**Lab 12 Notes: Interrupts**

**Key Terms (Cortex M based)**

* **Atomic operation:** is a process which cannot be interrupted once started.
* **Trigger**: A hardware event (the automatic transfer of software execution to a hardware event; an interrupt)
* **Process**: is the action of software as it executes
* **Thread**: The path of action a software takes as it is executed
  + **Background thread**: The execution of the interrupt service routine
  + **Foreground thread**: The execution of programs in the main() routine
  + **Multi-threaded**: foreground and background threads cooperating to perform an overall

task.

* + **Operating system:** Allows multiple foreground threads
* **Thread mode:** Main program
* **Handler mode:** ISR
* **EnableInterupts():** Allows interrupts to happen at this time
  + PRIMASK = 0
* **DisableInterupts():** Postpone interrupts until a later time.
  + PRIMASK = 1
* **PRIMASK:** Interrupt Priority Mask
* **BASEPRI**: Is a register for setting interrupt priority.
  + If BASEPRI is set to 3, requests level 0,1, and 2 can interrupt. Anything above 3 will be postponed. If BASEPRI is set to 0 then nothing can interrupt the current interrupt.
* **Context switch:** The steps an interrupt process routine goes through switching from **foreground** thread to a **background** thread, or when going from lower priority thread to higher priority thread.
* **Stack:** Is a specialized buffer which stores data from the top, down. As new data comes in, they push down the older data with the top being the most recent.
* **SP:** (Stack Pointer) is a register that stores the address of the last program request in a stack.
  + **MSP** (Main Stack Pointer) is the default stack pointer used by exception handlers
  + **PSP**  (Process Stack Pointer) used by user application code
* **LR:** (Link Register) holds the address to return to when a function call completes.
* **PC:** (Program Counter), also known as the Instruction Pointer(IP), is incremented after fetching an instruction. The program counter holds the memory address of the next instruction.
* **IPSR:** (Interrupt program status register) contains the exception type number of the current ISR.
* **ISR:** (Interrupt service routine), also known as an interrupt handler is a callback function who’s execution is triggered when the hardware requests an interrupt.
  + **Polled Interrupts:** One large ISR that handles all requests.
  + **Vectored Interrupts:** Many small ISRs, each with specific source of interrupts.
* **Tail chaining:** occurs when one ISR executes immediately after another
* **Nested Interrupts:** occurs when a higher priority interrupt suspends an ISR.
* **RIS** (Raw Interrupt Status) aka the interrupt trigger is set by a hardware event, and the software (acknowledgement) clears the flag.
* IS (Interrupt Sense)
* IEV (Interrupt event)
* IME (Interrupt mask enable)

**Conditions for Interrupts to be executed:**

1. **Device is armed** – Allow the hardware trigger to interrupt.
2. **NVIC is enabled for the device** – Set the NVIC enable bit under 0xE000.E100 NVIC register.
3. **EnableInterupts()** – the function will set PRIMASK = 0.
4. **Has priority** – BASEPRI is set higher than the current running interrupt.
5. **Trigger** – The external hardware event that will trigger the interrupt becomes active/true

**Interrupt process routine (the context switch):**

1. Finish processing the current instruction when an interrupt trigger occurs
2. **Push registers** R0, R1, R2, R3, R12, R13 **(Stack Pointer)**, R14 **(Link Register)**, and R15 **(Program Counter)** to the **stack** (a special region in memory that stores temporary variables created by functions() --- including main() function).
3. Link register (LR) is set to **0xFFFF.FFF9** (this will return the control to the main program when the function call completes).

0xE1 Return to Handler mode MSP (using floating point state on TM4C)  
  0xE9 Return to Thread mode MSP (using floating point state on TM4C)  
  0xED Return to Thread mode PSP (using floating point state on TM4C)  
  0xF1 Return to Handler mode MSP  
 **0xF9 Return to Thread mode MSP** ← in this class we will always be using this one  
  0xFD Return to Thread mode PSP

1. Interrupt Program Status Register (IPSR) is set to the interrupt number being processed.
2. Program Counter (PC) is loaded with the interrupt vector (address of the ISR vector).

**DO’s and DON’Ts of ISR (Interrupt Service Routine)**

* **DO** acknowledge (clear) the trigger flag
* **DO** make the ISR’s as short as possible
* **DON’T** have delay/busy wait loops in the ISR

**Common Errors:**

* The system will crash if the ISR doesn’t acknowledge or disarm the device requesting the interrupt.
* The ISR software should not disable

**When to use Interrupts and busy-wait**

* **Busy-wait:** 
  + When I/O timing is predictable
  + I/O structure is simple and fixed
* **Interrupts:**
  + When I/O timing is variable
  + When I/O is complex
  + I/O devices exist in the system with different speeds
  + Infrequent but critical events:
    - Power failure
    - Memory faults
    - Machine error
  + Periodic interrupts:
    - Real-time clocks
    - Data Acquisition systems
    - Control systems

**Synchronizing threads**

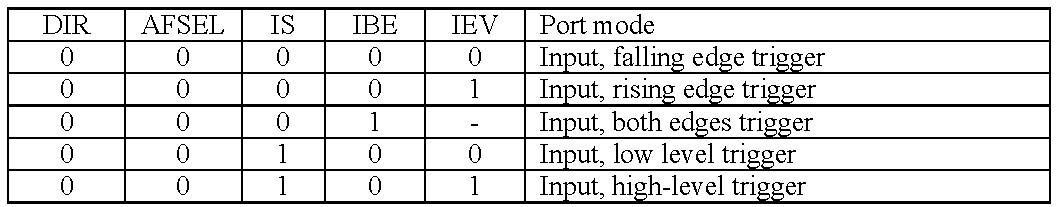
* **Binary semaphore:** uses 1 shared variable with two operations
  + **Signal:** sets the flag
  + **Wait:** check if the flag is set or not
    - **If set:** clear the flag, perform task
    - **If not set:** do nothing
  + The flag must exist as a private-global variable (using the “**static**” qualifier).
  + You must limit the access to this flag to as few modules as possible to avoid bugs.
* **The Mailbox:** uses 2 shared variables (Mail, Status)
  + **Mail:** contains 1 data
  + **Status:** contains a binary semaphore flag
* **FIFO Queue:** uses 2 shared variables AND allows buffering
  + Similar to the mailbox sync
  + Can process multiple data improving the overall performance of a complex system

**Nested Vectored Interrupt Controller (NVIC) Key points**

* Interrupts on Cortex-M are controlled by NVIC
  + Has a 32-bit vector that points to the memory location of the ISR handler.
    - The vector is stored on ROM at the beginning of memory
* ISR’s can be attached to interrupts by writing the ISR as regular C functions
* Vectors are defined at compile time since they are stored in ROM
* Each ISR must acknowledge the interrupt in software by clearing the flag that caused the interrupt:  GPIO\_PORTF\_ICR\_R = 0x10;
  + See GPIOICR register description
* Only the top 3 bits of the 8bits are used to specify interrupt priority level 0-7
  + 0 – highest priority
* NVIC Enable Register (EN0 – EN3) has the following effect:
  + Writing a 1 to the corresponding IRQ number of the interrupt will enable the NVIC
    - Writing a 0 to the same corresponding IRQ bit position above will have no effect
  + To disable interrupts, we need to use the (DIS0 – DIS3) register
* Systick interrupts do not need to be enabled

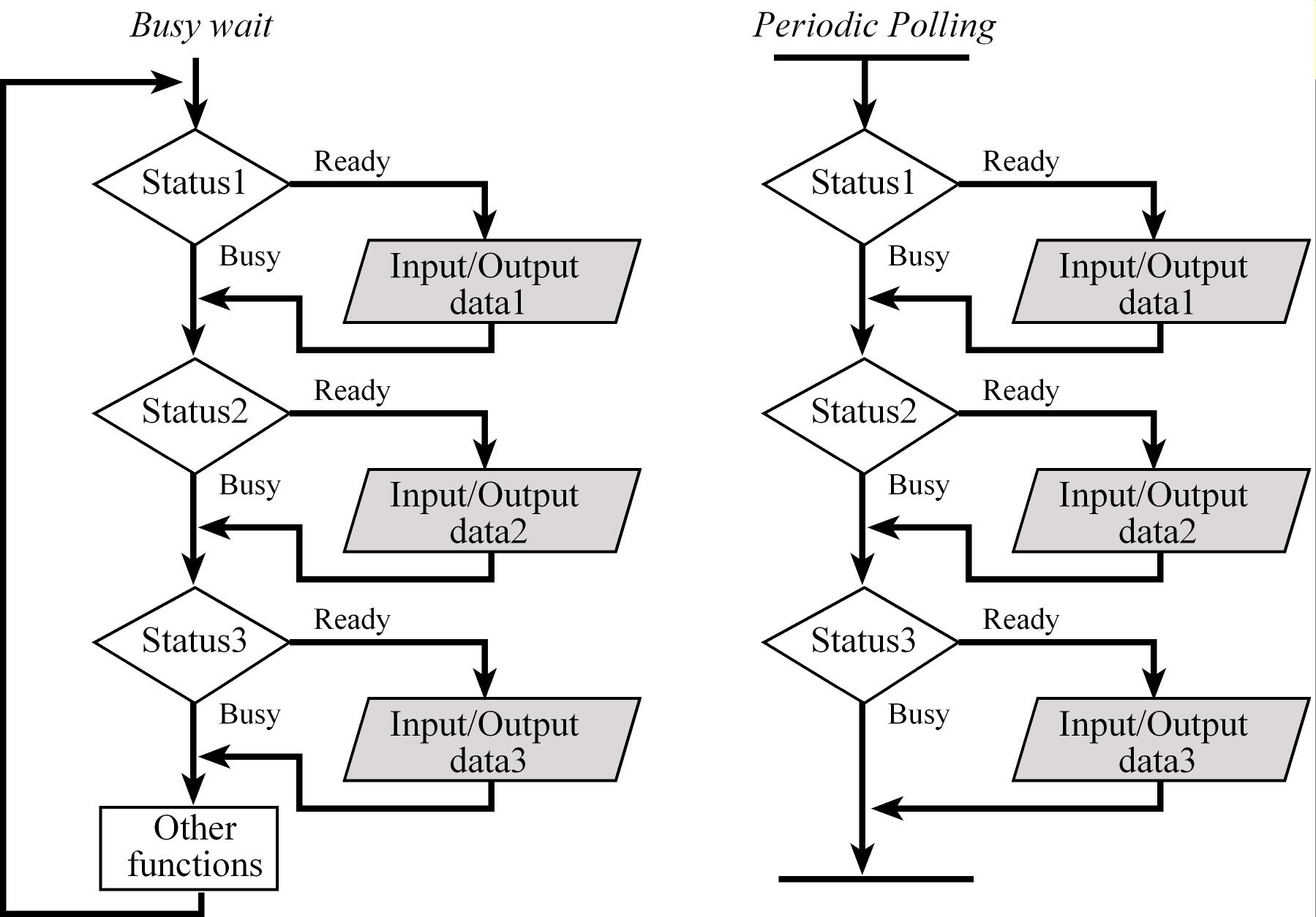
**Edge Triggered Interrupts**

* Using edge triggered interfacing allows the software to respond quickly to changes in the external world.
* Each of the digital I/O pins on the TM4C can be configured for edge triggering.
* **Edge triggering port configuration**:
  + **Port Configuration**
    - Enable the port clock (RCGC2)
    - Enable the digital pins (DEN) for selected pins on the port
    - Clear the corresponding bits in (PCTL) register:
    - Clear the bits in alternate function select (AFSEL) register.
    - Clear the direction (DIR) register to make them input
    - Clear the (AMSEL) to disable analog function
  + **Edge Triggering Configuration**



* + - Clear the IS (interrupt sense) bit to enable edge triggering
    - Clear the IBE (interrupt both edges) bit, or set IBE to trigger on rising and falling edge
    - Clear IEV (interrupt event) bit for falling edge, Set IEV for rising edge trigger
    - Set the IME (interrupt mask enable;arm) to 1 so that the trigger will cause an interrupt
* **RIS** (Raw Interrupt Status) aka the interrupt trigger is set by a hardware event, and the software (acknowledgement) clears the flag.

**Periodic Interrupts**

* Used when we need to run a routine at regular intervals
* Uses the systick timer to create periodic interrupts
* Periodic interrupts are useful in the following systems:
  + **Smoke detectors** use periodic interrupts:
    - Wakes up every 30 seconds
      * Flash an LED
      * Check if there is smoke or no smoke
  + **Data acquisition** systems:
    - A sample rate is established
      * Time between ADC samples must be equal and known for a digital signal process to function properly
  + **MCU based control system**:
    - Periodic rate for reading data
      * Read data from sensors and output commands to the actuators
* Busy-wait vs Periodic polling:
* 
  + With periodic polling:
    - I/O’s are polled on a regular basis
      * If no device needs service then the interrupt returns
  + With busy-wait:
    - The main program polls the I/O devices continuously
* When to use periodic polling:
  + Is appropriate for low bandwidth devices where real-time response is not necessary
  + When the I/O hardware cannot generate interrupts directly
  + When we wish to perform the I/O functions in the background