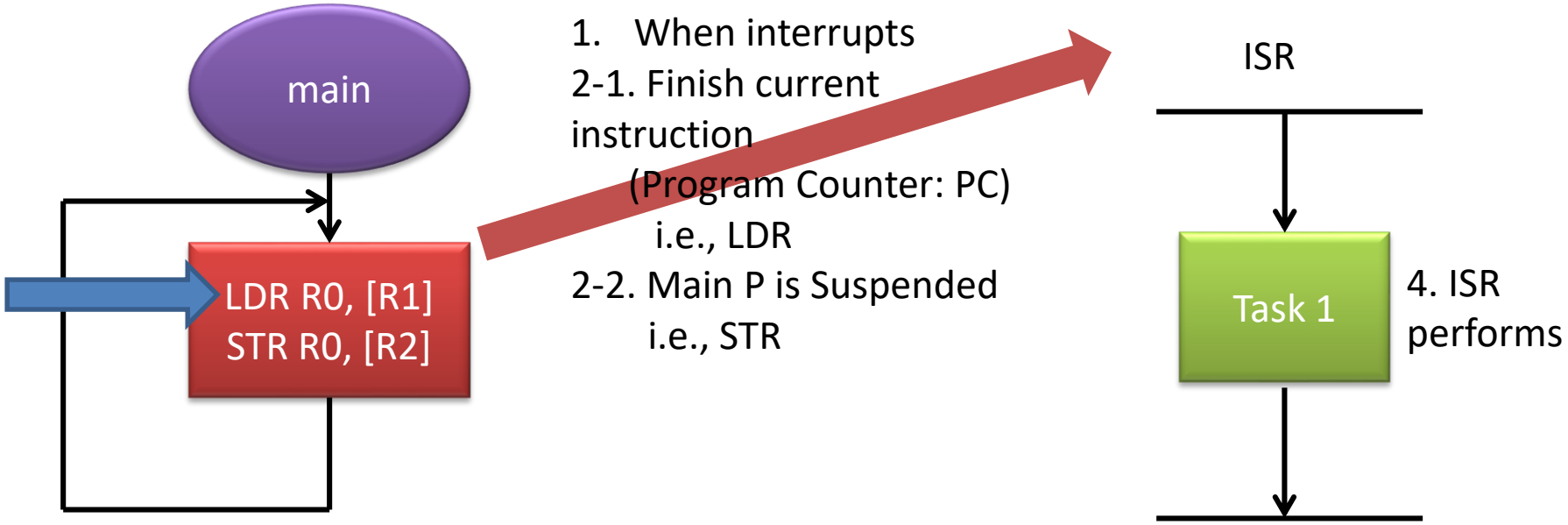
Can manage two tasks
based on priorities

- Interrupt synchronization is used where the system is fairly complex (i.e., many I/O devices, RT systems)
- The most flexible synchronization method
- Not all I/O module has this: needs a support by I/O modules
- We will learn about this later

- We’ve talked about general synchronization methods.
  - Actually we’ve used these but just did not appreciate the concept.
- Having these concepts in our mind,
  - We will learn about one of serial interfaces, UART.
  - We will apply busy-wait (polling) synchronization on the UART example (lab H).

Context switch

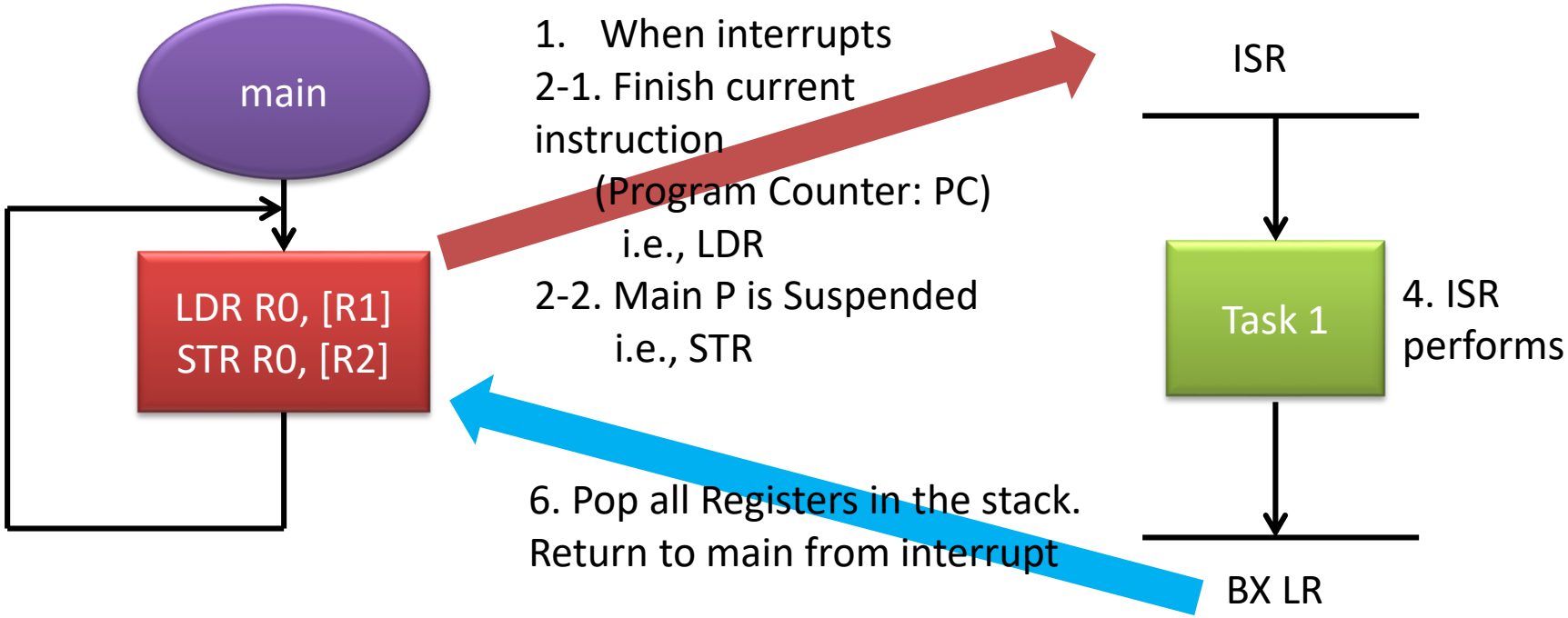
- Two threads are existing: main P and interrupting task
  - Main program triggers Interrupt Service Routine (ISR)
  - ISR: functions without input nor output
- When the flag in RIS set, interrupt occurs
  - Now, the context is switched from Main to ISR



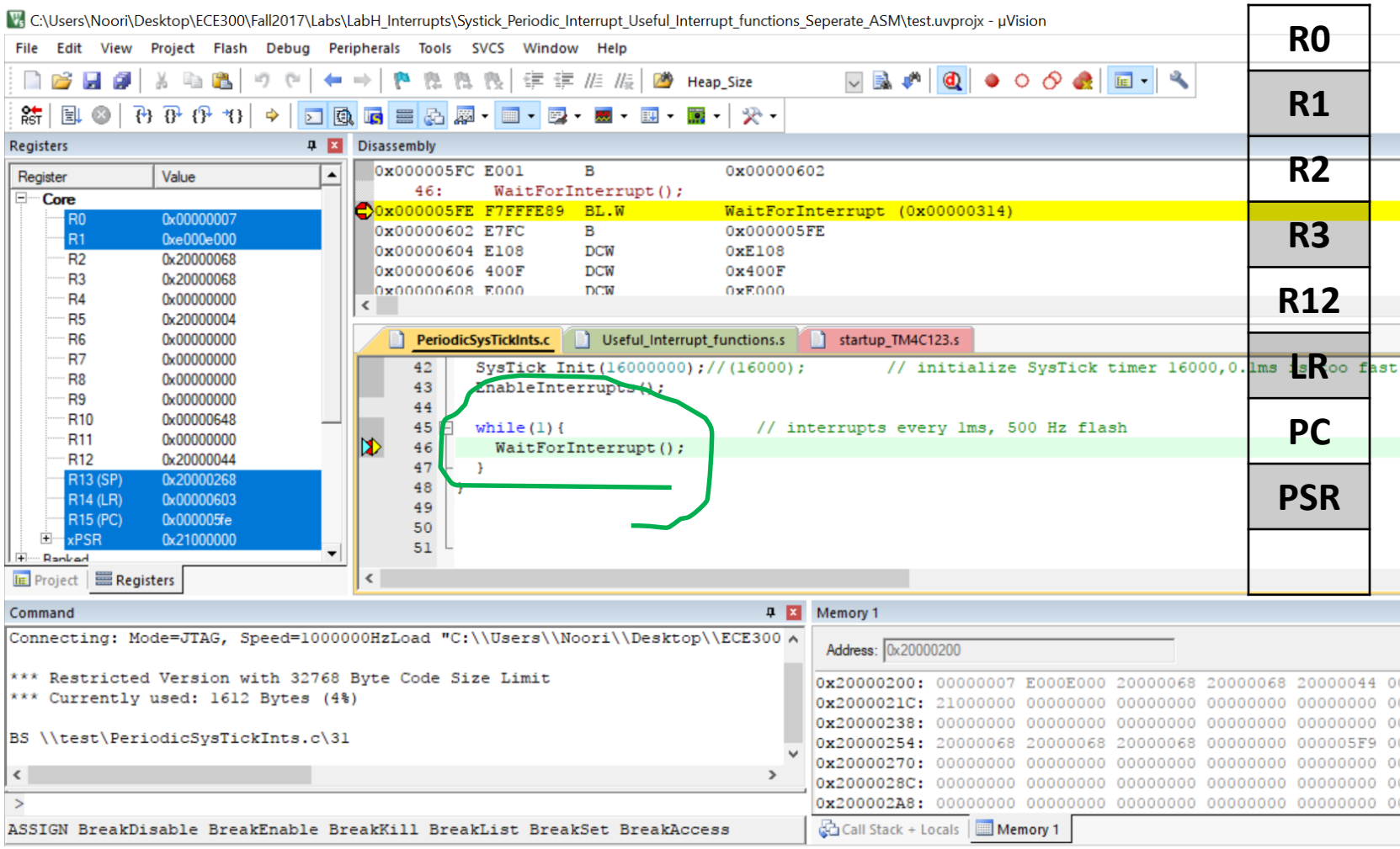
3. LR=0xFFFF.FFF9
(I am running in ISR)

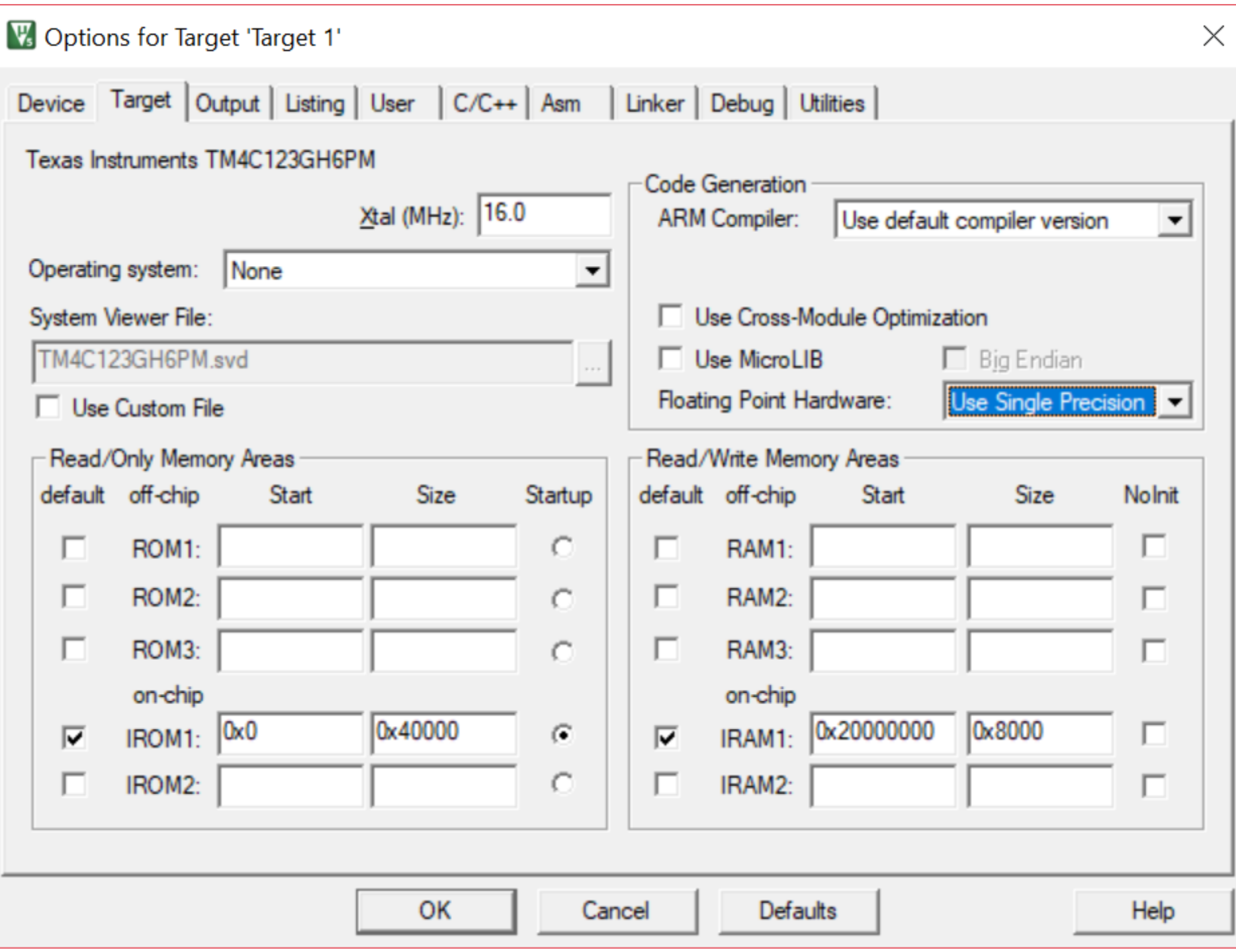
5. PC is loaded ISR

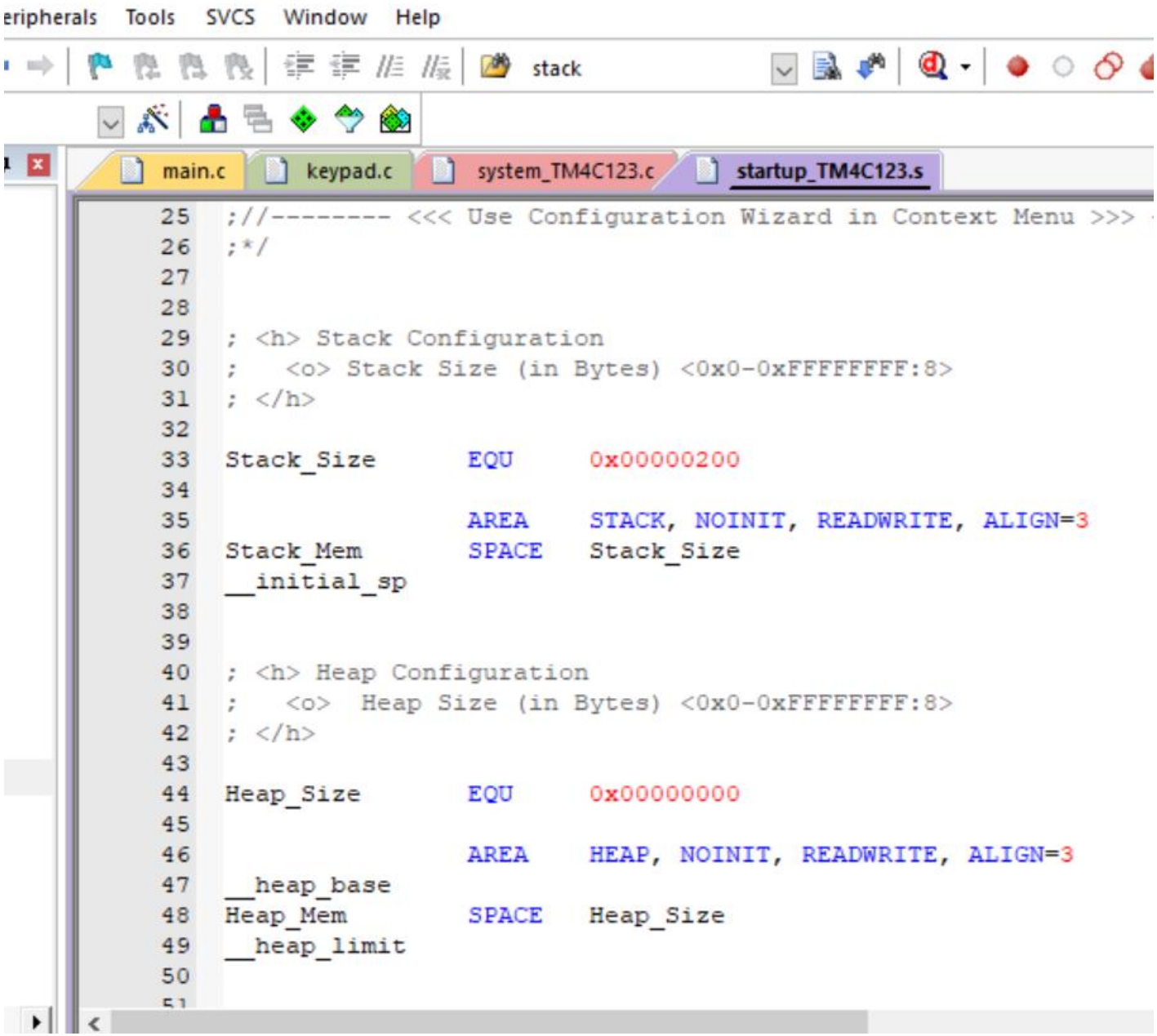
SP ⟶


main
LDR R0, [R1]
STR R0, [R2]
1. When interrupts
2-1. Finish current instruction
(Program Counter: PC)
i.e., LDR
2-2. Main P is Suspended
i.e., STR
ISR
Task 1
4. ISR performs
6. Pop all Registers in the stack.
Return to main from interrupt
BX LR

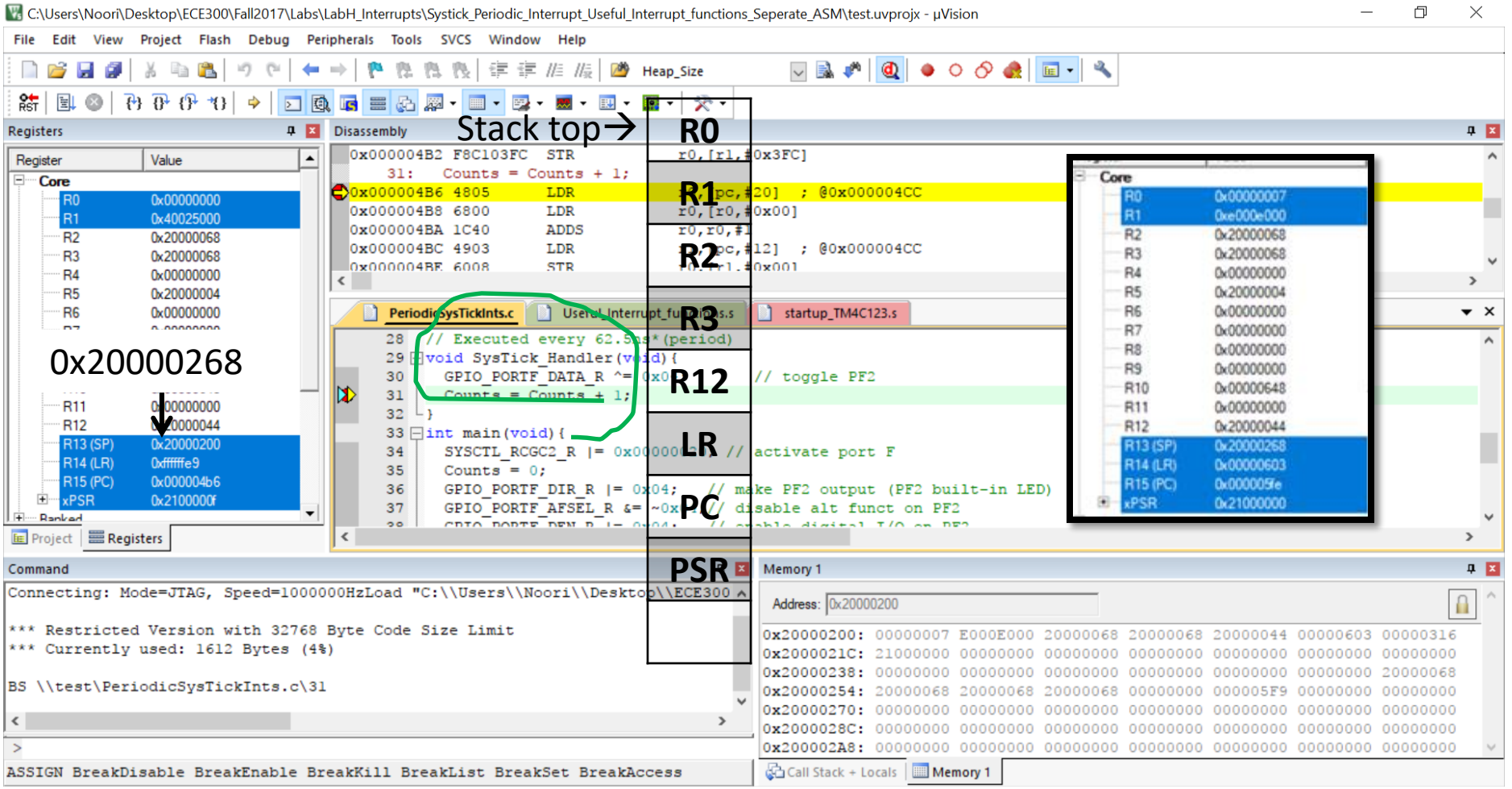
Ex: Before into interrupt (SysTick)

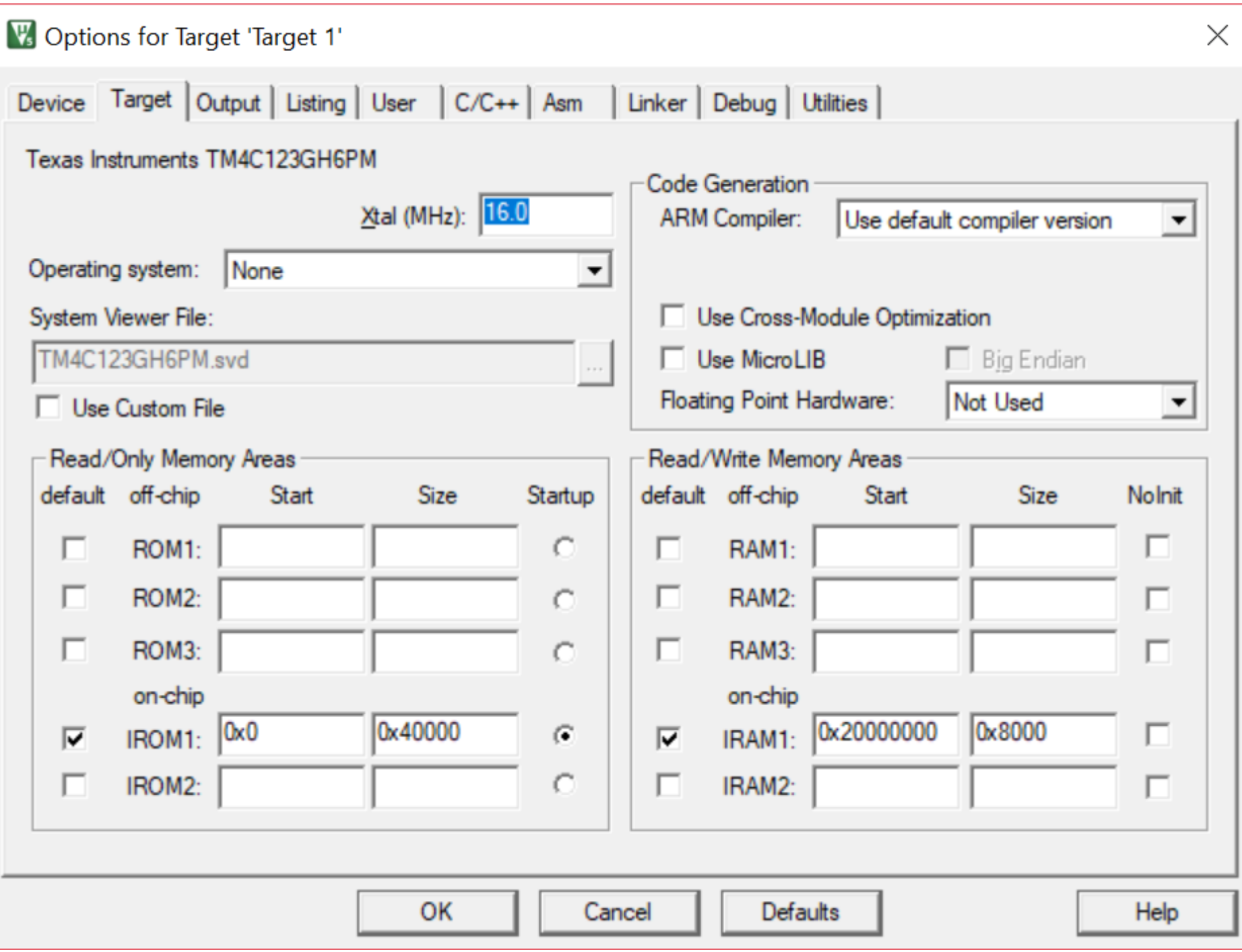


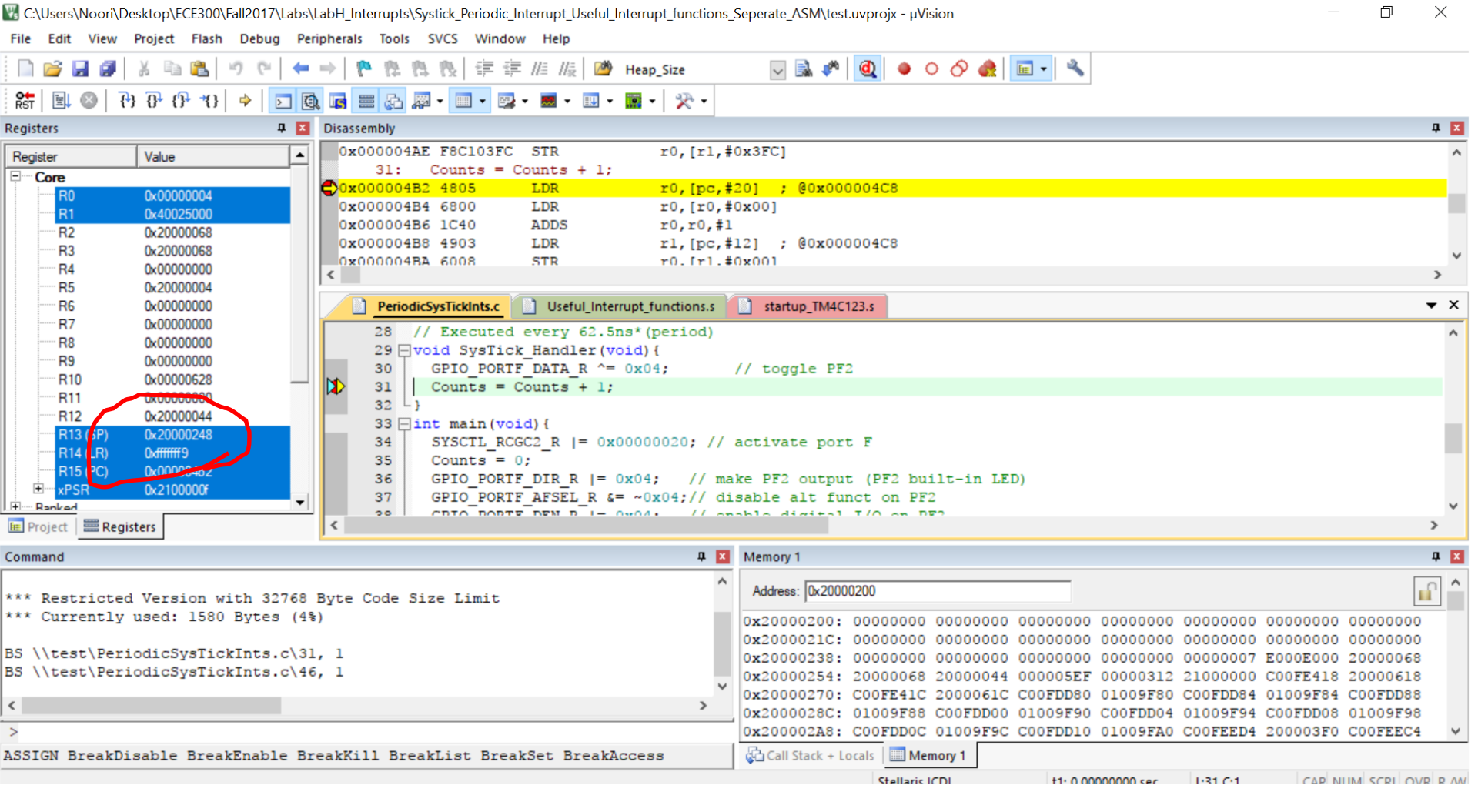

Options for Target 'Target 1'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Texas Instruments TM4C123GH6PM
Xtal (MHz):
16.0
Operating system:
None
System Viewer File:
TM4C123GH6PM.svd
Use Custom File
Code Generation
ARM Compiler:
Use default compiler version
Use Cross-Module Optimization
Use MicroLIB
Big Endian
Floating Point Hardware:
Use Single Precision
Read/Only Memory Areas
default
off-chip
Start
Size
Startup
ROM1:
ROM2:
ROM3:
on-chip
IROM1:
0x0
0x40000
IROM2:
Read/Write Memory Areas
default
off-chip
Start
Size
NoInit
RAM1:
RAM2:
RAM3:
on-chip
IRAM1:
0x20000000
0x8000
IRAM2:
OK
Cancel
Defaults
Help


eripherals Tools SVCS Window Help
stack
main.c
keypad.c
system_TM4C123.c
startup_TM4C123.s
25 ;//-------- <<< Use Configuration Wizard in Context Menu >>>
26 ;*/
27
28
29 ; <h> Stack Configuration
30 ; <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
31 ; </h>
32
33 Stack_Size EQU 0x00000200
34
35 AREA STACK, NOINIT, READWRITE, ALIGN=3
36 Stack_Mem SPACE Stack_Size
37 __initial_sp
38
39
40 ; <h> Heap Configuration
41 ; <o> Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
42 ; </h>
43
44 Heap_Size EQU 0x00000000
45
46 AREA HEAP, NOINIT, READWRITE, ALIGN=3
47 __heap_base
48 Heap_Mem SPACE Heap_Size
49 __heap_limit
50

When it is in ISR




Options for Target 'Target 1'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Texas Instruments TM4C123GH6PM
Xtal (MHz):
16.0
Operating system:
None
System Viewer File:
TM4C123GH6PM.svd
Use Custom File
Code Generation
ARM Compiler:
Use default compiler version
Use Cross-Module Optimization
Use MicroLIB
Big Endian
Floating Point Hardware:
Not Used
Read/Only Memory Areas
default
off-chip
Start
Size
Startup
ROM1:
ROM2:
ROM3:
on-chip
IROM1:
0x0
0x40000
IROM2:
Read/Write Memory Areas
default
off-chip
Start
Size
NoInit
RAM1:
RAM2:
RAM3:
on-chip
IRAM1:
0x20000000
0x8000
IRAM2:
OK
Cancel
Defaults
Help


C:\Users\Noori\Desktop\ECE300\Fall2017\Labs\LabH_Interrupts\Systick_Periodic_Interrupt_Useful_Interrupt_functions_Seperate_ASM\test.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Heap_Size
Registers
Register Value
Core
R0 0x00000004
R1 0x40025000
R2 0x20000068
R3 0x20000068
R4 0x00000000
R5 0x20000004
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000628
R11 0x00000000
R12 0x20000044
R13 (SP) 0x20000248
R14 (LR) 0xffffffff9
R15 (PC) 0x000004B2
xPSR 0x2100000f
Banked
Project
Registers
Disassembly
0x000004AE F8C103FC STR r0,[r1,#0x3FC]
31: Counts = Counts + 1;
0x000004B2 4805 LDR r0,[pc,#20] ; @0x000004C8
0x000004B4 6800 LDR r0,[r0,#0x00]
0x000004B6 1C40 ADDS r0,r0,#1
0x000004B8 4903 LDR r1,[pc,#12] ; @0x000004C8
0x000004BA 6008 STR r0,[r1,#0x00]
PeriodicSysTickInts.c
Useful_Interrupt_functions.s
startup_TM4C123.s
28 // Executed every 62.5ns*(period)
29 void SysTick_Handler(void){
30 GPIO_PORTF_DATA_R ^= 0x04; // toggle PF2
31 Counts = Counts + 1;
32 }
33 int main(void){
34 SYSCTL_RCGC2_R |= 0x00000020; // activate port F
35 Counts = 0;
36 GPIO_PORTF_DIR_R |= 0x04; // make PF2 output (PF2 built-in LED)
37 GPIO_PORTF_AFSEL_R &= ~0x04;// disable alt funct on PF2
Command
*** Restricted Version with 32768 Byte Code Size Limit
*** Currently used: 1580 Bytes (4%)
BS \\test\PeriodicSysTickInts.c\31, 1
BS \\test\PeriodicSysTickInts.c\46, 1
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess
Memory 1
Address: 0x20000200
0x20000200: 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x2000021C: 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000238: 00000000 00000000 00000000 00000000 00000007 E000E000 20000068
0x20000254: 20000068 20000044 000005EF 00000312 21000000 C00FE418 20000618
0x20000270: C00FE41C 2000061C C00FDD80 01009F80 C00FDD84 01009F84 C00FDD88
0x2000028C: 01009F88 C00FDD00 01009F90 C00FDD04 01009F94 C00FDD08 01009F98
0x200002A8: C00FDD0C 01009F9C C00FDD10 01009FA0 C00FEED4 200003F0 C00FEEC4
Call Stack + Locals
Memory 1

that the exception is complete, and the processor initiates the appropriate exception return sequence.

Table 2-10. Exception Return Behavior

EXC_RETURN[31:0]	Description
0xFFFF.FFE0	Reserved
0xFFFF.FFE1	Return to Handler mode. Exception return uses floating-point state from **MSP**. Execution uses **MSP** after return.
0xFFFF.FFE2 - 0xFFFF.FFE8	Reserved
0xFFFF.FFE9	Return to Thread mode. Exception return uses floating-point state from **MSP**. Execution uses **MSP** after return.
0xFFFF.FFEA - 0xFFFF.FFEC	Reserved
0xFFFF.FFED	Return to Thread mode. Exception return uses floating-point state from **PSP**. Execution uses **PSP** after return.
0xFFFF.FFEE - 0xFFFF.FFF0	Reserved
0xFFFF.FFF1	Return to Handler mode. Exception return uses non-floating-point state from **MSP**. Execution uses **MSP** after return.
0xFFFF.FFF2 - 0xFFFF.FFF8	Reserved
0xFFFF.FFF9	Return to Thread mode. Exception return uses non-floating-point state from **MSP**. Execution uses **MSP** after return.
0xFFFF.FFFA - 0xFFFF.FFFC	Reserved
0xFFFF.FFFD	Return to Thread mode.

Summary: Context switch 5 steps

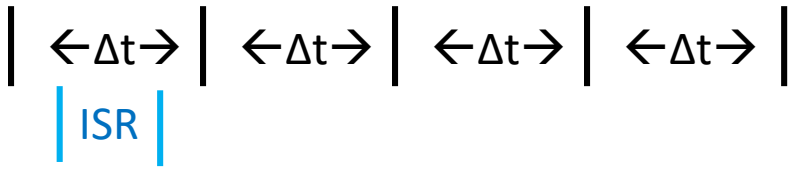
1. Finish instruction
2. Suspend: push to stack the followings
   - R0, R1,R2,R3, R12, R14(LR), R15(PS), PSR
   - R4-R11 will not be used in ISR (AAPCS)
3. LR=0xFFFF.FFF9
   - To set the LR to a special code
   - The code means I am running in ISR

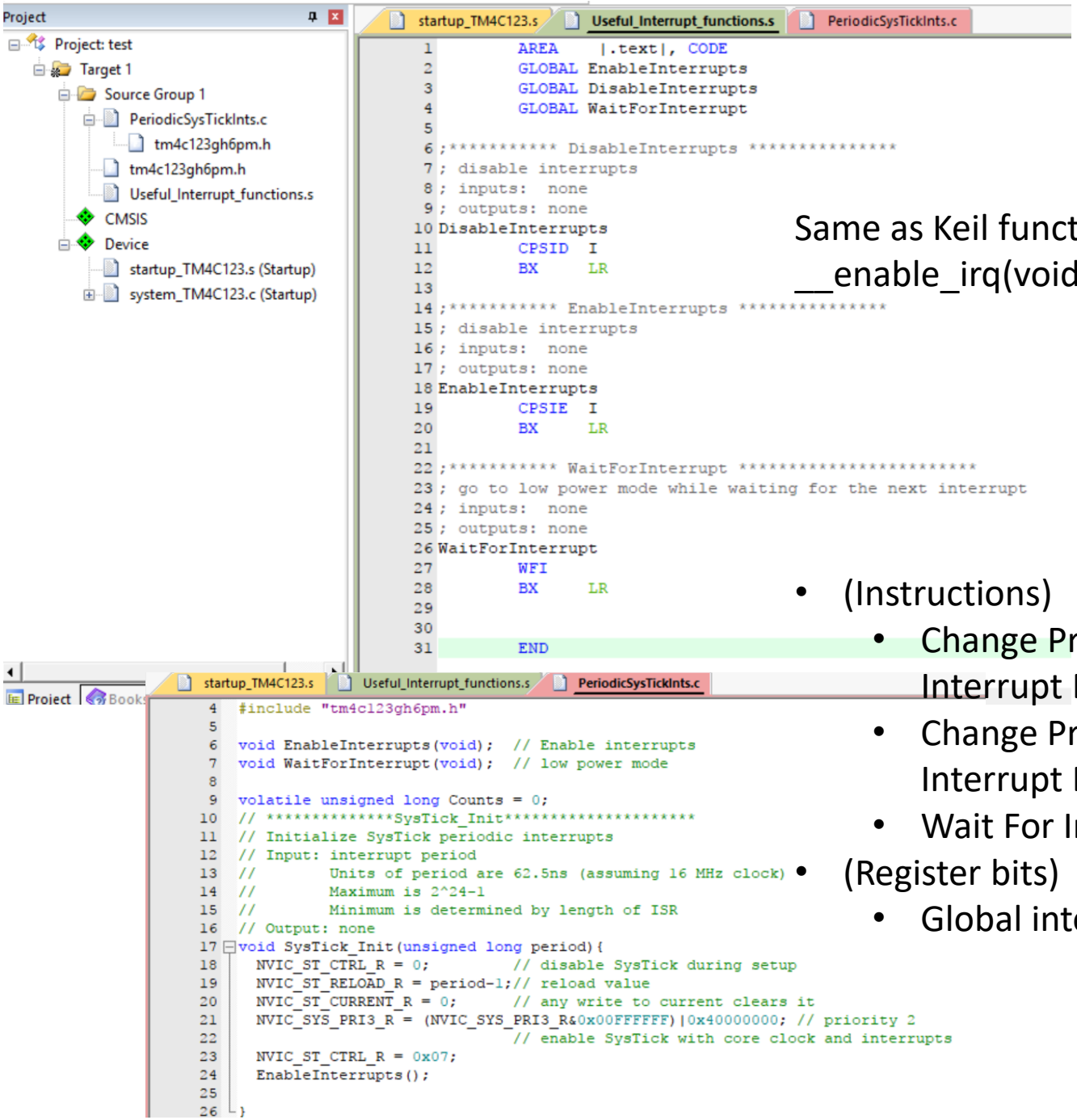
4. IPSR (Interrupt Program Status Register)
   - Contains the interrupt number of the device
   - Among the hundreds of interrupts, the IPSR register is set to the interrupt number.
5. PC is loaded with ISR

- Switch back: To come back to main P at the end of ISR
  - BX LR: This pops all 8 registers in the stack

Dos and Donts for ISR

- Do
  1) Acknowledge: clear a trigger flag by software
     - Not clearing the flag: the interrupt continuous to be triggered again and again: crash
  2) Short ISR guarantee
     - All interrupt get services
- Do not do
  1) No delay loop in ISR
  2) Triggering time in between ISRs (Δt) must be long compared to the time executing each ISR
     - Otherwise the pending ISR never get serviced



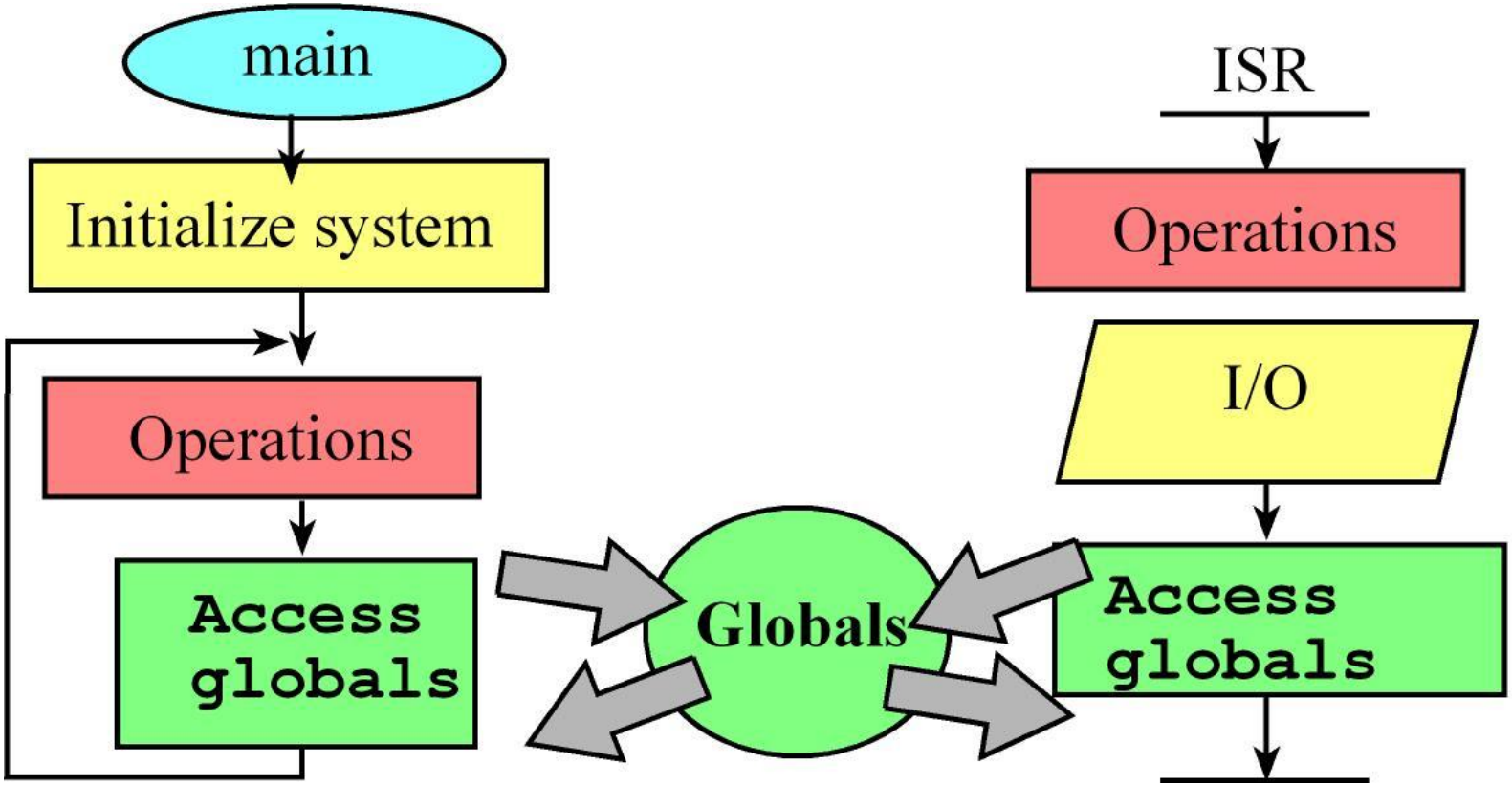


Same as Keil function __enable_irq(void);

- (Instructions)
  - Change Processor State Interrupt Disable
  - Change Processor State Interrupt Enable
  - Wait For Interrupt
- (Register bits)
  - Global interrupt enable: I bi

- We’ve talked about **a general process of Interrupt** as a kind of synchronization methods (speed mismatch compensation)
- And we’ve talked about **Context Switch**: when there is an interrupt, current program context has to be switched to ISR

- Now, we will talk about **Inter Thread Communication**: **ways** that main thread communicates with ISR threads while interrupting is occurred
  - 3 ways will be discussed: **Flag/mailbox/FIFO**

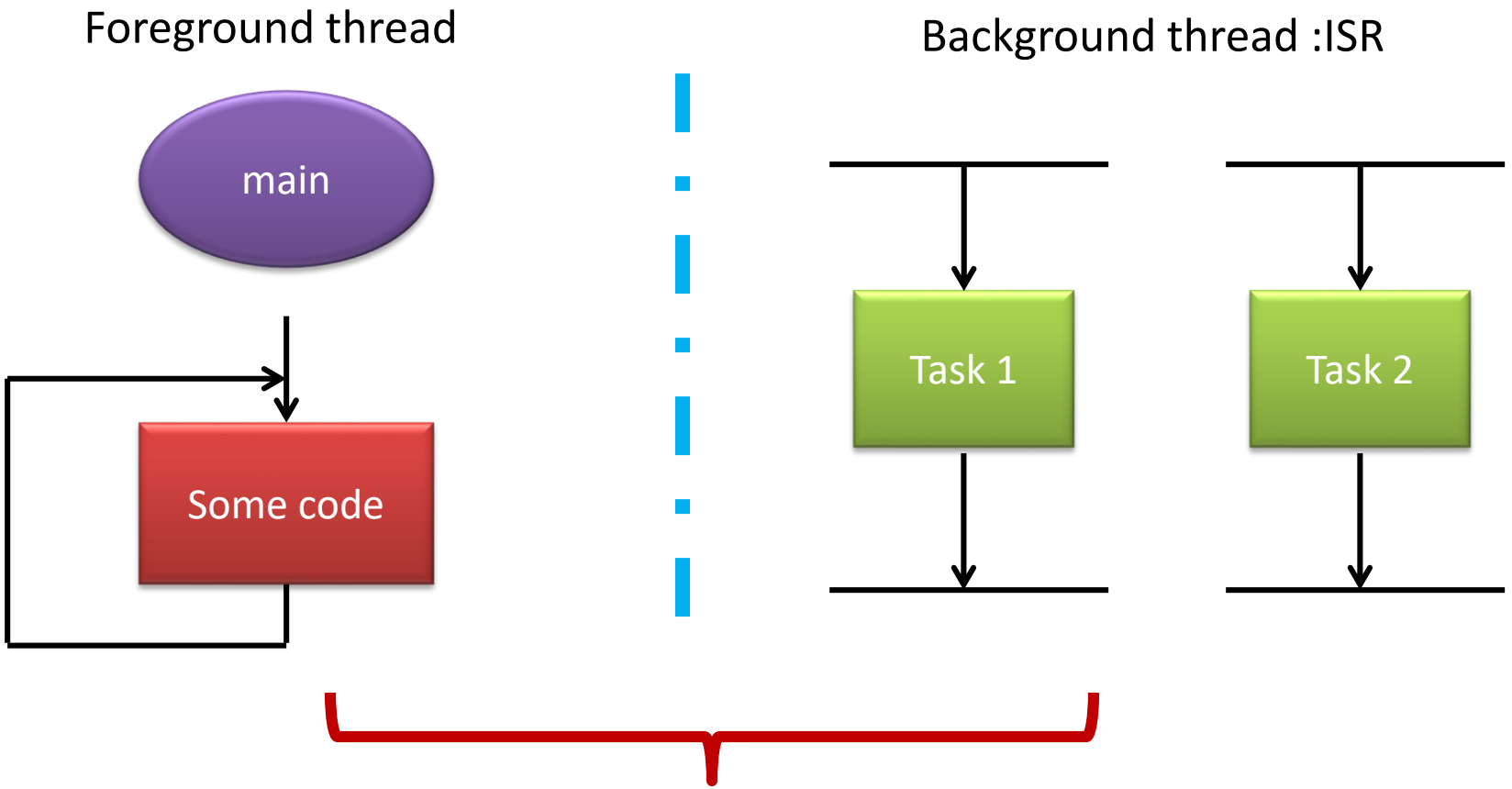


Threads communicate via shared global variables.

Multi-threading: one main program (foreground thread) and multiple ISRs (background threads)

- For regular function calls
  - Using the registers and stack to pass parameters
- **Interrupt threads**
  - Separating registers and stack.
  - Registers are automatically saved by the processor as it switches from the main program (foreground thread) to the **interrupt service routine** (background thread).
  - Exiting an ISR will restore the registers back to their previous values.
- One cannot pass data from the main program to the interrupt service routine using registers or the stack.

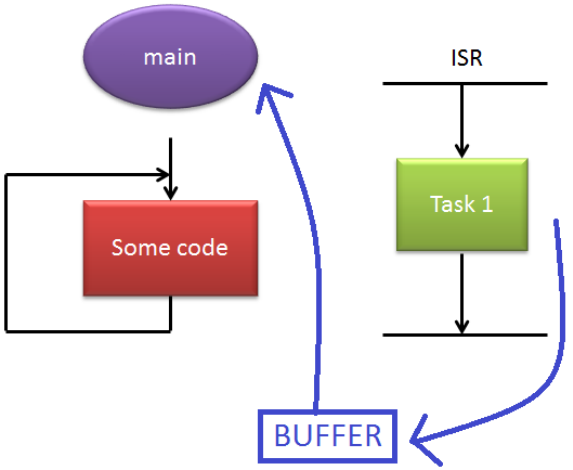
Inter Thread Communication



Can’t use local variables to communicate between two types of thread

- Losing information as we change context when we perform ISR

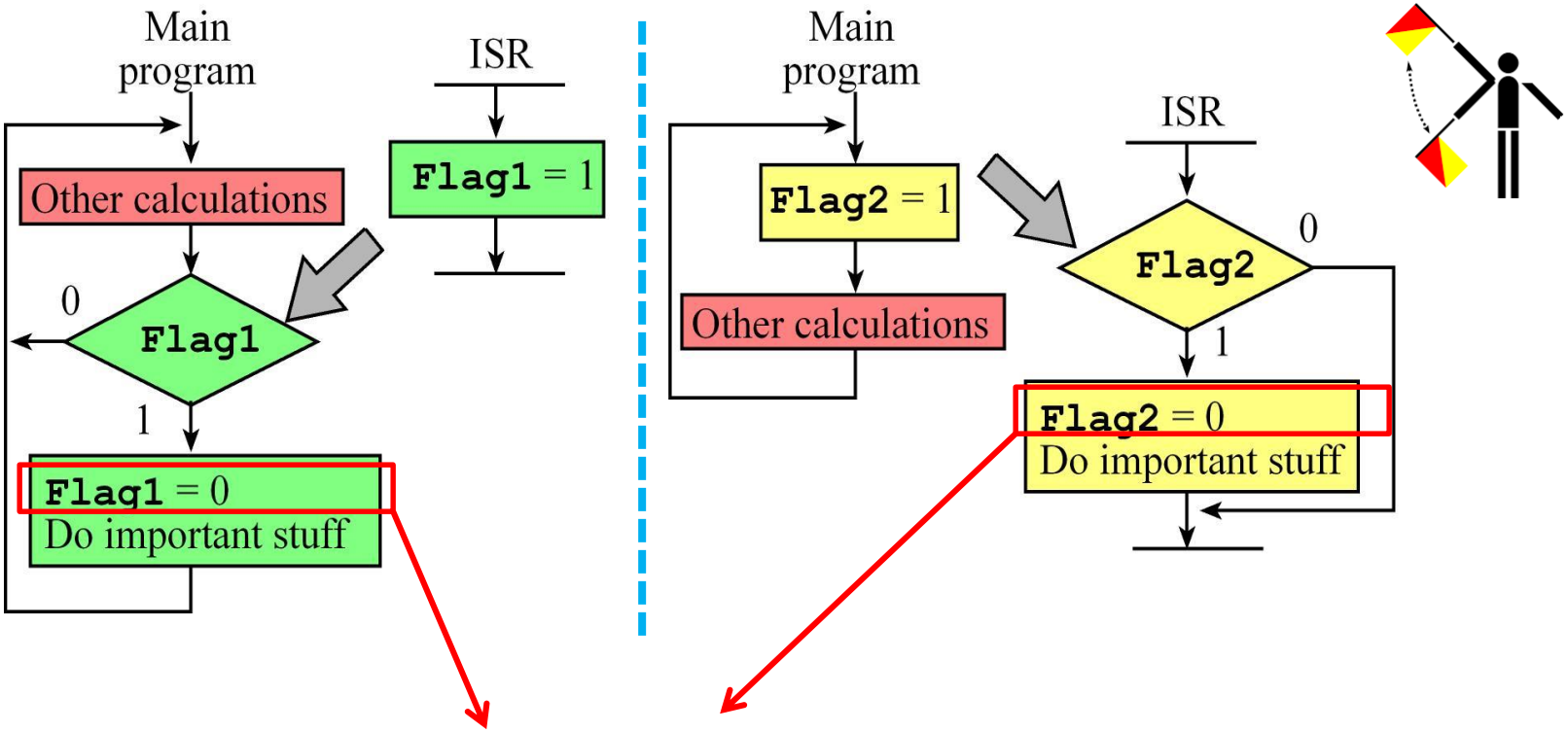
- Therefore we use: **Global memory or variable**
  1. Flag
  2. Mail box
     - Data
     - Status
  3. Buffers: FIFO, a queue, allows us to buffer info.



If ISR wants to communicate with the foreground thread, adds info to a buffer (FIFO)

- We will talk about interrupt communication methods (between mainP and ISR) by means of those global memories

1. Flag method (a binary semaphore)

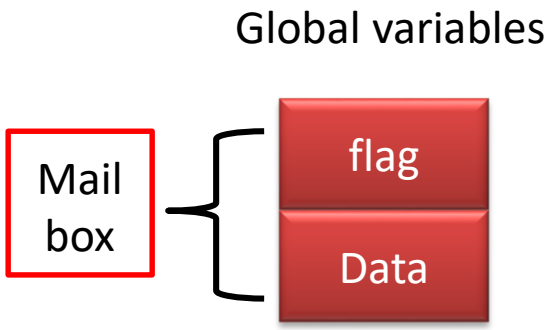


Acknowledge: Clear the flags when it is done (the flag can be triggered by both)

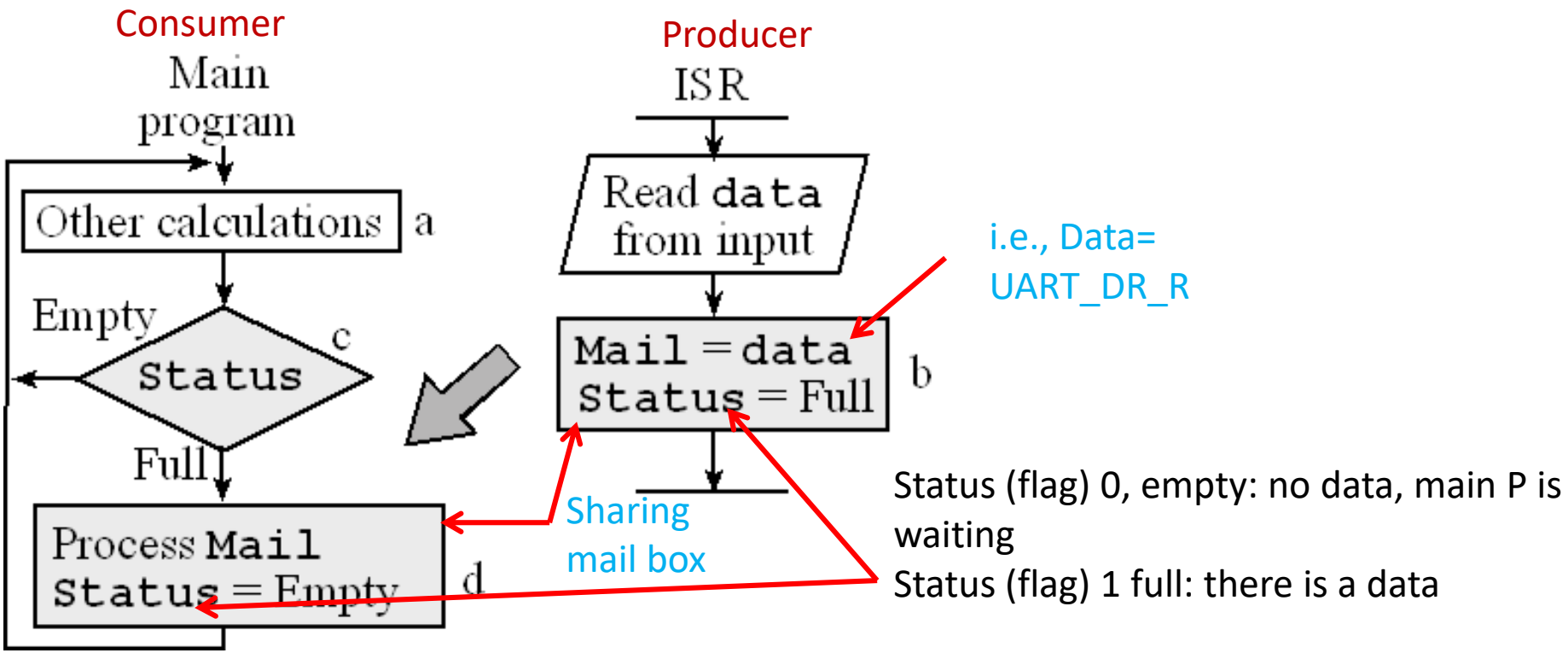
- Background and foreground threads are communicating via the global variable flags
- If Flag is 0, the interrupt has not yet happened. If it is 1, the interrupt has occurred.

- The flag must exist as **a private global variable** with restricted access to only these two code pieces.
  - In C: the qualifier **static** to a global variable to restrict access to software within the same file.

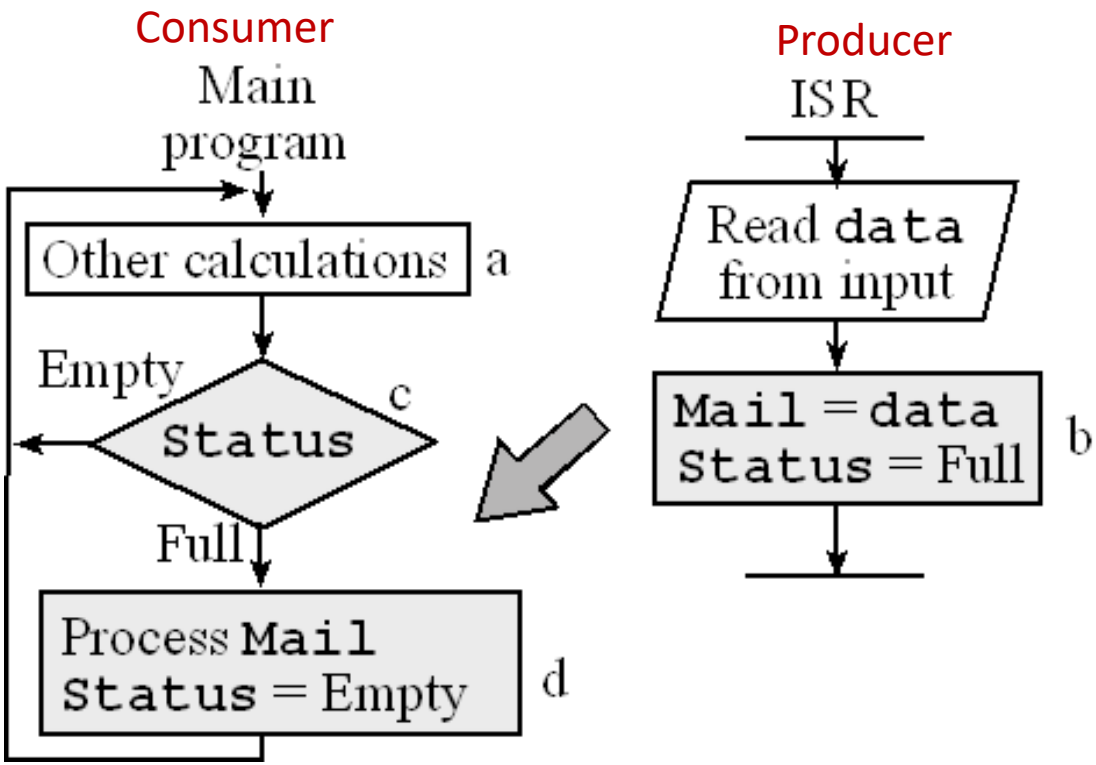
2. Mail box method (Producer-Consumer paradigm)



- Accessible for both background and foreground threads
- Background thread produces data or Foreground thread produces data



a. The mailbox is empty, the input device is idle and the main program is performing other tasks while occasionally checking the status of the mailbox.
b. The ISR reads data from input device and saves it in Mail, and then it sets Status to full.



c. The main program recognizes Status is full (in block C).

d. The main program consumes data from Mail, sets Status to empty.

- Notice that even though there are two threads, only one is active at a time (the main program -->ISR-->the main program)

Mail box context switch example

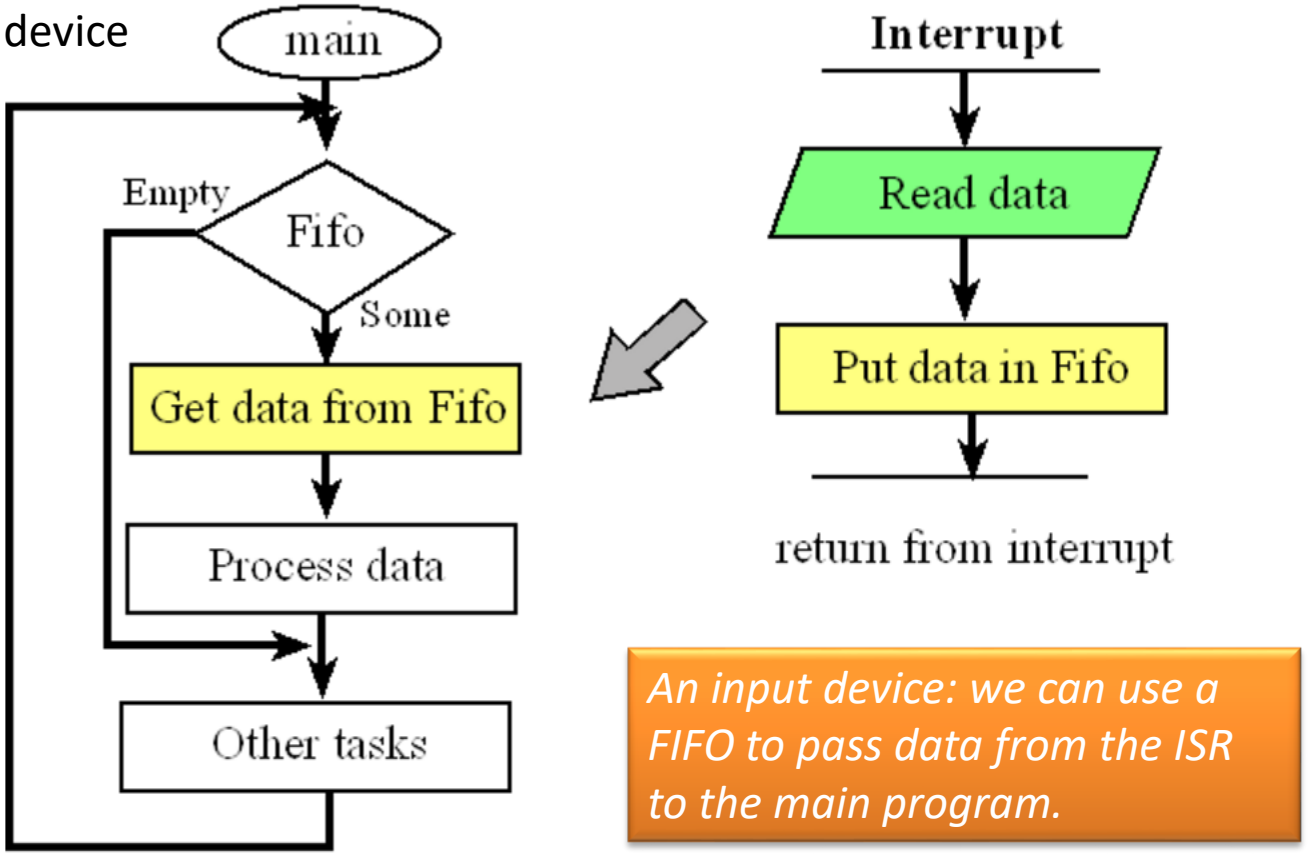
```
void SysTick_Handler(void){
  // Reads ADC & store in mailbox w/ flag
  Distance = Convert(ADC0_In());
  Flag = 1;
}
```

3. FIFO queue (First In First Out) method

- Similar to the mailbox but allows buffering: storing data in a first come first served manner.
  - An extending idea of mail box with many data
- Two functions
  - Put: data-in
  - Get: data-out
- Properties of FIFO: Order preserving
  - Order which I put in will be the order that I will get in
  - Data is streamed from one thread to the other

BTW, Stack is a LIFO (last in first out)

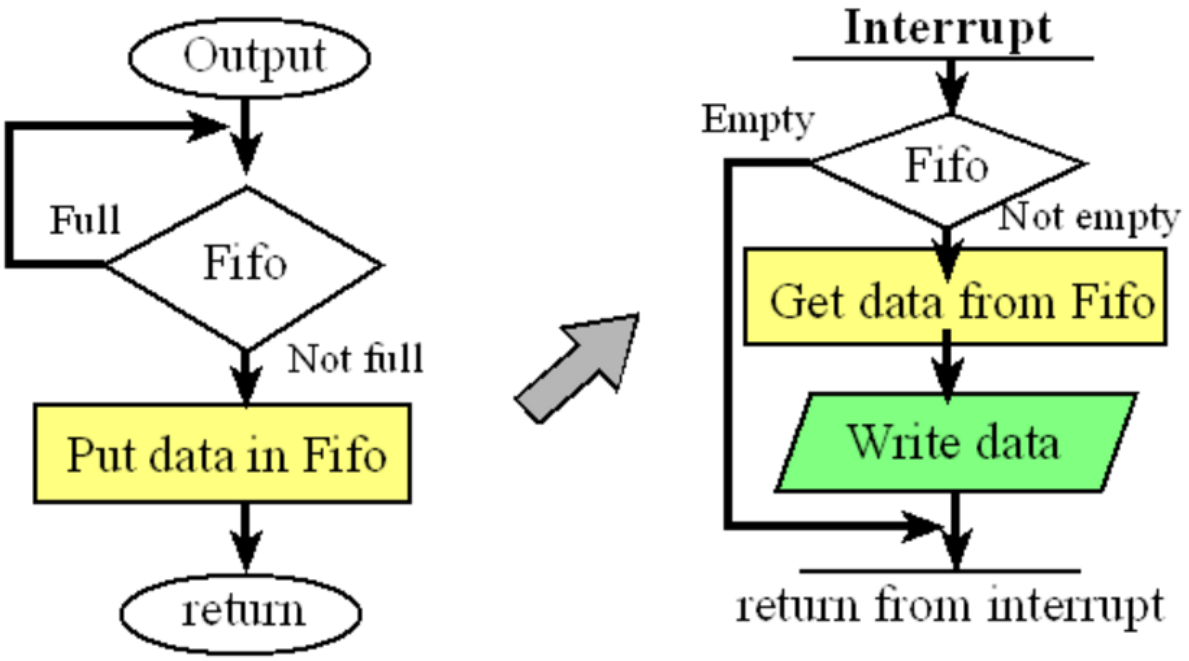
For an input device



An input device: we can use a FIFO to pass data from the ISR to the main program.

1. The ISR reads the data from the input device, and puts the data in the FIFO.
2. Whenever the main program is idle, it will attempt to get data from the FIFO.
3. If data were to exist, that data will be processed.

For an output device

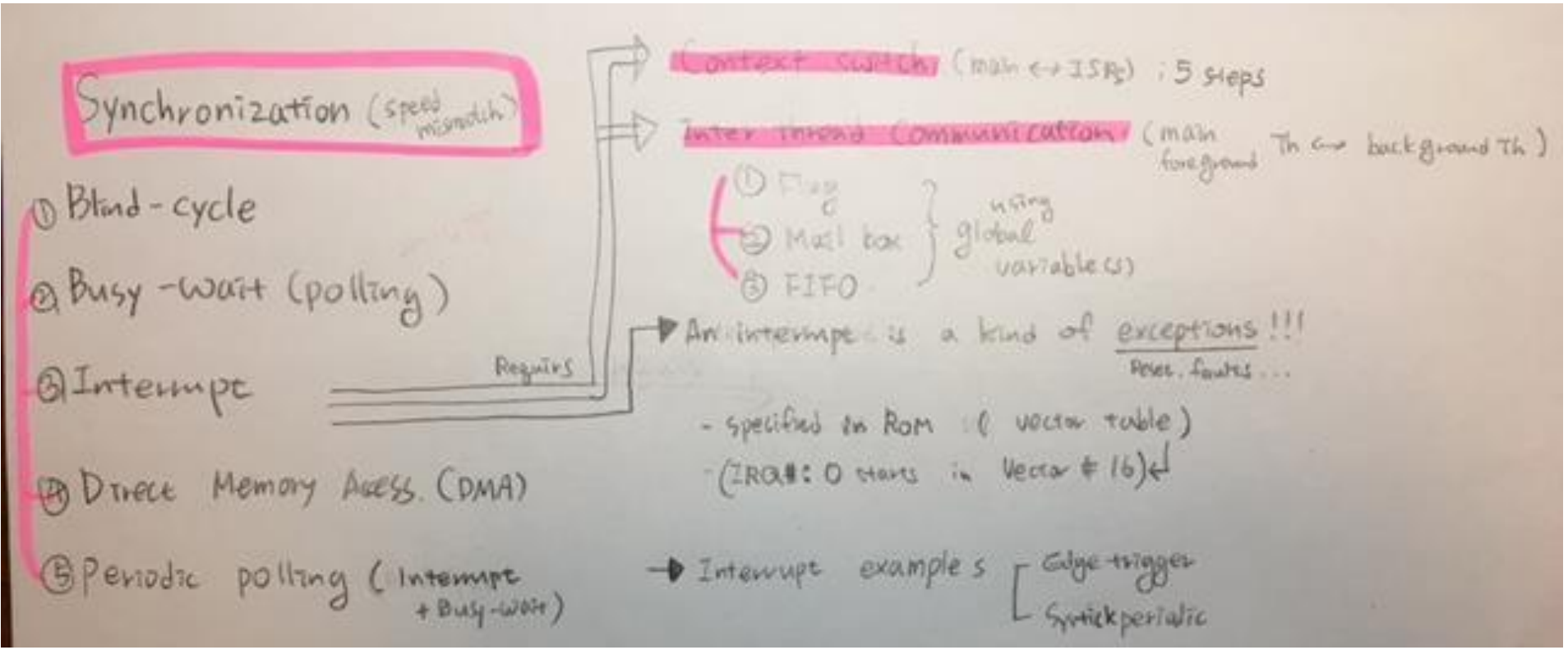


For an output device: we can use a FIFO to pass data from the main program to the ISR.

1. The main program puts data into the FIFO whenever it wishes to perform output.
2. Check FIFO whether it is full or not
3. An interrupt occurs when the output device is idle
4. The ISR gets from the FIFO
5. Write the data to the output device.

Whenever the ISR sees the FIFO is empty, it could cause the output device to become idle.

- We have talked about



Exceptions

- Events that need servicing
- Every exception has an associated 32-bit vector that stores memory location where the ISR (handles exceptions) is located

```
EXPORT  __Vectors
__Vectors                                ; address      interrupt
    DCD     StackMem + Stack            ; 0x00000000 Top of Stack
    DCD     Reset_Handler               ; 0x00000004 Reset Handler
    DCD     NMI_Handler                 ; 0x00000008 NMI Handler
    DCD     HardFault_Handler           ; 0x0000000C Hard Fault Handler
    DCD     MemManage_Handler           ; 0x00000010 MPU Fault Handler
    DCD     BusFault_Handler            ; 0x00000014 Bus Fault Handler
    DCD     UsageFault_Handler          ; 0x00000018 Usage Fault Handler
    DCD     0                           ; 0x0000001C Reserved
    DCD     0                           ; 0x00000020 Reserved
    DCD     0                           ; 0x00000024 Reserved
    DCD     0                           ; 0x00000028 Reserved
    DCD     SVC_Handler                 ; 0x0000002C SVCall Handler
    DCD     DebugMon_Handler            ; 0x00000030 Debug Monitor Handler
    DCD     0                           ; 0x00000034 Reserved
    DCD     PendSV_Handler              ; 0x00000038 PendSV Handler
    DCD     SysTick_Handler             ; 0x0000003C SysTick Handler
     ⋮          ⋮                              ⋮
```

: init SP

: init PC, calls the reset vector

: Handler starts

- Why do we call it as a vector?

Interrupt vector table

From Wikipedia, the free encyclopedia
(Redirected from Interrupt vector)

This article is about the general concept. For its implementation found in x86 processors, see Interrupt descriptor table.

An "**interrupt vector table**" (IVT) is a data structure that associates a list of interrupt handlers with a list of interrupt requests in a table of interrupt vectors. An entry in the interrupt vector is the address of the interrupt handler. While the concept is common across processor architectures, each IVT may be implemented in an architecture-specific fashion. For example, a dispatch table is one method of implementing an interrupt vector table.

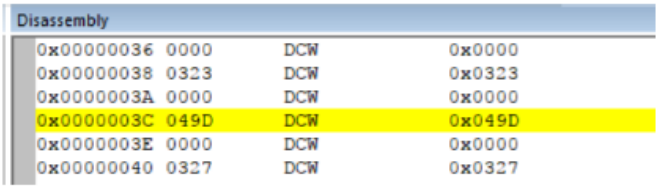
- What about entry?

(3pts) _____ address of each interrupt ISR can be called a Vector since it provides not only a direction to the related handler but also _____ address of the ISR. What is the same word that can be inserted in the two blanks to complete the ISR description properly? Refer to the following hints.

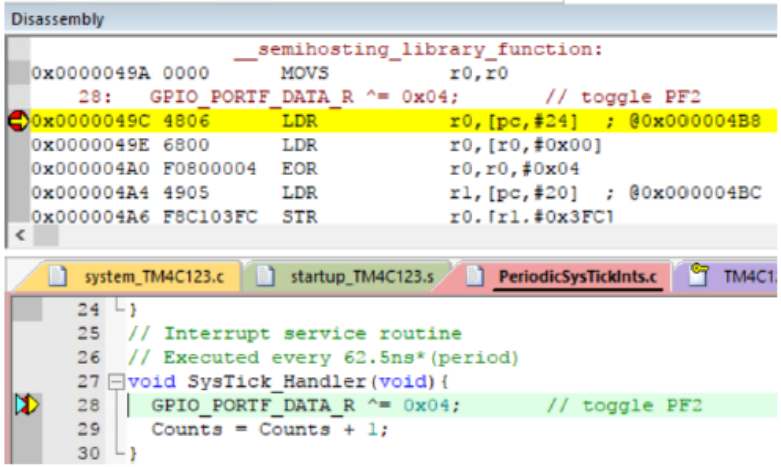
- The following is a portion of interrupt vector table for TM4C123 MCU.

Vector address	Number	IRQ	ISR name in Startup.s	NVIC	Priority bits
0x00000038	14	-2	PendSV_Handler	NVIC_SYS_PRI3_R	23 – 21
0x0000003C	15	-1	SysTick_Handler	NVIC_SYS_PRI3_R	31 – 29
0x00000040	16	0	GPIOPortA_Handler	NVIC_PRI0_R	7 – 5
0x00000044	17	1	GPIOPortB_Handler	NVIC_PRI0_R	15 – 13
0x00000048	18	2	GPIOPortC_Handler	NVIC_PRI0_R	23 – 21
0x0000004C	19	3	GPIOPortD_Handler	NVIC_PRI0_R	31 – 29
0x00000050	20	4	GPIOPortE_Handler	NVIC_PRI1_R	7 – 5

- The following window captures a portion of initialization of a program in disassembly window, before the program enters to its main function.



- The following window captures when the program is in ISR….



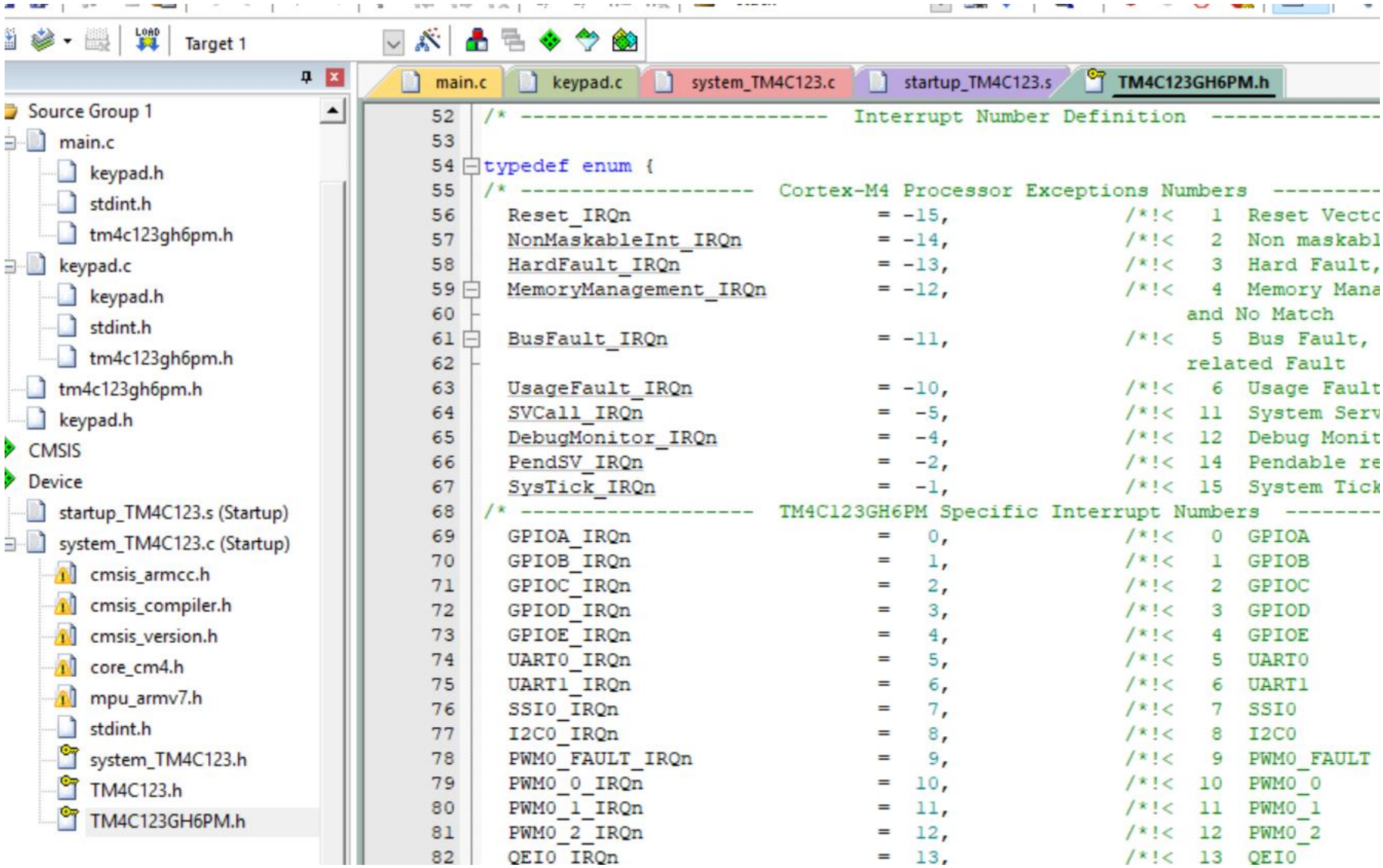
Types of exceptions

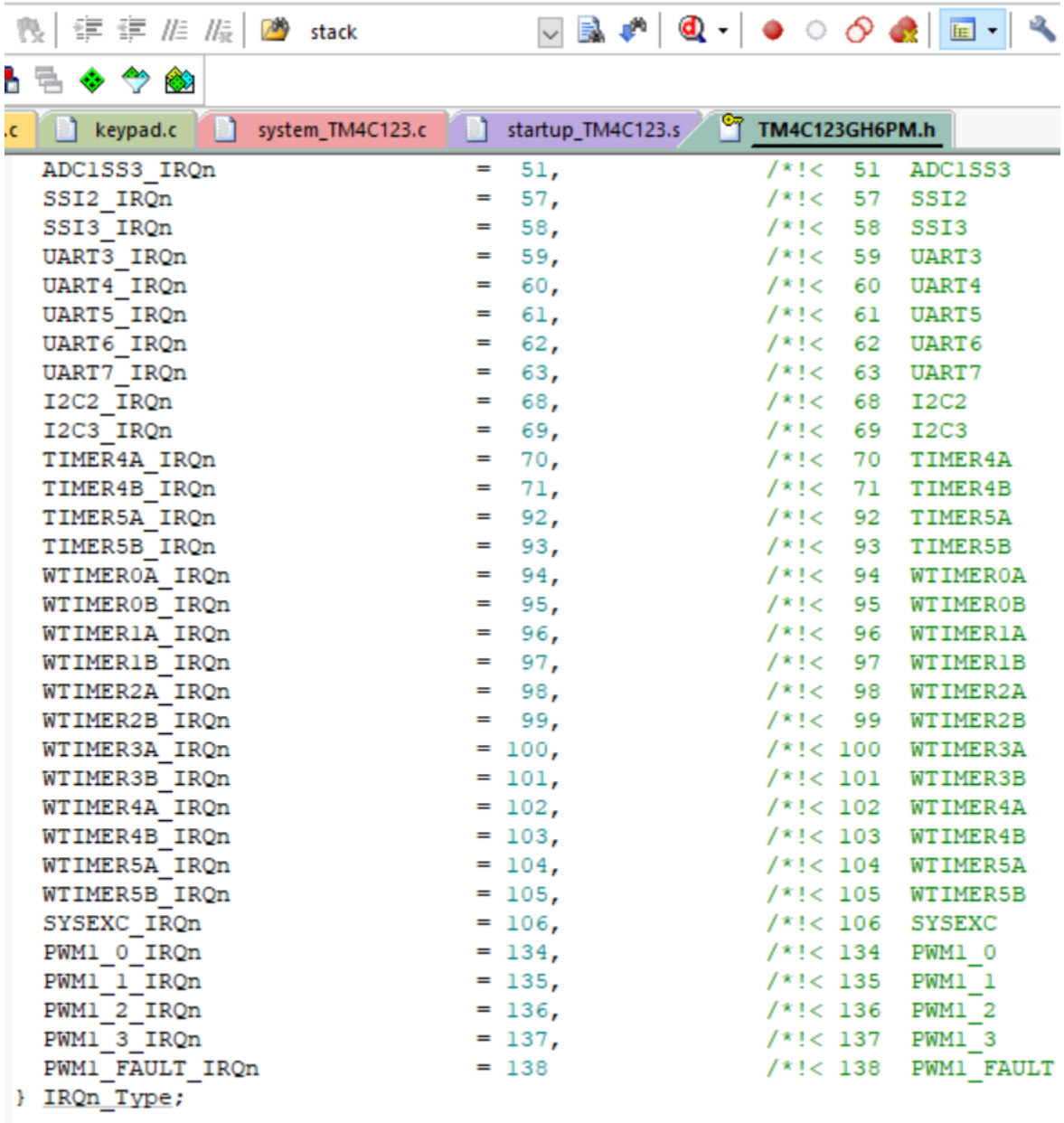
- All of the above + **interrupts**
  - Interrupts (IRQs) use vectors (entry of interrupt)
- An interrupt (IRQ) is an exception signaled by a peripheral or generated by a software request.
  - Can be prioritized
  - Asynchronous to instruction execution
  - Peripherals, which are capable of generating interrupts, include GPIO, UART, SSI, Timers, ADC, and so on.
  - Each (type of) interrupt needs to be supported by an MCU

- startup_TM4C123.s: each interrupt has a dummy interrupt handler that does not perform any thing.
- To write an interrupt handler
  1. Find out the name of the dummy interrupt handler in the interrupt vector table
  2. Reuse that name

```
EXPORT   __Vectors
__Vectors                                        ; address       interrupt
     DCD        StackMem + Stack                ; 0x00000000 Top of Stack
     DCD        Reset_Handler                   ; 0x00000004 Reset Handler
     DCD        NMI_Handler                     ; 0x00000008 NMI Handler
     DCD        HardFault_Handler               ; 0x0000000C Hard Fault Handler
     DCD        MemManage_Handler               ; 0x00000010 MPU Fault Handler
     DCD        BusFault_Handler                ; 0x00000014 Bus Fault Handler
     DCD        UsageFault_Handler              ; 0x00000018 Usage Fault Handler
     DCD        0                               ; 0x0000001C Reserved
     DCD        0                               ; 0x00000020 Reserved
     DCD        0                               ; 0x00000024 Reserved
     DCD        0                               ; 0x00000028 Reserved
     DCD        SVC_Handler                     ; 0x0000002C SVCall Handler
     DCD        DebugMon_Handler                ; 0x00000030 Debug Monitor
Handler
     DCD        0                               ; 0x00000034 Reserved
     DCD        PendSV_Handler                  ; 0x00000038 PendSV Handler
     DCD        SysTick_Handler                 ; 0x0000003C SysTick Handler
```

Vector Number	Exception Type	Priority	Vector Address
0	–	–	0x0000.0000
1	Reset	-3 (highest)	0x0000.0004
2	NMI	-2	0x0000.0008
3	Hard Fault	-1	0x0000.000C
4	Memory Management Fault	programmable	0x0000.0010
5	Bus Fault	programmable	0x0000.0014
6	Usage Fault	programmable	0x0000.0018
7 ~ 10	Reserved	–	0x0000.001C ~ 0x0000.0028
11	SVCall	programmable	0x0000.002C
12	Debug Monitor	Programmable	0x0000.0030
13	Reserved	–	0x0000.0034
14	PendSV	programmable	0x0000.0038
15	SysTick	programmable	0x0000.003C
16 ~ 255	Interrupts	programmable	0x0000.0040 ~ 0x0000.03FC


Target 1
main.c
keypad.c
system_TM4C123.c
startup_TM4C123.s
TM4C123GH6PM.h
Source Group 1
main.c
keypad.h
stdint.h
tm4c123gh6pm.h
keypad.c
keypad.h
stdint.h
tm4c123gh6pm.h
tm4c123gh6pm.h
keypad.h
CMSIS
Device
startup_TM4C123.s (Startup)
system_TM4C123.c (Startup)
cmsis_armcc.h
cmsis_compiler.h
cmsis_version.h
core_cm4.h
mpu_armv7.h
stdint.h
system_TM4C123.h
TM4C123.h
TM4C123GH6PM.h
52 /* ------------------------- Interrupt Number Definition ------------
53
54 typedef enum {
55 /* ------------------ Cortex-M4 Processor Exceptions Numbers ---------
56 Reset_IRQn = -15, /*!< 1 Reset Vecto
57 NonMaskableInt_IRQn = -14, /*!< 2 Non maskabl
58 HardFault_IRQn = -13, /*!< 3 Hard Fault,
59 MemoryManagement_IRQn = -12, /*!< 4 Memory Mana
60 and No Match
61 BusFault_IRQn = -11, /*!< 5 Bus Fault,
62 related Fault
63 UsageFault_IRQn = -10, /*!< 6 Usage Fault
64 SVCall_IRQn = -5, /*!< 11 System Serv
65 DebugMonitor_IRQn = -4, /*!< 12 Debug Monit
66 PendSV_IRQn = -2, /*!< 14 Pendable re
67 SysTick_IRQn = -1, /*!< 15 System Tick
68 /* ------------------ TM4C123GH6PM Specific Interrupt Numbers --------
69 GPIOA_IRQn = 0, /*!< 0 GPIOA
70 GPIOB_IRQn = 1, /*!< 1 GPIOB
71 GPIOC_IRQn = 2, /*!< 2 GPIOC
72 GPIOD_IRQn = 3, /*!< 3 GPIOD
73 GPIOE_IRQn = 4, /*!< 4 GPIOE
74 UART0_IRQn = 5, /*!< 5 UART0
75 UART1_IRQn = 6, /*!< 6 UART1
76 SSI0_IRQn = 7, /*!< 7 SSI0
77 I2C0_IRQn = 8, /*!< 8 I2C0
78 PWM0_FAULT_IRQn = 9, /*!< 9 PWM0_FAULT
79 PWM0_0_IRQn = 10, /*!< 10 PWM0_0
80 PWM0_1_IRQn = 11, /*!< 11 PWM0_1
81 PWM0_2_IRQn = 12, /*!< 12 PWM0_2
82 QEI0_IRQn = 13, /*!< 13 QEI0


stack
keypad.c
system_TM4C123.c
startup_TM4C123.s
TM4C123GH6PM.h
ADC1SS3_IRQn = 51, /*!< 51 ADC1SS3
SSI2_IRQn = 57, /*!< 57 SSI2
SSI3_IRQn = 58, /*!< 58 SSI3
UART3_IRQn = 59, /*!< 59 UART3
UART4_IRQn = 60, /*!< 60 UART4
UART5_IRQn = 61, /*!< 61 UART5
UART6_IRQn = 62, /*!< 62 UART6
UART7_IRQn = 63, /*!< 63 UART7
I2C2_IRQn = 68, /*!< 68 I2C2
I2C3_IRQn = 69, /*!< 69 I2C3
TIMER4A_IRQn = 70, /*!< 70 TIMER4A
TIMER4B_IRQn = 71, /*!< 71 TIMER4B
TIMER5A_IRQn = 92, /*!< 92 TIMER5A
TIMER5B_IRQn = 93, /*!< 93 TIMER5B
WTIMER0A_IRQn = 94, /*!< 94 WTIMER0A
WTIMER0B_IRQn = 95, /*!< 95 WTIMER0B
WTIMER1A_IRQn = 96, /*!< 96 WTIMER1A
WTIMER1B_IRQn = 97, /*!< 97 WTIMER1B
WTIMER2A_IRQn = 98, /*!< 98 WTIMER2A
WTIMER2B_IRQn = 99, /*!< 99 WTIMER2B
WTIMER3A_IRQn = 100, /*!< 100 WTIMER3A
WTIMER3B_IRQn = 101, /*!< 101 WTIMER3B
WTIMER4A_IRQn = 102, /*!< 102 WTIMER4A
WTIMER4B_IRQn = 103, /*!< 103 WTIMER4B
WTIMER5A_IRQn = 104, /*!< 104 WTIMER5A
WTIMER5B_IRQn = 105, /*!< 105 WTIMER5B
SYSEXC_IRQn = 106, /*!< 106 SYSEXC
PWM1_0_IRQn = 134, /*!< 134 PWM1_0
PWM1_1_IRQn = 135, /*!< 135 PWM1_1
PWM1_2_IRQn = 136, /*!< 136 PWM1_2
PWM1_3_IRQn = 137, /*!< 137 PWM1_3
PWM1_FAULT_IRQn = 138 /*!< 138 PWM1_FAULT
} IRQn_Type;

INTERRUPT VECTORS

Vector address	Number	IRQ	ISR name in Startup.s	NVIC	Priority bits
0x00000038	14	-2	PendSV_Handler	NVIC_SYS_PRI3_R	23 – 21
0x0000003C	15	-1	SysTick_Handler	NVIC_SYS_PRI3_R	31 – 29
0x00000040	16	0	GPIOPortA_Handler	NVIC_PRI0_R	7 – 5
0x00000044	17	1	GPIOPortB_Handler	NVIC_PRI0_R	15 – 13
0x00000048	18	2	GPIOPortC_Handler	NVIC_PRI0_R	23 – 21
0x0000004C	19	3	GPIOPortD_Handler	NVIC_PRI0_R	31 – 29
0x00000050	20	4	GPIOPortE_Handler	NVIC_PRI1_R	7 – 5
0x00000054	21	5	UART0_Handler	NVIC_PRI1_R	15 – 13
0x00000058	22	6	UART1_Handler	NVIC_PRI1_R	23 – 21
0x0000005C	23	7	SSI0_Handler	NVIC_PRI1_R	31 – 29
0x00000060	24	8	I2C0_Handler	NVIC_PRI2_R	7 – 5
0x00000064	25	9	PWMFault_Handler	NVIC_PRI2_R	15 – 13
0x00000068	26	10	PWM0_Handler	NVIC_PRI2_R	23 – 21
0x0000006C	27	11	PWM1_Handler		31 – 29
0x00000070	28	12	PWM2_Handler		7 – 5
0x00000074	29	13	Quadrature0_Handler		15 – 13
0x00000078	30	14	ADC0_Handler		23 – 21
0x0000007C	31	15	ADC1_Handler		31 – 29
0x00000080	32	16	ADC2_Handler	NVIC_PRI4_R	7 – 5
0x00000084	33	17	ADC3_Handler	NVIC_PRI4_R	15 – 13
0x00000088	34	18	WDT_Handler	NVIC_PRI4_R	23 – 21
0x0000008C	35	19	Timer0A_Handler	NVIC_PRI4_R	31 – 29
0x00000090	36	20	Timer0B_Handler	NVIC_PRI5_R	7 – 5
0x00000094	37	21	Timer1A_Handler	NVIC_PRI5_R	15 – 13
0x00000098	38	22	Timer1B_Handler	NVIC_PRI5_R	23 – 21
0x0000009C	39	23	Timer2A_Handler	NVIC_PRI5_R	31 – 29
0x000000A0	40	24	Timer2B_Handler	NVIC_PRI6_R	7 – 5
0x000000A4	41	25	Comp0_Handler	NVIC_PRI6_R	15 – 13
0x000000A8	42	26	Comp1_Handler	NVIC_PRI6_R	23 – 21
0x000000AC	43	27	Comp2_Handler	NVIC_PRI6_R	31 – 29
0x000000B0	44	28	SysCtl_Handler	NVIC_PRI7_R	7 – 5
0x000000B4	45	29	FlashCtl_Handler	NVIC_PRI7_R	15 – 13
0x000000B8	46	30	GPIOPortF_Handler	NVIC_PRI7_R	23 – 21
0x000000BC	47	31	GPIOPortG_Handler	NVIC_PRI7_R	31 – 29
0x000000C0	48	32	GPIOPortH_Handler	NVIC_PRI8_R	7 – 5
0x000000C4	49	33	UART2_Handler	NVIC_PRI8_R	15 – 13
0x000000C8	50	34	SSI1_Handler	NVIC_PRI8_R	23 – 21
0x000000CC	51	35	Timer3A_Handler	NVIC_PRI8_R	31 – 29
0x000000D0	52	36	Timer3B_Handler	NVIC_PRI9_R	7 – 5
0x000000D4	53	37	I2C1_Handler	NVIC_PRI9_R	15 – 13
0x000000D8	54	38	Quadrature1_Handler	NVIC_PRI9_R	23 – 21
0x000000DC	55	39	CAN0_Handler	NVIC_PRI9_R	31 – 29
0x000000E0	56	40	CAN1_Handler	NVIC_PRI10_R	7 – 5
0x000000E4	57	41	CAN2_Handler	NVIC_PRI10_R	15 – 13
0x000000E8	58	42	Ethernet_Handler	NVIC_PRI10_R	23 – 21
0x000000EC	59	43	Hibernate_Handler	NVIC_PRI10_R	31 – 29
0x000000F0	60	44	USB0_Handler	NVIC_PRI11_R	7 – 5
0x000000F4	61	45	PWM3_Handler	NVIC_PRI11_R	15 – 13
0x000000F8	62	46	uDMA_Handler	NVIC_PRI11_R	23 – 21
0x000000FC	63	47	uDMA_Error	NVIC_PRI11_R	31 – 29

Where to find the priority bits

Assigned interrupts examples in TM4C123G

Vector Number	Interrupt Request Number (IRQ#)	Vector Address	Description
0~15	–	0x0000.0000 ~ 0x0000.003C	Processor exceptions
16	0	0x0000.0040	GPIO Port A
17	1	0x0000.0044	GPIO Port B
18	2	0x0000.0048	GPIO Port C
19	3	0x0000.004C	GPIO Port D
20	4	0x0000.0050	GPIO Port E
21	5	0x0000.0054	UART0
22	6	0x0000.0058	UART1

- Each interrupt has **an vector number** and an **IRQ number**. Let *x* be the vector number and let *n* be the IRQ number. The vector number is
  - *x* = *n*+16 (i.e., 16=0+16)
- The IRQ defines the position in the vector table. The vector address is
  - 4**x* (i.e., 16*4=0x0000.0040) Each register size 32bits, 4byte
- For example SysTick is interrupt number 15 and IRQ number -1. Therefore the SysTick vector is at memory location 60, which is ROM location 0x0000003C (vector address)

Vector address	Number	IRQ	ISR name in Startup.s
0x00000038	14	-2	PendSV_Handler
0x0000003C	15	-1	SysTick_Handler

Nested Vectored Interrupt Controller (NVIC)

- Hardware unit that coordinates among interrupts from multiple sources
  1. Define **priority level** of each interrupt source (**`NVIC_PRIx_R`** registers)
  2. Separate **enable flag** for each interrupt source (**`NVIC_EN0_R`** and **`NVIC_EN1_R`**)

1.NVIC PRI Registers

- **High order three bits of each byte define priority**

Address	31 – 29	23 – 21	15 – 13	7 – 5	Name
0xE000E400	GPIO Port D	GPIO Port C	GPIO Port B	GPIO Port A	NVIC_PRI0_R
0xE000E404	SSI0, Rx Tx	UART1, Rx Tx	UART0, Rx Tx	GPIO Port E	NVIC_PRI1_R
0xE000E408	PWM Gen 1	PWM Gen 0	PWM Fault	I2C0	NVIC_PRI2_R
0xE000E40C	ADC Seq 1	ADC Seq 0	Quad Encoder	PWM Gen 2	NVIC_PRI3_R
0xE000E410	Timer 0A	Watchdog	ADC Seq 3	ADC Seq 2	NVIC_PRI4_R
0xE000E414	Timer 2A	Timer 1B	Timer 1A	Timer 0B	NVIC_PRI5_R
0xE000E418	Comp 2	Comp 1	Comp 0	Timer 2B	NVIC_PRI6_R
0xE000E41C	GPIO Port G	GPIO Port F	Flash Control	System Control	NVIC_PRI7_R
0xE000E420	Timer 3A	SSI1, Rx Tx	UART2, Rx Tx	GPIO Port H	NVIC_PRI8_R
0xE000E424	CAN0	Quad Encoder 1	I2C1	Timer 3B	NVIC_PRI9_R
0xE000E428	Hibernate	Ethernet	CAN2	CAN1	NVIC_PRI10_R
0xE000E42C	uDMA Error	uDMA Soft Tfr	PWM Gen 3	USB0	NVIC_PRI11_R
0xE000ED20	**SysTick**	PendSV	--	Debug	NVIC_SYS_PRI3_R

- For those interrupts with IRQ numbers (n) greater and or equal to 0, we can find its priority register bits by dividing n by 4.
  - Let m = n/4 (integer divide). The priority register number will be m. We can find the bit field for that IRQ by looking the remainder, p = n%4, where p = 0, 1, 2, or 3. The three bits will be (8*p)+7, (8*p)+6, and (8*p)+5.
- For example, Timer3A is IRQ number 35. n=35, so m=35/4 = 8. p=35%4=3.
  - Priority register is 8 (NVIC_PRI8_R)
  - Priority bits: (8*3)+7, (8*3)+6, and (8*3)+5, which are 31, 30, and 29.

Vector address	Number	IRQ	ISR name in Startup.s	NVIC	Priority bits
0x000000CC	51	35	Timer3A_Handler	NVIC_PRI8_R	31 – 29
0x000000B8	46	30	GPIOPortF Handler	NVIC PRI7 R	23 – 21

2.NVIC Enable Registers (simple)

- The 32 bits in register **NVIC_EN0_R** control the IRQ numbers 0 to 31 (exc #: 16 – 47).
  - UART0 is IRQ=5. To enable UART0 interrupts we set bit 5 in **NVIC_EN0_R**.
- The 32 bits in **NVIC_EN1_R** control the IRQ numbers 32 to 63 (exc #: 48 – 79).
  - UART2 is IRQ=33. To enable UART interrupts we set bit 1 (33-32=1) in **NVIC_EN1_R**.

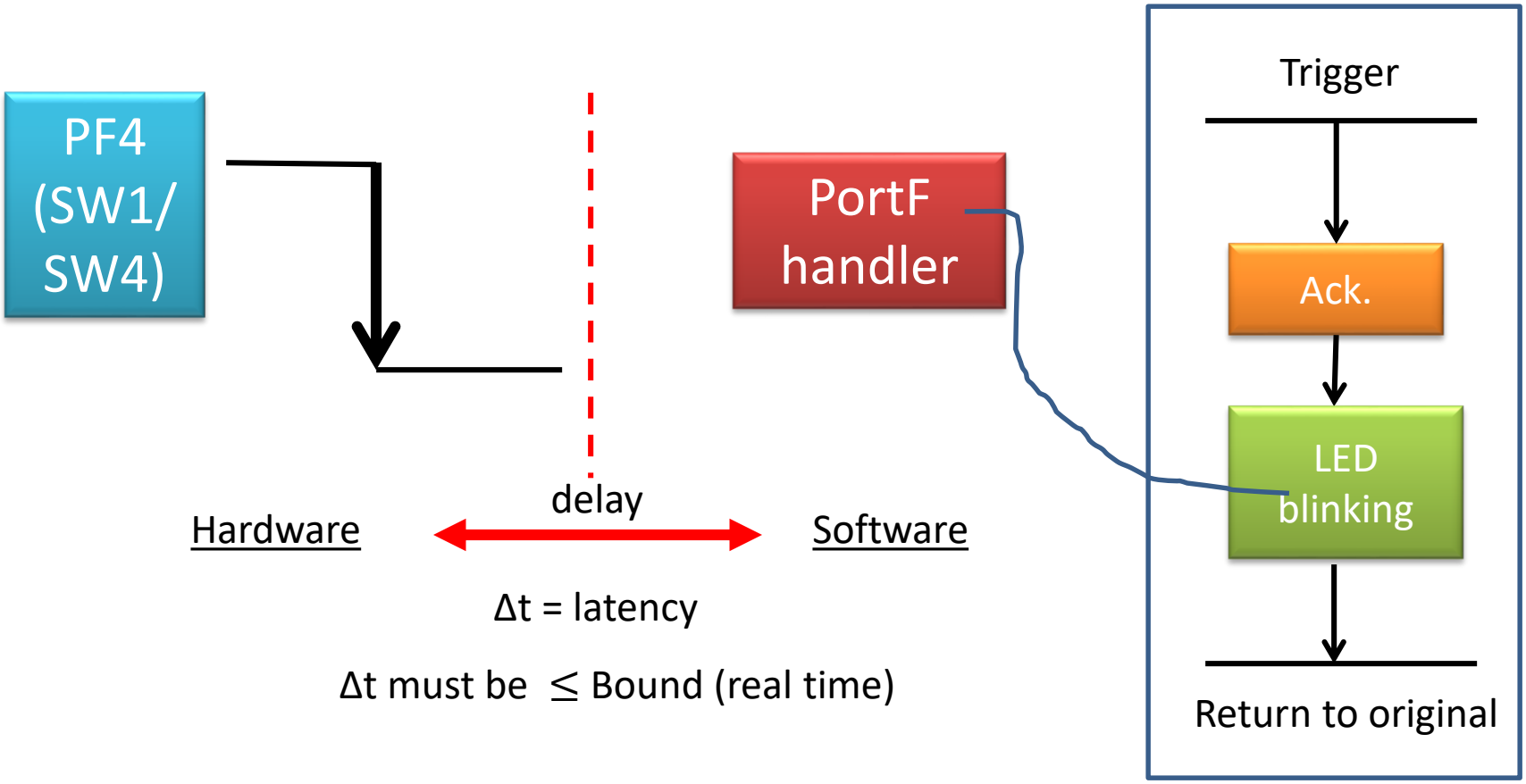
Summary

If you know a “IRQ”, you should be able to find

Exception number, Address, Priority register + bit fields, Enable register + bit field.

Let’s talk about interrupt examples

Processing interrupts in ARM Cortex-M4 (ex: Edge triggered interrupt)



```
// Green
#define GPIO_PF3_Data           (*((volatile unsigned long *)0x40025020))
// Blue
#define GPIO_PF1_Data           (*((volatile unsigned long *)0x40025008))
void DisableInterrupts(void); // Disable interrupts
void EnableInterrupts(void);  // Enable interrupts
void WaitForInterrupt(void);
/*
* Input:
* PF0 --> SW2
* PF4 --> SW1
* Output:
* PF1 --> Red
* PF3 --> Green
*/
```

Set up to configure Edge trigger

1. Setup hardware: I/O devices, i.e., switches
2. Configure software: GPIO pin configuration
3. Setup edge trigger details
4. Find a Vector handler
5. Set priority
6. Acknowledge (configure) local and global enable bits

1. Hardware: PF1/PF4 → onboard Switches
2. Software: input
   – AFSEL, PCTL, AMSEL: All disable

3. Edge trigger setup

(Arm=trigger caused by interrupt)
1: Allows interrupts that are generated by the corresponding pin

DIR (input)	IS (trigger type)	IBE (both edges)	IEV (edge type)	IM	Port mode (edg trigger mode)
	Interrupt **S**ense (1: level sen.)	**I**nterrupt **B**oth **E**dges	**I**nterrupt **EV**ent	**I**nterrupt **M**ask **E**nable bit (0: busy wait)	
0					Falling edge
0					Rising edge
0					Both edge

Pressing a switch: it would interrupt on the touch and the release

4. Vectors
   - Interrupt uses vectors (entry of interrupt)
   - Search for PortF handler

0x000000B8	46	30	GPIOPortF Handler	NVIC PRI7 R	23 – 21

if two things happen at the same time, which one goes first?

5. Priority

- Which is the most important
- If several ISRs: which ISR should be suspended
- Programmable priority range: 0-7

Vector address	Number	IRQ	ISR name in Startup.s	NVIC	Priority bits
0x00000038	14	-2	PendSV_Handler	NVIC_SYS_PRI3_R	23 – 21
0x0000003C	15	-1	SysTick_Handler	NVIC_SYS_PRI3_R	31 – 29
0x00000040	16	0	GPIOPortA_Handler	NVIC_PRI0_R	7 – 5
0x00000044	17	1	GPIOPortB_Handler	NVIC_PRI0_R	15 – 13
0x00000048	18	2	GPIOPortC_Handler	NVIC_PRI0_R	23 – 21
0x0000004C	19	3	GPIOPortD_Handler	NVIC_PRI0_R	31 – 29
0x00000050	20	4	GPIOPortE_Handler	NVIC_PRI1_R	7 – 5
0x00000054	21	5	UART0_Handler	NVIC_PRI1_R	15 – 13
0x00000058	22	6	UART1_Handler	NVIC_PRI1_R	23 – 21

6. Enable (two registers-local and global)

- Specific enable: NVIC_EN0_R or NVIC_EN1_R
  - Port F: set bit30 of NVIC_EN0_R

Address	31	30	29-7	6	5	4	3	2	1	0	Name
0xE000E100	G	F	…	UART1	UART0	E	D	C	B	A	NVIC_EN0_R
0xE000E104			…						UART2	H	NVIC_EN1_R

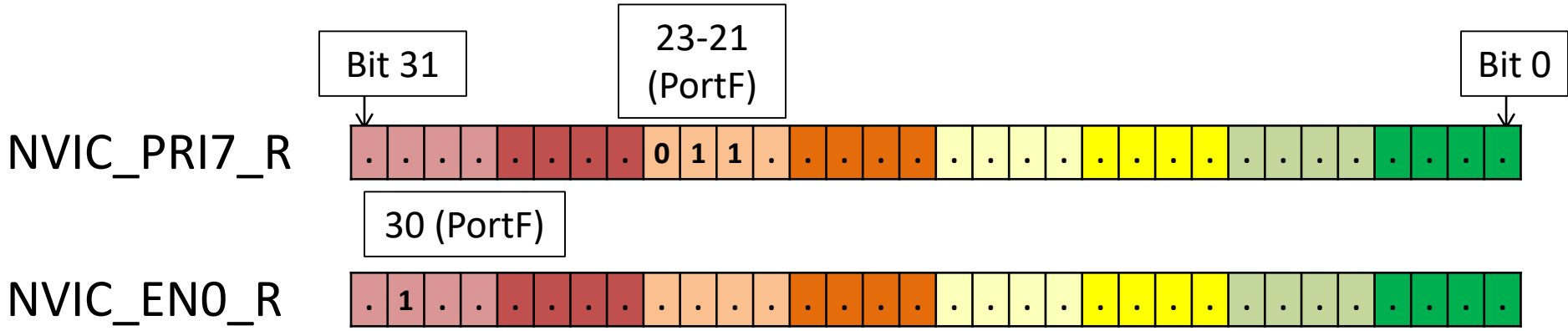
- Global enable: Primask register I bit
  - I=0 // to enable
  - I=1 // to disable

} EnableInterrupts();

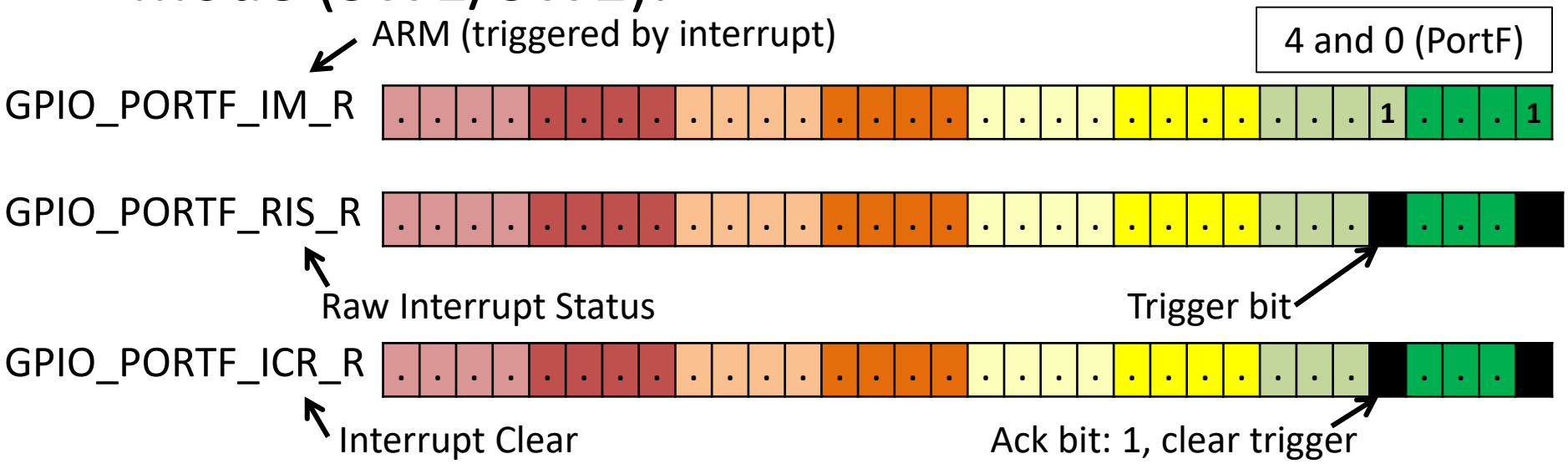
In ARM Cortex-M4 (via assembly instructions)
- **CPSIE I** (Change Processor State – Enable Interrupts)
- **CPSID I** (Change Process State – Disable Interrupts)

Or Keil µVision supports
- void __enable_irq(void) // CPSIE I;
- void __disable_irq(void) // CPSID I;

Port F edge trigger Interrupt



- Three registers in the port F edge triggered mode (SW1/SW2):



Edge triggered interrupt

```
// Green
#define GPIO_PF3_Data           (*((volatile unsigned long *)0x40025020))
// Blue
#define GPIO_PF1_Data           (*((volatile unsigned long *)0x40025008))
void DisableInterrupts(void); // Disable interrupts
void EnableInterrupts(void);  // Enable interrupts
void WaitForInterrupt(void);
/*
* Input:
* PF0 --> SW2
* PF4 --> SW1
* Output:
* PF1 --> Red
* PF3 --> Green
*/
```

```
void GPIOF_Handler(void) { //Find the handl
  unsigned int i, j, reading;
  // SW1(PF4) -> Green
    if(GPIO_PORTF_RIS_R & 0x10) {
```

```
int main(void) {
  // initialize PLL
  PLL_Init_50MHz();
  // initialize
  SysTick_Init();

  // initialize PortF
  PortF_Init();
  while(1) {
    __wfi();
  }
}
```

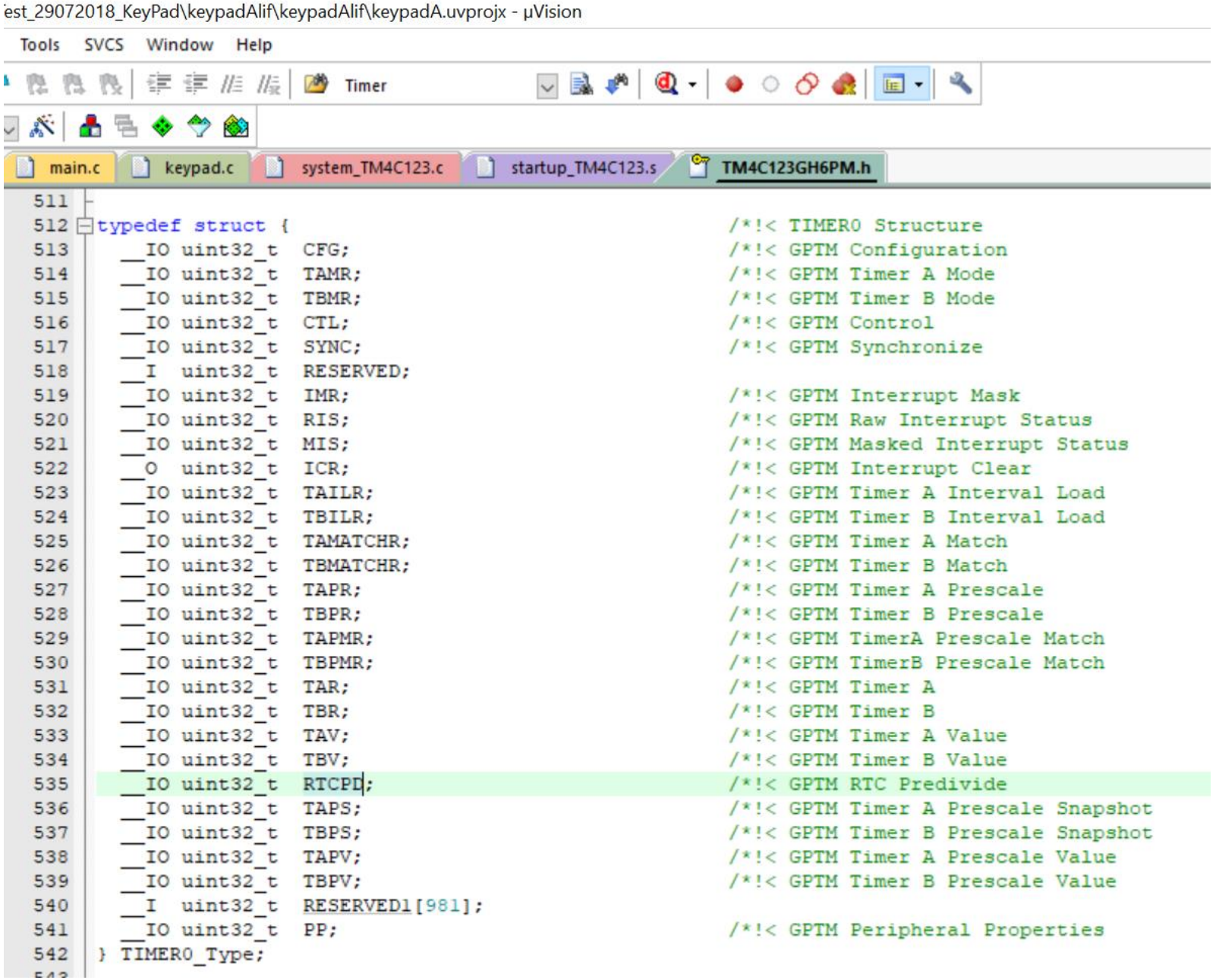
```
int main(void) {
  // initialize PLL
  PLL_Init_50MHz();
  // initialize
  SysTick_Init();
  // initialize PortF
  PortF_Init();
  while(1) {
    WaitForInterrupt();
  }
}
```

- The trigger flags are bit 0/4 of the **GPIO_PORTF_RIS_R** register
- This flag can be cleared by writing a 0x10/0x01 to **GPIO_PORTF_ICR_R**.
- Request an interrupt on the falling edge of PF4 or PF1 (when the user button is pressed)

https://www.allaboutcircuits.com/technical-articles/switch-bounce-how-to-deal-with-it/

Skip from here if we don't have enough time

Note that you can also use general "Timer"
as 16- or 32-bit programmable periodic timer
Refer to chapter **11.4.1 or page 722 datasheet**


est_29072018_KeyPad\keypadAlif\keypadAlif\keypadA.uvprojx - µVision
Tools SVCS Window Help
Timer
main.c keypad.c system_TM4C123.c startup_TM4C123.s TM4C123GH6PM.h
511
512 typedef struct { /*!< TIMER0 Structure
513 __IO uint32_t CFG; /*!< GPTM Configuration
514 __IO uint32_t TAMR; /*!< GPTM Timer A Mode
515 __IO uint32_t TBMR; /*!< GPTM Timer B Mode
516 __IO uint32_t CTL; /*!< GPTM Control
517 __IO uint32_t SYNC; /*!< GPTM Synchronize
518 __I uint32_t RESERVED;
519 __IO uint32_t IMR; /*!< GPTM Interrupt Mask
520 __IO uint32_t RIS; /*!< GPTM Raw Interrupt Status
521 __IO uint32_t MIS; /*!< GPTM Masked Interrupt Status
522 __O uint32_t ICR; /*!< GPTM Interrupt Clear
523 __IO uint32_t TAILR; /*!< GPTM Timer A Interval Load
524 __IO uint32_t TBILR; /*!< GPTM Timer B Interval Load
525 __IO uint32_t TAMATCHR; /*!< GPTM Timer A Match
526 __IO uint32_t TBMATCHR; /*!< GPTM Timer B Match
527 __IO uint32_t TAPR; /*!< GPTM Timer A Prescale
528 __IO uint32_t TBPR; /*!< GPTM Timer B Prescale
529 __IO uint32_t TAPMR; /*!< GPTM TimerA Prescale Match
530 __IO uint32_t TBPMR; /*!< GPTM TimerB Prescale Match
531 __IO uint32_t TAR; /*!< GPTM Timer A
532 __IO uint32_t TBR; /*!< GPTM Timer B
533 __IO uint32_t TAV; /*!< GPTM Timer A Value
534 __IO uint32_t TBV; /*!< GPTM Timer B Value
535 __IO uint32_t RTCPD; /*!< GPTM RTC Predivide
536 __IO uint32_t TAPS; /*!< GPTM Timer A Prescale Snapshot
537 __IO uint32_t TBPS; /*!< GPTM Timer B Prescale Snapshot
538 __IO uint32_t TAPV; /*!< GPTM Timer A Prescale Value
539 __IO uint32_t TBPV; /*!< GPTM Timer B Prescale Value
540 __I uint32_t RESERVED1[981];
541 __IO uint32_t PP; /*!< GPTM Peripheral Properties
542 } TIMER0_Type;

```
void timer0Init(unsigned long Hertz, BOOL irqEn ){
        TIMER0_Type * Timer;
        Timer = TIMER0;
        SYSCTL->RCGCTIMER |= 0x01;
/*1. Ensure the timer is disabled, TnEN GPTMCTL */
        Timer->CTL &= ~ 0x01;
/*2. write GPTM cofiguration register (GPTMCFG) with 0x00000000 */
        Timer->CFG = 0x00000000;
/*3. Configue the TnMr field in the GPTM timer n Mode Register GOTMTnMR */
/*3a or b 0x1 for One-Shot mode or 0x02 for Periodic mode */
        Timer->TAMR |= 0x2;
/*4. Optionally...*/
        Timer->TAMR |= (1<<4); //down counter
/*5. Load the start value into the GPTM Timer n Interval load register (GPTMTnILR) */
        Timer->TAILR = (SystemClock / Hertz)-1; //count value
/*6. If interrupts are required, set the bits in the GPTM interrupt mark register GPTMIMR */
        if(irqEn) Timer->IMR |= 0x01;
        else Timer->IMR &= ~0x01;
/*7. set TnEN bit in the GPTMCTL register to enable the timer and start counting */
/*8. Poll the GPTMRIS register or wait for the interrupt to generated(if enabled). In
   both cases, the status flags are cleared by writing a 1 to the appropriate bit
  of the GPTM Interrupt clear Register GPTMICR*/
        if(irqEn)               __NVIC_EnableIRQ(TIMER0A_IRQn);
        Timer->CTL |= 0x01;
}
```

```
void TIMER0A_Handler(void){
        TIMER0->ICR |= 0x01; //ack.
        //Do something
        }
```

Register 10: GPTM Timer A Interval Load (GPTMTAILR), offset 0x028

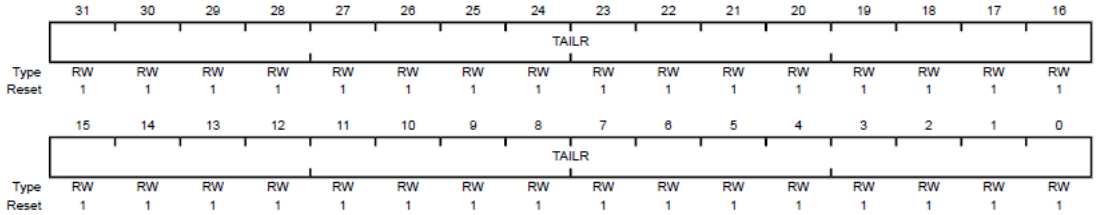
When the timer is counting down, this register is used to load the starting count value into the timer. When the timer is counting up, this register sets the upper bound for the timeout event.

When a 16/32-bit GPTM is configured to one of the 32-bit modes, **GPTMTAILR** appears as a 32-bit register (the upper 16-bits correspond to the contents of the **GPTM Timer B Interval Load (GPTMTBILR)** register). In a 16-bit mode, the upper 16 bits of this register read as 0s and have no effect on the state of **GPTMTBILR**.

When a 32/64-bit Wide GPTM is configured to one of the 64-bit modes, **GPTMTAILR** contains bits 31:0 of the 64-bit count and the **GPTM Timer B Interval Load (GPTMTBILR)** register contains bits 63:32.

GPTM Timer A Interval Load (GPTMTAILR)

16/32-bit Timer 0 base: 0x4003.0000
16/32-bit Timer 1 base: 0x4003.1000
16/32-bit Timer 2 base: 0x4003.2000
16/32-bit Timer 3 base: 0x4003.3000
16/32-bit Timer 4 base: 0x4003.4000
16/32-bit Timer 5 base: 0x4003.5000
32/64-bit Wide Timer 0 base: 0x4003.6000
32/64-bit Wide Timer 1 base: 0x4003.7000
32/64-bit Wide Timer 2 base: 0x4004.C000
32/64-bit Wide Timer 3 base: 0x4004.D000
32/64-bit Wide Timer 4 base: 0x4004.E000
32/64-bit Wide Timer 5 base: 0x4004.F000
Offset 0x028
Type RW, reset 0xFFFF.FFFF



Bit/Field	Name	Type	Reset	Description
31:0	TAILR	RW	0xFFFF.FFFF	GPTM Timer A Interval Load Register Writing this field loads the counter for Timer A. A read returns the current value of GPTMTAILR.

Periodic basis interrupts

- A computer to request an interrupt on a periodic basis
- An interrupt handler will be executed at fixed time intervals.
- Essential for the implementation of real-time data acquisition and real-time control systems.
  - Because software servicing must be performed at accurate time intervals.

For example

Implementing a digital controller that executes a control algorithm 100 times per a second

- Set up the internal timer hardware to request an interrupt every 10ms
- The ISR will execute the digital control algorithm and then return to the main thread periodically

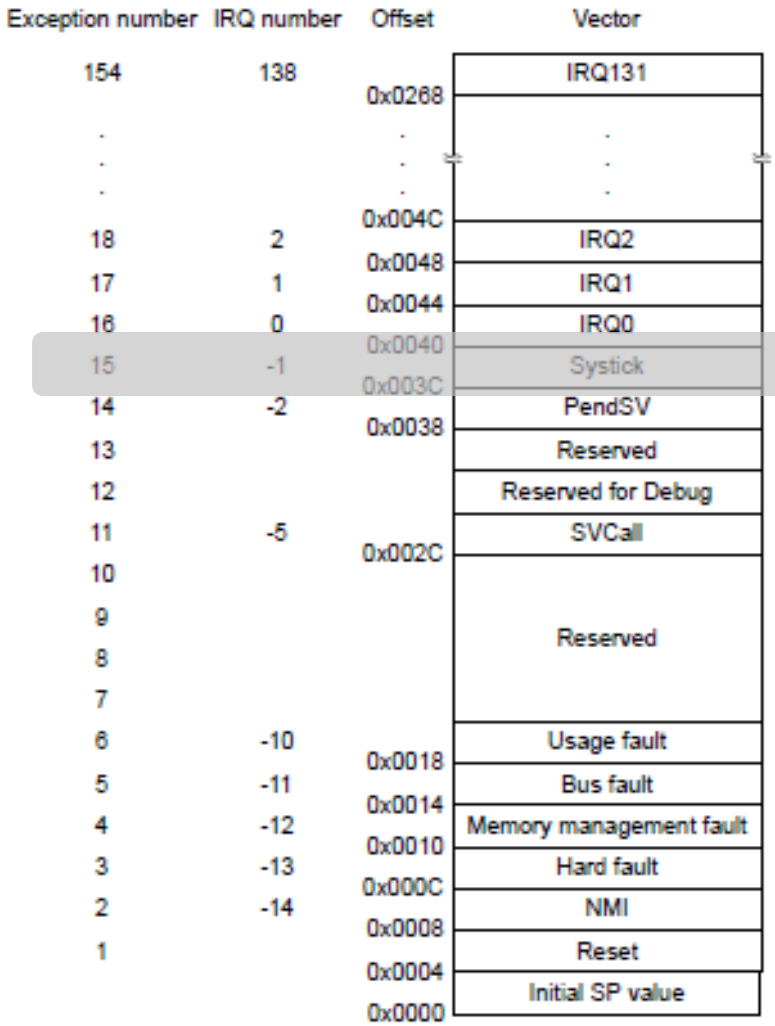
For example

Perform analog input and/or analog output: sample the ADC 100 times a second

- Set up the internal timer hardware to request an interrupt every 10 ms.
- The ISR will sample the ADC, process (or save) the data, and then return to the main thread periodically

107p

Figure 2-6. Vector Table



INTERRUPT VECTORS

Vector address	Exception Number	IRQ	ISR name in Startup.s	NVIC	Priority bits
0x00000038	14	-2	PendSV_Handler	NVIC_SYS_PRI3_R	23 – 21
0x0000003C	15	-1	SysTick_Handler	NVIC_SYS_PRI3_R	31 – 29
0x00000040	16	0	GPIOPortA_Handler	NVIC_PRI0_R	7 – 5
0x00000044	17	1	GPIOPortB_Handler	NVIC_PRI0_R	15 – 13
0x00000048	18	2	GPIOPortC_Handler	NVIC_PRI0_R	23 – 21
0x0000004C	19	3	GPIOPortD_Handler	NVIC_PRI0_R	31 – 29
0x00000050	20	4	GPIOPortE_Handler		
0x00000054	21	5	UART0_Handler		
0x00000058	22	6	UART1_Handler		
0x0000005C	23	7	SSI0_Handler		
0x00000060	24	8	I2C0_Handler		
0x00000064	25	9	PWMFault_Handler		
0x00000068	26	10	PWM0_Handler		
0x0000006C	27	11	PWM1_Handler		
0x00000070	28	12	PWM2_Handler		
0x00000074	29	13	Quadrature0_Handler		
0x00000078	30			NVIC_PRI3_R	23 – 21
0x0000007C	31			NVIC_PRI3_R	31 – 29
0x00000080	32			NVIC_PRI4_R	7 – 5
0x00000084	33			NVIC_PRI4_R	15 – 13
0x00000088	34			NVIC_PRI4_R	23 – 21
0x0000008C	35	19	Timer0A_Handler	NVIC_PRI4_R	31 – 29
0x00000090	36	20	Timer0B_Handler	NVIC_PRI5_R	7 – 5
0x00000094	37	21	Timer1A_Handler	NVIC_PRI5_R	15 – 13
0x00000098	38	22	Timer1B_Handler	NVIC_PRI5_R	23 – 21
0x0000009C	39	23	Timer2A_Handler	NVIC_PRI5_R	31 – 29
0x000000A0	40	24	Timer2B_Handler	NVIC_PRI6_R	7 – 5
0x000000A4	41	25	Comp0_Handler	NVIC_PRI6_R	15 – 13
0x000000A8	42	26	Comp1_Handler	NVIC_PRI6_R	23 – 21
0x000000AC	43	27	Comp2_Handler	NVIC_PRI6_R	31 – 29
0x000000B0	44	28	SysCtl_Handler	NVIC_PRI7_R	7 – 5
0x000000B4	45	29	FlashCtl_Handler	NVIC_PRI7_R	15 – 13
0x000000B8	46	30	GPIOPortF_Handler	NVIC_PRI7_R	23 – 21
0x000000BC	47	31	GPIOPortG_Handler	NVIC_PRI7_R	31 – 29
0x000000C0	48	32	GPIOPortH_Handler	NVIC_PRI8_R	7 – 5
0x000000C4	49	33	UART2_Handler	NVIC_PRI8_R	15 – 13
0x000000C8	50	34	SSI1_Handler	NVIC_PRI8_R	23 – 21
0x000000CC	51	35	Timer3A_Handler	NVIC_PRI8_R	31 – 29
0x000000D0	52	36	Timer3B_Handler	NVIC_PRI9_R	7 – 5
0x000000D4	53	37	I2C1_Handler	NVIC_PRI9_R	15 – 13
0x000000D8	54	38	Quadrature1_Handler	NVIC_PRI9_R	23 – 21
0x000000DC	55	39	CAN0_Handler	NVIC_PRI9_R	31 – 29
0x000000E0	56	40	CAN1_Handler	NVIC_PRI10_R	7 – 5
0x000000E4	57	41	CAN2_Handler	NVIC_PRI10_R	15 – 13
0x000000E8	58	42	Ethernet_Handler	NVIC_PRI10_R	23 – 21
0x000000EC	59	43	Hibernate_Handler	NVIC_PRI10_R	31 – 29
0x000000F0	60	44	USB0_Handler	NVIC_PRI11_R	7 – 5
0x000000F4	61	45	PWM3_Handler	NVIC_PRI11_R	15 – 13
0x000000F8	62	46	uDMA_Handler	NVIC_PRI11_R	23 – 21
0x000000FC	63	47	uDMA_Error	NVIC_PRI11_R	31 – 29

Systick interrupt handler and address

(Systick interrupt priority control)
Nested vector interrupt controller and specific bits to control (different from handler)

SysTick periodic interrupt

- A simple way to create periodic interrupts
- A periodic interrupt is one that is requested on a fixed time basis.
- SysTick is a 24-bit counter that decrements at the bus clock frequency.

The SysTick registers

- Used to create a periodic interrupt

Address	31-24	23-17	16	15-3	2	1	0	Name
0xE000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	**NVIC_ST_CTRL_R**
0xE000E014	0	24-bit RELOAD value						**NVIC_ST_RELOAD_R**
0xE000E018	0	24-bit CURRENT value of SysTick counter						**NVIC_ST_CURRENT_R**

Address	31-29	28-24	23-21	20-8	7-5	4-0	Name
0xE000ED20	TICK	0	PENDSV	0	DEBUG	0	**NVIC_SYS_PRI3_R**

NVIC Registers

- **High order three bits of each byte define priority**

Address	31 – 29	23 – 21	15 – 13	7 – 5	Name
0xE000E400	GPIO Port D	GPIO Port C	GPIO Port B	GPIO Port A	NVIC_PRI0_R
0xE000E404	SSI0, Rx Tx	UART1, Rx Tx	UART0, Rx Tx	GPIO Port E	NVIC_PRI1_R
0xE000E408	PWM Gen 1	PWM Gen 0	PWM Fault	I2C0	NVIC_PRI2_R
0xE000E40C	ADC Seq 1	ADC Seq 0	Quad Encoder	PWM Gen 2	NVIC_PRI3_R
0xE000E410	Timer 0A	Watchdog	ADC Seq 3	ADC Seq 2	NVIC_PRI4_R
0xE000E414	Timer 2A	Timer 1B	Timer 1A	Timer 0B	NVIC_PRI5_R
0xE000E418	Comp 2	Comp 1	Comp 0	Timer 2B	NVIC_PRI6_R
0xE000E41C	GPIO Port G	GPIO Port F	Flash Control	System Control	NVIC_PRI7_R
0xE000E420	Timer 3A	SSI1, Rx Tx	UART2, Rx Tx	GPIO Port H	NVIC_PRI8_R
0xE000E424	CAN0	Quad Encoder 1	I2C1	Timer 3B	NVIC_PRI9_R
0xE000E428	Hibernate	Ethernet	CAN2	CAN1	NVIC_PRI10_R
0xE000E42C	uDMA Error	uDMA Soft Tfr	PWM Gen 3	USB0	NVIC_PRI11_R
0xE000ED20	**SysTick**	PendSV	--	Debug	NVIC_SYS_PRI3_R

SysTick Interrupt procedure

- f_{BUS} : the frequency of the bus clock
- n : the value of the **RELOAD** register
- The frequency of the periodic interrupt : $f_{BUS}/(n+1)$.

1) Clear the **ENABLE** bit to turn off SysTick during initialization.
2) Set the **RELOAD** register with the value n.
3) Write any value to **NVIC_ST_CURRENT_R** to clear the counter.

4) Write the desired mode to the control register, **NVIC_ST_CTRL_R**.
   - We must set **CLK_SRC**=1, because **CLK_SRC**=0 external clock mode is not implemented on the LM3S/TM4C family.
   - We need to set the **ENABLE**bit so the counter will run.
   - We set **INTEN** to enable interrupts.
5) We establish the priority of the SysTick interrupts using the TICK field in the **NVIC_SYS_PRI3_R** register.

- The **CURRENT** value counts down from 1 to 0: the **COUNT** flag is set.
- On the next clock, the **CURRENT** is loaded with the **RELOAD** value.
- Rolls over every *n*+1 counts: the **COUNT** flag will be set every *n*+1 counts.

```
Sequence:
n,n-1,n-2,n-3… 2,1,0,n,n-1…
```

Set a flag

Set with Reload value n

Recall: SysTick Timer Init examples

```
#define NVIC_ST_CTRL_R(*((volatile uint32_t *)0xE000E010))
#define NVIC_ST_RELOAD_R(*((volatile uint32_t *)0xE000E014))
#define NVIC_ST_CURRENT_R(*((volatile uint32_t *)0xE000E018))
```

- General case

```
void SysTick_Init(void){
 NVIC_ST_CTRL_R = 0; // 1) disable SysTick during setup
 NVIC_ST_RELOAD_R = 0x00FFFFFF; // 2) maximum reload value
 NVIC_ST_CURRENT_R = 0; // 3) any write to CURRENT clears it
 NVIC_ST_CTRL_R = 0x05; // 4) enable SysTick with core clock 0101
}
```

- Interrupt available

```
void SysTick_Init(uint32_t period){
  Counts = 0;
  NVIC_ST_CTRL_R = 0;              // disable SysTick during setup
  NVIC_ST_RELOAD_R = period-1;     // reload value
  NVIC_ST_CURRENT_R = 0;           // any write to current clears it
  NVIC_SYS_PRI3_R = (NVIC_SYS_PRI3_R&0x00FFFFFF)|0x40000000; // priority 2
  NVIC_ST_CTRL_R = 0x07;           // enable, core bus clock, arm 0111
}
```

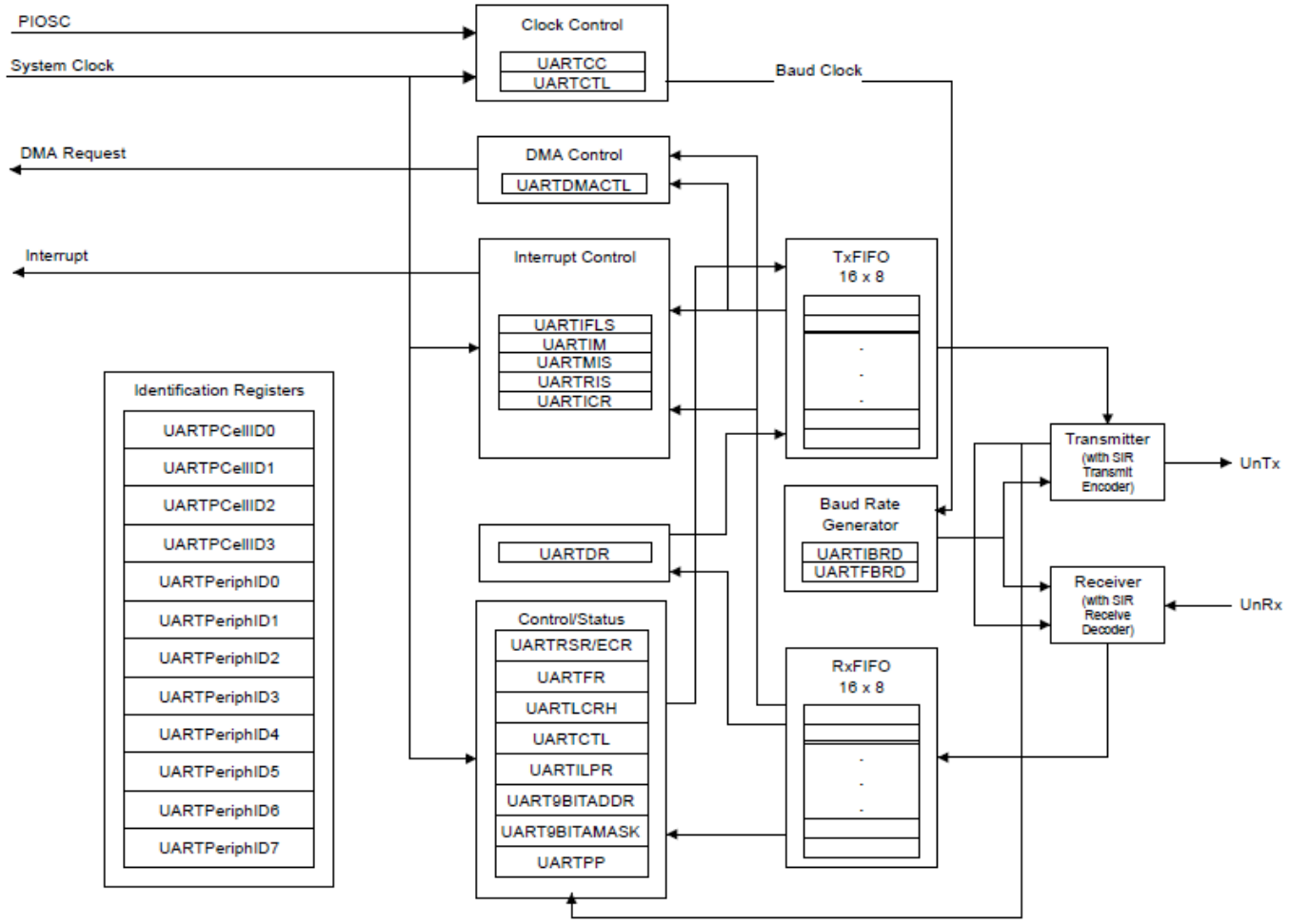
An important notice

- SysTick is the only interrupt on the TM4C that has an automatic acknowledge.
- Notice there is no explicit software step in the ISR to clear the **COUNT** flag.

UART interrupt (pp.894)

14.1 Block Diagram

Figure 14-1. UART Module Block Diagram



Register 9: UART Interrupt FIFO Level Select (UARTIFLS), offset 0x034

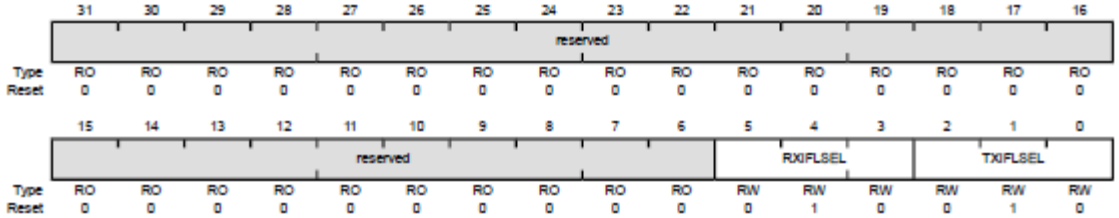
The **UARTIFLS** register is the interrupt FIFO level select register. You can use this register to define the FIFO level at which the `TXRIS` and `RXRIS` bits in the **UARTRIS** register are triggered.

The interrupts are generated based on a transition through a level rather than being based on the level. That is, the interrupts are generated when the fill level progresses through the trigger level. For example, if the receive trigger level is set to the half-way mark, the interrupt is triggered as the module is receiving the 9th character.

Out of reset, the `TXIFLSEL` and `RXIFLSEL` bits are configured so that the FIFOs trigger an interrupt at the half-way mark.

UART Interrupt FIFO Level Select (UARTIFLS)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x034
Type RW, reset 0x0000.0012



Bit/Field	Name	Type	Reset	Description
31:6	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
5:3	RXIFLSEL	RW	0x2	UART Receive Interrupt FIFO Level Select The trigger points for the receive interrupt are as follows: Value Description 0x0 RX FIFO ≥ ⅛ full 0x1 RX FIFO ≥ ¼ full 0x2 RX FIFO ≥ ½ full (default) 0x3 RX FIFO ≥ ¾ full 0x4 RX FIFO ≥ ⅞ full 0x5-0x7 Reserved

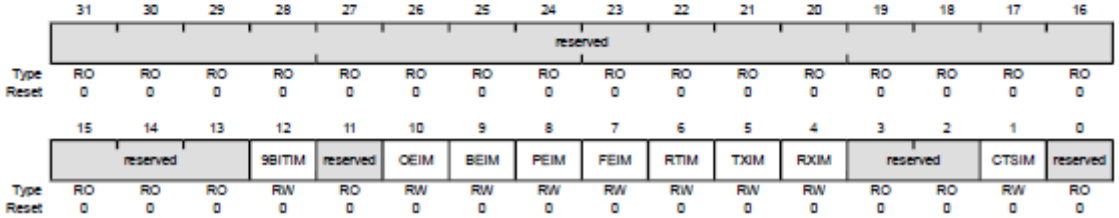
Register 10: UART Interrupt Mask (UARTIM), offset 0x038

The **UARTIM** register is the interrupt mask set/clear register.

On a read, this register gives the current value of the mask on the relevant interrupt. Setting a bit allows the corresponding raw interrupt signal to be routed to the interrupt controller. Clearing a bit prevents the raw interrupt signal from being sent to the interrupt controller.

UART Interrupt Mask (UARTIM)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x038
Type RW, reset 0x0000.0000



5	TXIM	RW	0	UART Transmit Interrupt Mask

Value	Description
0	The TXRIS interrupt is suppressed and not sent to the interrupt controller.
1	An interrupt is sent to the interrupt controller when the TXRIS bit in the **UARTRIS** register is set.

4	RXIM	RW	0	UART Receive Interrupt Mask

Value	Description
0	The RXRIS interrupt is suppressed and not sent to the interrupt controller.
1	An interrupt is sent to the interrupt controller when the RXRIS bit in the **UARTRIS** register is set.

Reading

Vol.1	Vol.2
Ch.9 (9.4, 9.5, 9.6, 9.7)	Ch.3 (3.7) Ch.4 (4.5) Ch.5 (5.4, 5.5, 5.7)