Interrupt Processing I

ENCE361 Embedded Systems 1

Course Coordinator: Ciaran Moore (ciaran.moore@Canterbury.ac.nz)

Lecturer: Le Yang (le.yang@canterbury.ac.nz)

Department of Electrical and Computer Engineering

Where we're going today

- **Introduction to interrupts**
- Context, interrupt service routine (ISR) and interrupt vector
- Example code
- Homework

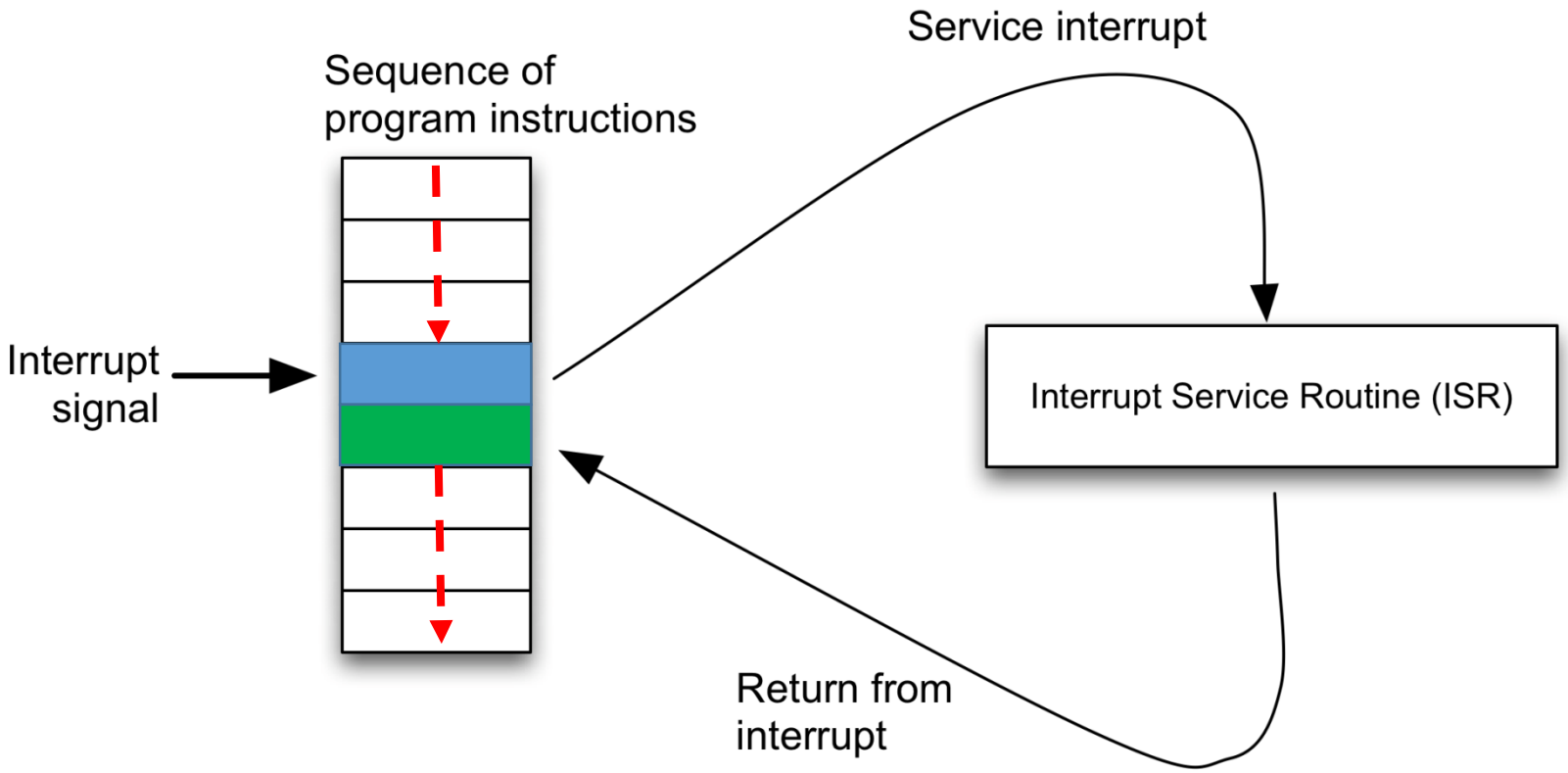
Introduction to Interrupts (1)

- Interrupts are events requiring attentions of the microcontroller
  - Most events are from peripherals
    - ADC conversion is complete
    - Time out of timer
    - System tick (SysTick) …
  - Most microcontrollers support a number of interrupts

  - Microcontroller stops the execution of the current program (*briefly*), after an interrupt request, to run a (*short*) interrupt service routine (ISR)
    - Indicates that ISR has higher priority
  - Microcontroller returns to where it was stopped

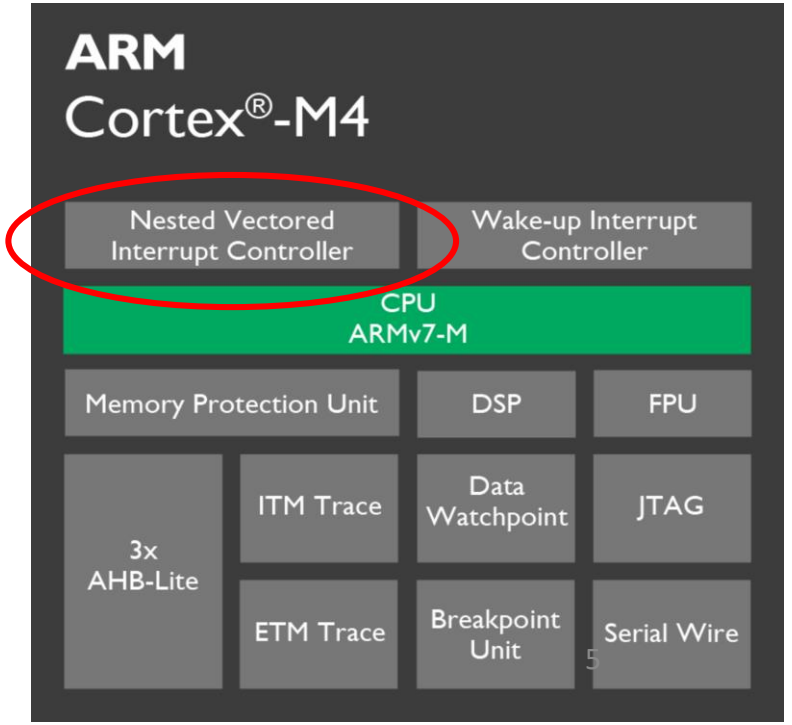
Introduction to Interrupts (2)

- How the microcontroller handles an interrupt request: schematic overview
  - ISR is executed between two neighboring instructions



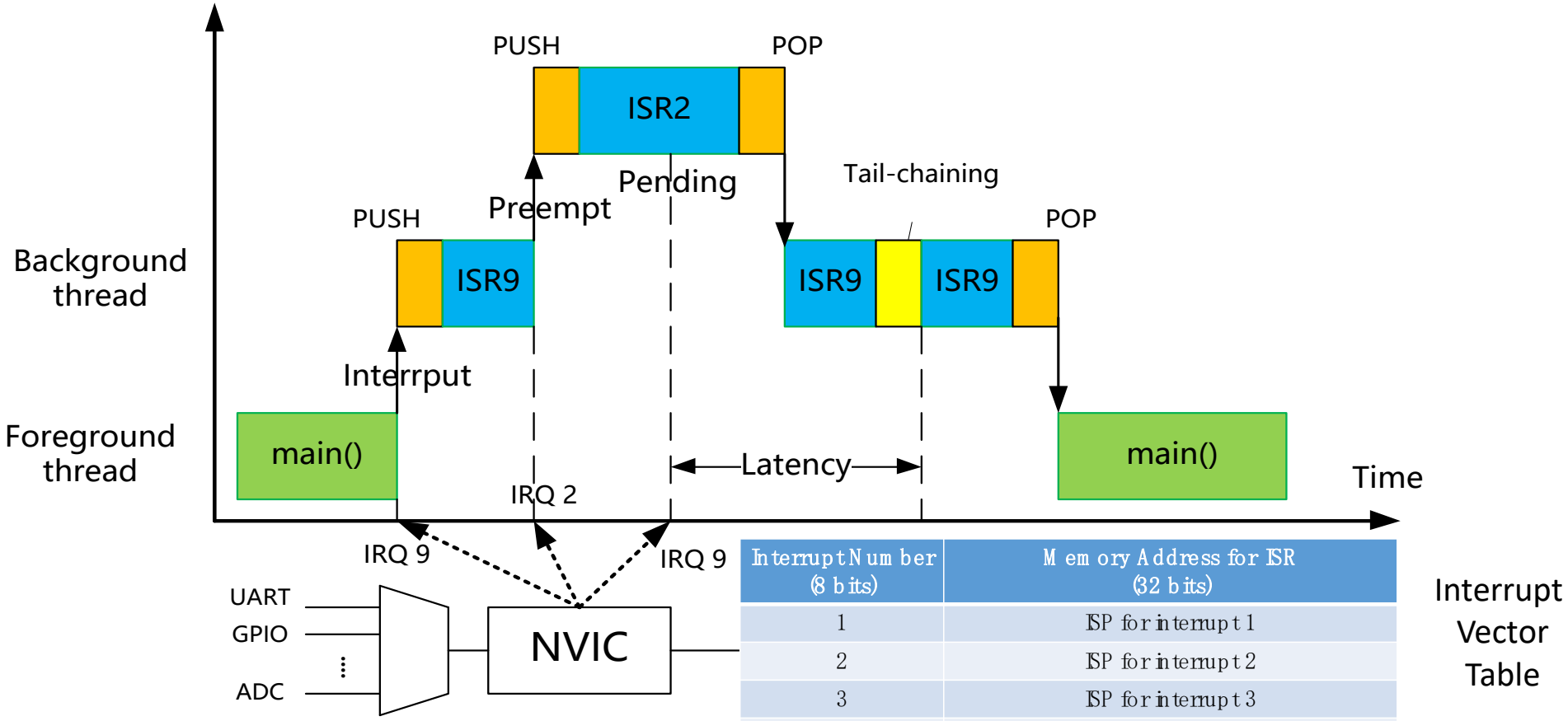
Priority (1)

- Priority represents the relative importance of an event
- Interrupts may have different priority levels
  - Some interrupts such as reset have fixed (highest) priority
  - Others have programmable priority
    - Interrupts can be preempted (interrupted)
  - Interrupts can be enabled/disabled
- Nested Vectored Interrupt Controller (NIVC)
  - Prioritizes and handles all interrupts
  - Assign preempt priority number for preemption
  - Assign sub-priority number for ordering interrupts with same group preempt priority (see more in lecture 14)



Priority (2)

- How the microcontroller handles multiple interrupts with different priority



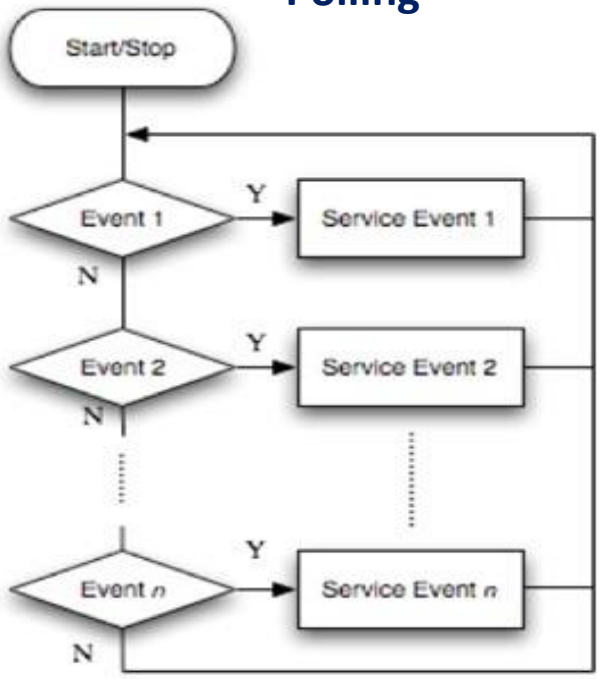
Interrupt Number (8 bits)	Memory Address for ISR (32 bits)
1	ISP for interrupt 1
2	ISP for interrupt 2
3	ISP for interrupt 3
...	...

Interrupt Vector Table

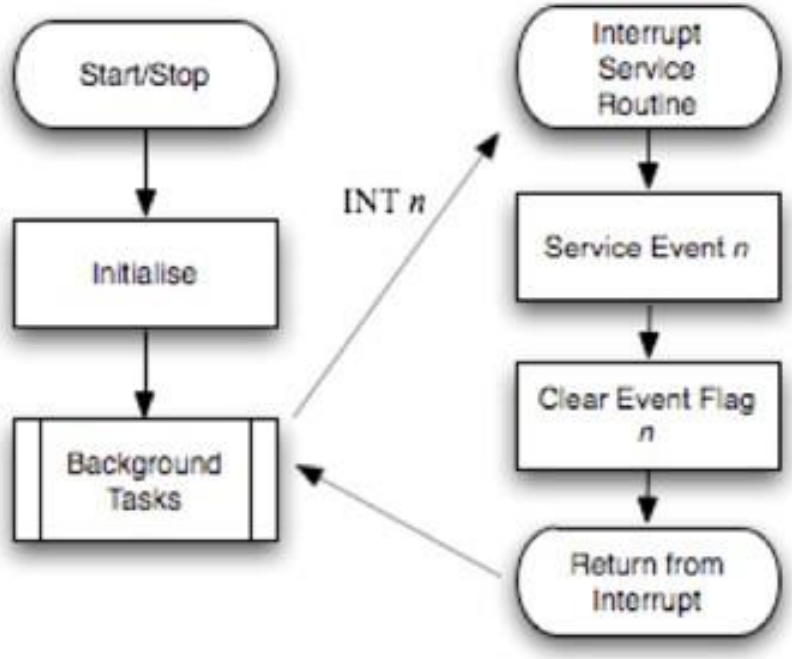
Polling vs. Interrupts (1)

- Polling: check repeatedly whether there are peripherals needing services
  - Efficient in responding to regular/synchronous events
  - Waste of microcontroller clock cycles when e.g., I/O peripherals are slow
  - Difficult to support real-time application with many devices to check

Polling



Interrupts



Polling vs. Interrupts (2)

- Interrupts: microcontroller runs background program normally and stops, when needed, to respond to peripheral events
  - More efficient than polling in handling asynchronous events

- Scenarios where the use of interrupts should be considered
  - Events are asynchronous and their service needs to be completed within a short time
  - Events occur at a high average repetition rate or at varying intervals

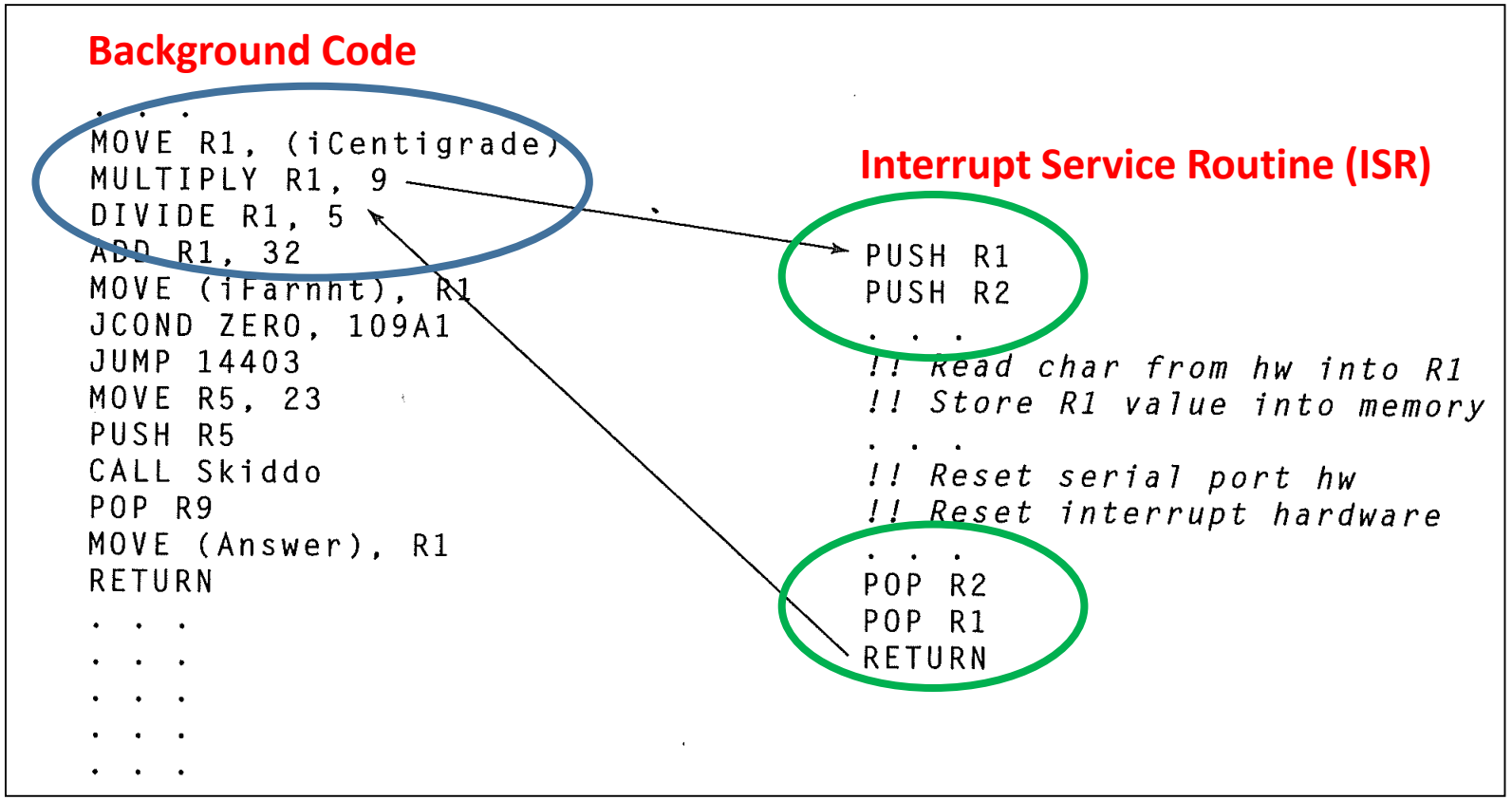
  - Microcontroller is relatively busy with the task it must perform
  - A foreground/background model (more in Lecture 12) can simplify the control code for a real-time system

Where we’re going today

- Introduction to interrupts
- **Context, interrupt service routine (ISR) and interrupt vector**
- Example code
- Homework

Context (1)

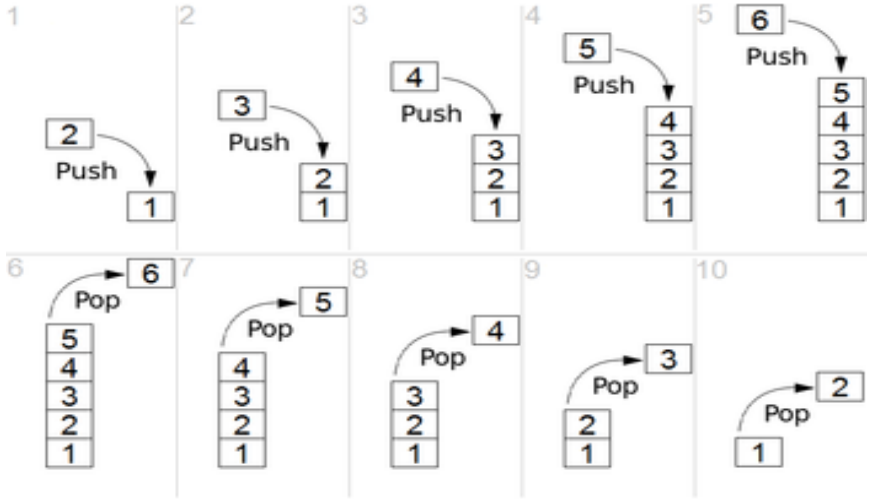
- Important to save context before executing ISR to respond to an interrupt



From Page 87 in D. E. Simon's book "An Embedded Software Primer"

Context (2)

- ARM Cortex-M4 has a stack-based exception (interrupt) model
  - Stack is Last-In First-Out (LIFO)
  - Context is pushed before executing ISR: (PUSH)
  - Context is restored after executing ISR: (POP)
- Stack operations handled by hardware in ARM Cortex-M4
  - No need to write assembly code for pushing and restoring
    - Program counter (PC)
    - Program status register (PSR)： N, Z, C, V flags
    - Link Register (LR)
    - General purpose registers R0-R3, R12
  - Extra assembly codes are needed if protection of extra context is required



Anatomy of Interrupt Service Routine (ISR)

- Prologue
  - PUSH to stack the context not automatically saved but possibly affected in ISR
  - If pre-emption by higher priority interrupts is allowed, globally enable interrupts
- Service (more in future lectures)
  - Perform essential task related to interrupt (e.g. read ADC output to a buffer)
  - If necessary, clear the state of interrupting hardware.
  - If pre-emption by higher priority interrupts is allowed, globally disable interrupts
- Epilogue
  - POP the part of the context saved on entry
  - Return from ISR to the background program

Interrupt Vector (1)

- Interrupt vector = starting address of ISR for an interrupt
  - Usually stored in a table (vector table) indexed by interrupt number (IRQx)

Table 2-8. Exception Types

Exception Type
-
Reset
Non-Maskable Interrupt (NMI)
Hard Fault
Memory Management

Table 2-9. Interrupts

Vector Number	Interrupt Number (Bit in Interrupt Registers)	Vector Address or Offset	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

From Page 103-107, TM4C123GH6PM Data Sheet.pdf

Interrupt Vector (2)

- Register an interrupt before enabling it
  - Put the ISR address in the interrupt vector table
- How to register an interrupt
  1. Call the interrupt controller API function **IntRegister()**

```
#include"inc/hw_ints.h"      // FAULT_SYSTICK specifies the offset of SysTick in vector table
IntRegister(FAULT_SYSTICK, SysTickIntHandler);     // SysTickIntHandler is the ISR
```

  2. Call the API function associated with the specific peripheral

```
SysTickIntRegister(SysTickIntHandler);                    // SysTickIntHandler is the ISR
```

- More in TivaWare Peripheral Driver Library Users Manual.pdf

Summary of Servicing an Interrupt

- Complete the instruction being currently executed
- PUSH some part of the context automatically to the stack
- Save other parts of the context via user-defined code on the stack
- Load registered interrupt vector into the program counter to run ISR
- Terminate ISR with a return from interrupt instruction
- Restore the parts of the context saved via user-defined code
- POP the remaining context automatically from the stack
- Resume execution of the interrupted code

Where we're going today

- Introduction to interrupts
- Context, interrupt service routine (ISR) and interrupt vector
- **Example code**
- Homework

Example code: ADCdemo1.c (1)

```
// Interrupt handler for the SysTick interrupt
void SysTickIntHandler(void)
{
   // Initiate a conversion
   ADCProcessorTrigger(ADC0_BASE, 3);
   g_ulSampCnt++;
}
// The handler for the ADC conversion complete interrupt
void ADCIntHandler(void)
{
        uint32_t ulValue;
        // Get the single sample from ADC0
        ADCSequenceDataGet(ADC0_BASE, 3, &ulValue);

        // Place it in the circular buffer (advancing write index)
        writeCircBuf (&g_inBuffer, ulValue);

        // Clean up, clearing the interrupt
        ADCIntClear(ADC0_BASE, 3);
}
```

Example code: ADCdemo1.c (2)

```
// *********************************************************************
// Initialisation function for the clock
// *********************************************************************
void initClock (void)
{
    // Set the clock rate to 20 MHz
    SysCtlClockSet(SYSCTL_SYSDIV_10 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);

    // Set up the period for the SysTick timer
     SysTickPeriodSet(SysCtlClockGet() / SAMPLE_RATE_HZ);
    //
    // Register the interrupt handler
    SysTickIntRegister(SysTickIntHandler);
    //
    // Enable interrupt and device
    SysTickIntEnable();
    SysTickEnable();
}
```

Example code: ADCdemo1.c (3)

```
 void initADC (void)
{
   SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
   ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);
   ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_CH0 | ADC_CTL_IE | ADC_CTL_END);
   ADCSequenceEnable(ADC0_BASE, 3);
   ADCIntRegister (ADC0_BASE, 3, ADCIntHandler);
   ADCIntEnable(ADC0_BASE, 3);
}

int main(void)
{
   initClock ();
   initADC ();
   initDisplay ();
   initCircBuf (&g_inBuffer, BUF_SIZE);

   IntMasterEnable(); // Enable interrupts to the processor.
   . . . .
}
```

Homework

1. Can the ADCdemo1.c program achieve a comparable performance without using interrupts? If so, how? If not, why not?

2. By looking within the microprocessor datasheet **TM4C123GH6PM Data Sheet.pdf**, find out how many sorts of interrupts can be generated by the ADC and Timer modules. List all the types (assuming all timers are the same and ADC channels are the same).

3. In general, how can a compiler decide which of the registers need to have their contents saved as part of the context? Refer to Slides 10 and 11.