BBM 432 – EMBEDDED SYSTEMS

Week-4

Outline

- Timers
- Interrupts

SysTick timer

- SysTick is a simple counter that we can use to create time delays and generate periodic interrupts.
- The basis of SysTick is a 24-bit down counter that runs at the bus clock frequency.

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

Table 9.1. SysTick registers.

Initalizing SysTick timer

- There are four steps to initialize the SysTick timer.
- First, we clear the **ENABLE** bit to turn off SysTick during initialization.
- Second, we set the **RELOAD** register.
- Third, we write to the **NVIC_ST_CURRENT_R** value to clear the counter.
- Lastly, we write the desired mode to the control register, **NVIC_ST_CTRL_R**.

SysTick timer

- We set the **CLK_SRC** bit specifying the core clock will be used.
- We will set **CLK_SRC**=1, so the counter runs off the system clock.
- Later, we will set **INTEN** to enable interrupts, but in this first example we clear **INTEN** so interrupts will not be requested. We need to set the **ENABLE** bit so the counter will run.

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

SysTick timer

- When 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.
- In this way, the SysTick counter (**CURRENT**) is continuously decrementing.

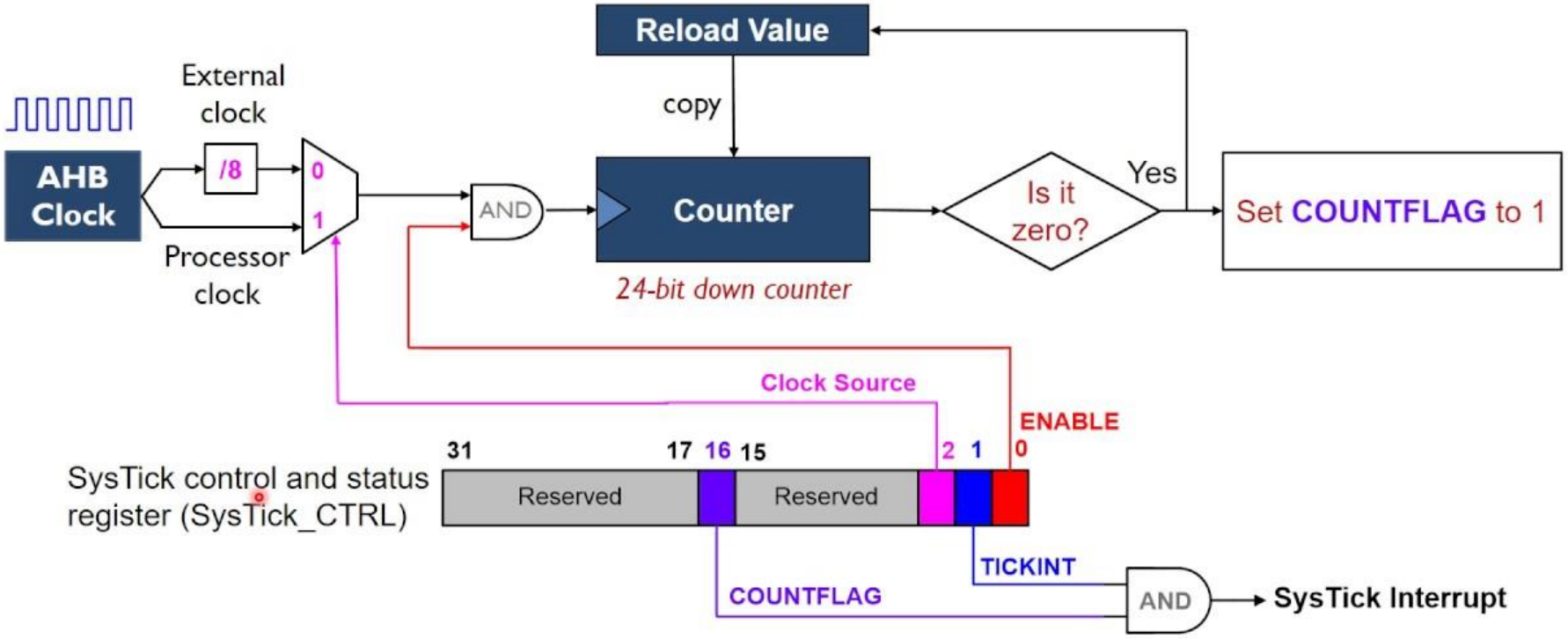
Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

SysTick timer

- If the **RELOAD** value is *n*, then the SysTick counter operates at modulo *n*+1 (…*n*, *n*-1, *n*-2 … 1, 0, *n*, *n*-1, …).
- In other words, it rolls over every *n*+1 counts.
- In this chapter we set **RELOAD** to 0x00FFFFFF, so the **CURRENT** value is a simple indicator of what time it is now.

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

Diagram of System Timer (SysTick)



Counter timing

- Without activating the phase-lock-loop (PLL), our TM4C123 LaunchPad will run at 16 MHz, meaning the SysTick counter decrements every 62.5 ns.
- If we activate the PLL to run the microcontroller at 50 MHz, then the SysTick counter decrements every 20 ns.
- In general, if the period of the core bus clock is *t*, then the **COUNT** flag will be set every (*n*+1)*t*.

Counter timing

- Reading the **NVIC_ST_CTRL_R** control register will return the **COUNT** flag in bit 16 and then clear the flag.
- Also, writing any value to the **NVIC_ST_CURRENT_R** register will reset the counter to zero and clear the **COUNT** flag.
- To determine the time, one simply reads the **NVIC_ST_CURRENT_R** register.

Sample initialization

```
#define NVIC_ST_CTRL_R      (*((volatile unsigned long *)0xE000E010))
#define NVIC_ST_RELOAD_R    (*((volatile unsigned long *)0xE000E014))
#define NVIC_ST_CURRENT_R   (*((volatile unsigned long *)0xE000E018))
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 = 0x00000005;     // 4) enable SysTick with core clock
}
```

Sample code

- To determine the time, one simply reads the **NVIC_ST_CURRENT_R** register.
- Program shows how to measure the elapsed time between calls to a function.
- Assume the system calls the function **Action()** over and over.

```
unsigned long Now;      // 24-bit time at this call (62.5ns)
unsigned long Last;     // 24-bit time at previous call (62.5ns)
unsigned long Elapsed;  // 24-bit time between calls (62.5ns)
void Action(void){      // function under test
  Now = NVIC_ST_CURRENT_R;        // what time is it now?
  Elapsed = (Last-Now)&0x00FFFFF; // 24-bit difference
  Last = Now;                     // set up for next
  ...
}
```

Sample code

- **Now** is the time (in 62.5ns units) when the function has been called.
- **Last** is the time (also in 62.5ns units) when the function was called previously.
- Subtract **Last-Now**.
- Since the time is only 24 bits and the software variables are 32 bits we "**and**" with 0x00FFFFFF to create a 24-bit difference.

```
unsigned long Now;      // 24-bit time at this call (62.5ns)
unsigned long Last;     // 24-bit time at previous call (62.5ns)
unsigned long Elapsed;  // 24-bit time between calls (62.5ns)
void Action(void){      // function under test
  Now = NVIC_ST_CURRENT_R;        // what time is it now?
  Elapsed = (Last-Now)&0x00FFFFFF; // 24-bit difference
  Last = Now;                     // set up for next
  ...
}
```

Sample code

- The first measurement will be wrong because there is no previous execution from which to measure.
- The system will be accurate as long as the elapsed time is less than a second.
- More precisely, as long as the elapsed time is less than 2^{24}*62.5ns.
- This is similar to the problem of using an analog clock to measure elapsed time.
- For example you notice the clock says 10:00 when you go to sleep, and you notice it says 7:00 when you wake up. As long as you are sure you slept less than 12 hours, you are confident you slept for 9 hours.

Other timers

- Our TM4C123 microcontroller has some 32-bit and some 64-bit timers, but we will use SysTick because it is much simpler to configure.
- We just have to be aware that we are limited to 24 bits.

Internal Clock

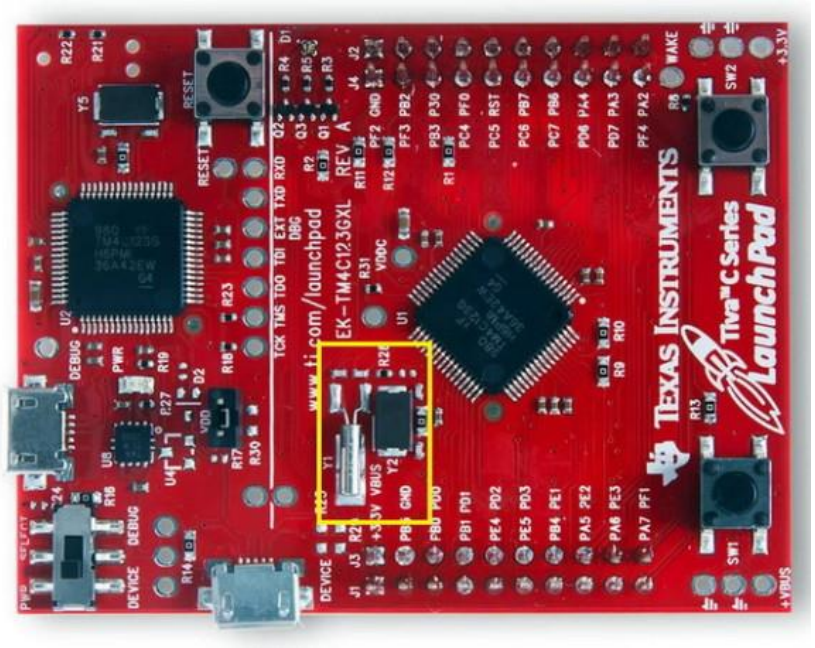
- The default bus speed for the LM4F/TM4C internal oscillator is 16 MHz ±1%.
- The internal oscillator is significantly less precise than the crystal, but it requires less power and does not need an external crystal.
- If we wish to have accurate control of time, we will activate the external crystal (called the main oscillator) use the PLL to select the desired bus speed.

External Clock

- EK-TM4C123GXL boards have an external 16 MHz crystal.
- Phase-lock-loop (PLL) allows the software to adjust the execution speed of the computer.
- There is a tradeoff between software execution speed and electrical power.
  - Slowing down the bus clock will require less power to operate and generate less heat.
  - Speeding up the bus clock allows for more calculations per second, at the cost of requiring more power to operate and generating more heat.

External Crystal

- An external crystal is attached to the LM4F/TM4C microcontroller.
- Higher precision



PLL advantages

- PLL gives a more precise clock control.
- PLL gives flexibility.
  - High speed, high power
  - Low speed, low power

Ways to activate PLL

- There are two ways to activate the PLL. We could call a library function, or we could access the clock registers directly.
- In general, using library functions creates a better design because the solution will be more stable (less bugs) and will be more portable (easier to switch microcontrollers).
- However, the objective of the class is to present microcontroller fundamentals. Showing the direct access does illustrate some concepts of the PLL.

Main clock registers

XTAL	*Crystal Freq (MHz)*
0x0	Reserved
0x1	Reserved
0x2	Reserved
0x3	Reserved
0x4	3.579545 MHz
0x5	3.6864 MHz
0x6	4 MHz
0x7	4.096 MHz
0x8	4.9152 MHz
0x9	5 MHz
0xA	5.12 MHz
0xB	6 MHz (reset value)
0xC	6.144 MHz
0xD	7.3728 MHz
0xE	8 MHz
0xF	8.192 MHz

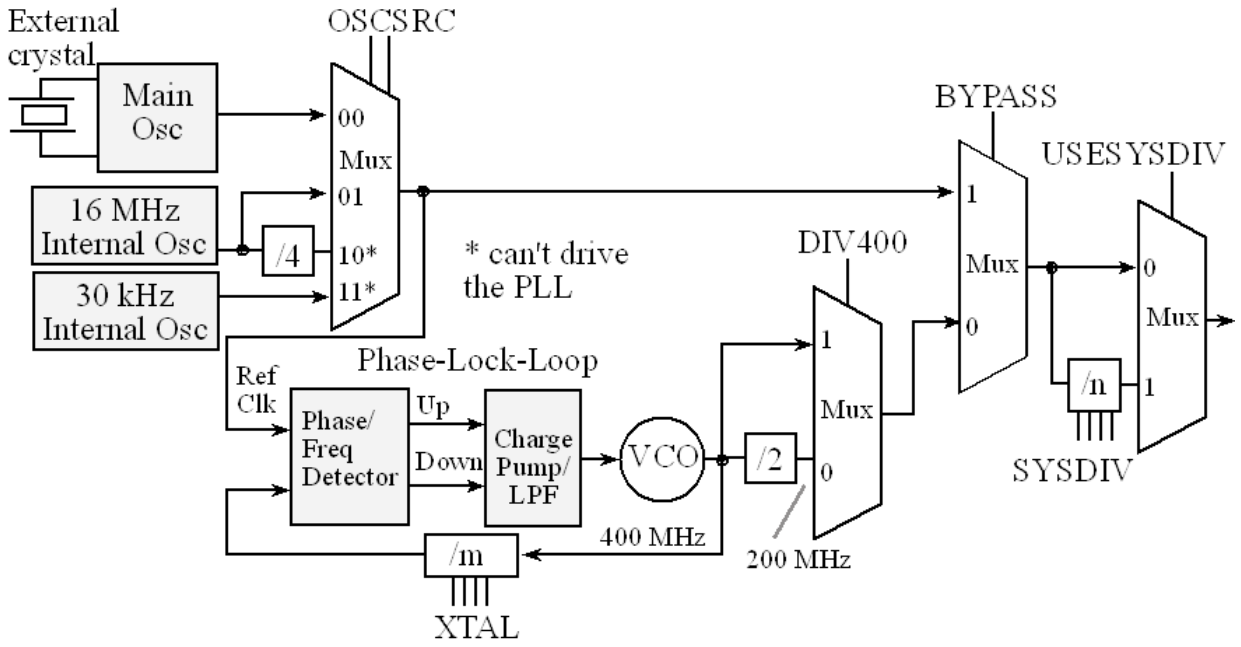
XTAL	*Crystal Freq (MHz)*
0x10	10.0 MHz
0x11	12.0 MHz
0x12	12.288 MHz
0x13	13.56 MHz
0x14	14.31818 MHz
0x15	16.0 MHz
0x16	16.384 MHz
0x17	18.0 MHz
0x18	20.0 MHz
0x19	24.0 MHz
0x1A	25.0 MHz
0x1B	Reserved
0x1C	Reserved
0x1D	Reserved
0x1E	Reserved
0x1F	Reserved

Address	26-23	22	13	11	10-6	5-4	Name
$400FE060	SYSDIV	USESYSDIV	PWRDN	BYPASS	XTAL	OSCSRC	SYSCTL_RCC_R
$400FE050					PLLRIS		SYSCTL_RIS_R

	31	30	28-22	13	11	6-4	
$400FE070	USERCC2	DIV400	SYSDIV2	PWRDN2	BYPASS2	OSCSRC2	SYSCTL_RCC2_R

Main oscillator

- The output of the main oscillator (Main Osc) is a clock at the same frequency as the crystal.
- By setting the OSCSRC bits to 0, the multiplexer control will select the main oscillator as the clock source.



Phase Lock Loop

- The main oscillator for the LM4F/TM4C LaunchPad will be 16 MHz.
- This means the reference clock (Ref Clk) input to the phase/frequency detector will be 16 MHz.
- For a 16 MHz crystal, we set the XTAL bits to 10101 (see Table 10.1).
- In this way, a 400 MHz output of the voltage controlled oscillator (VCO) will yield a 16 MHz clock at the other input of the phase/frequency detector.

Phase Lock Loop

- If the 400 MHz clock is too slow, the **up** signal will add to the charge pump, increasing the input to the VCO, leading to an increase in the 400 MHz frequency.
- If the 400 MHz clock is too fast, **down** signal will subtract from the charge pump, decreasing the input to the VCO, leading to a decrease in the 400 MHz frequency.
- Because the reference clock is stable, the feedback loop in the PLL will drive the output to a stable 400 MHz frequency.

Activate the LM4F/TM4C with a 16 MHz crystal to run at 80 MHz

- Program shows the steps 0 to 6 to activate the TM4C123 Launchpad with a 16 MHz main oscillator to run at 80 MHz.
- 0) Use RCC2 because it provides for more options.
- 1) The first step is to set BYPASS2 (bit 11).

```
void PLL_Init(void){
  // 0) Use RCC2
  SYSCTL_RCC2_R |=  0x80000000;  // USERCC2
  // 1) bypass PLL while initializing
  SYSCTL_RCC2_R |=  0x00000800;  // BYPASS2, PLL bypass
```

Address	26-23	22	13	11	10-6	5-4	Name
$400FE060	SYSDIV	USESYSDIV	PWRDN	BYPASS	XTAL	OSCSRC	SYSCTL_RCC_R
$400FE050					PLLRIS		SYSCTL_RIS_R

	31	30	28-22	13	11	6-4	
$400FE070	USERCC2	DIV400	SYSDIV2	PWRDN2	BYPASS2	OSCSRC2	SYSCTL_RCC2_R

Activate the LM4F/TM4C with a 16 MHz crystal to run at 80 MHz

2) The second step is to specify the crystal frequency in the four XTAL bits using the code in Table 10.1. The OSCSRC2 bits are cleared to select the main oscillator as the oscillator clock source.

```
// 2) select the crystal value and oscillator source
  SYSCTL_RCC_R = (SYSCTL_RCC_R
&~0x000007C0)   // clear XTAL field, bits 10-6
                + 0x00000540;   // 10101, configure for 16 MHz
crystal
  SYSCTL_RCC2_R &= ~0x00000070;  // configure for
main oscillator source
```

Activate the LM4F/TM4C with a 16 MHz crystal to run at 80 MHz

- 3) The third step is to clear PWRDN2 (bit 13) to activate the PLL.

```
// 3) activate PLL by clearing PWRDN
SYSCTL_RCC2_R &= ~0x00002000;
```

- 4) The fourth step is to configure and enable the clock divider using the 7-bit SYSDIV2 field. If the 7-bit SYSDIV2 is **n**, then the clock will be divided by **n**+1. To get the desired 80 MHz from the 400 MHz PLL, we need to divide by 5. So, we place a 4 into the SYSDIV2 field.

```
// 4) set the desired system divider
  SYSCTL_RCC2_R |= 0x40000000;   // use 400 MHz PLL
  SYSCTL_RCC2_R = (SYSCTL_RCC2_R&~ 0x1FC00000)  // clear system clock divider
                  + (4<<22);      // configure for 80 MHz clock
```

Activate the LM4F/TM4C with a 16 MHz crystal to run at 80 MHz

- 5) The fifth step is to wait for the PLL to stabilize by waiting for PLLRIS (bit 6) in the **SYSCTL_RIS_R** to become high.

```
// 5) wait for the PLL to lock by polling PLLLRIS
while((SYSCTL_RIS_R&0x00000040)==0){};  // wait for PLLRIS bit
```

- 6) The last step is to connect the PLL by clearing the BYPASS2 bit.

```
// 6) enable use of PLL by clearing BYPASS
SYSCTL_RCC2_R &= ~0x00000800;
```

To save power

- Optimize your code so that it needs fewer bus cycles.
- Reduce your bus frequency.

A question

- If we activate the PLL and change the bus clock to 50 MHz (20ns), what is the longest elapsed time we could measure with Program 9.2?

A question

- If we activate the PLL and change the bus clock to 50 MHz (20ns), what is the longest elapsed time we could measure with Program 9.2?

- The longest time is 2^{24} * 20ns is 335.5ms.

Accurate Time Delays using SysTick

- The accuracy of SysTick depends on the accuracy of the clock.
- The internal oscillator is significantly less precise than the crystal, but it requires less power.
- We use the PLL to derive a bus clock based on the 16 MHz crystal, the time measured or generated using SysTick will be very accurate.
- More specifically, the accuracy of the NX5032GA crystal on the LaunchPad board is ±50 parts per million (PPM), which translates to 0.005%, which is about ±5 seconds per day.

Improving accuracy

- One could spend more money on the crystal and improve the accuracy by a factor of 10.
- Not only are crystals accurate, they are stable. The NX5032GA crystal will vary only ±150 PPM as temperature varies from -40 to +150 ºC.
- Crystals are more stable than they are accurate, typically varying by less than 5 PPM per year.

Time delay based on SysTick

- Program 10.2 shows a simple function to implement time delays based on SysTick.
- The **RELOAD** register is set to the number of bus cycles one wishes to wait.
- If the PLL function of Program 10.1 has been executed, then the units of this delay will be 12.5 ns.
- Writing to **CURRENT** will clear the counter and will clear the count flag (bit 16) of the **CTRL** register

```
#define NVIC_ST_CTRL_R      (*((volatile unsigned long *)0xE000E010))
#define NVIC_ST_RELOAD_R    (*((volatile unsigned long *)0xE000E014))
#define NVIC_ST_CURRENT_R   (*((volatile unsigned long *)0xE000E018))
void SysTick_Init(void){
  NVIC_ST_CTRL_R = 0;               // disable SysTick during setup
  NVIC_ST_CTRL_R = 0x00000005;      // enable SysTick with core clock
}
// The delay parameter is in units of the 80 MHz core clock. (12.5 ns)
void SysTick_Wait(unsigned long delay){
  NVIC_ST_RELOAD_R = delay-1;  // number of counts to wait
  NVIC_ST_CURRENT_R = 0;       // any value written to CURRENT clears
  while((NVIC_ST_CTRL_R&0x00010000)==0){ // wait for count flag
  }
}
// 800000*12.5ns equals 10ms
void SysTick_Wait10ms(unsigned long delay){
  unsigned long i;
  for(i=0; i<delay; i++){
    SysTick_Wait(800000);  // wait 10ms
  }
}
```

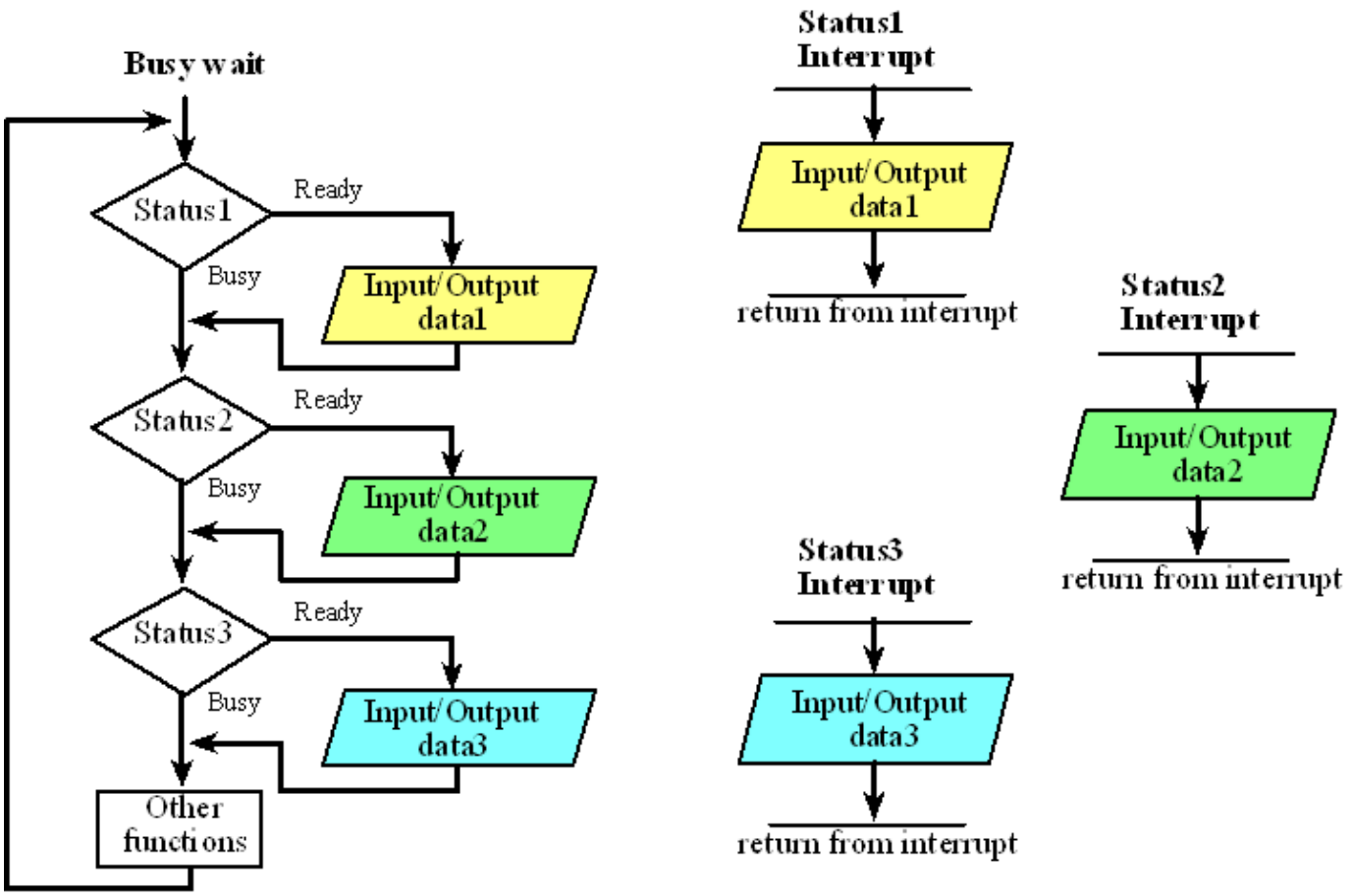
Accurate Time Delays using SysTick

- After SysTick has been decremented **delay** times, the count flag will be set and the **while** loop will terminate.
- Since SysTick is only 24 bits, the maximum time one can wait with **SysTick_Wait** is 2^{24}*12.5ns, which is about 200 ms.
- To provide for longer delays, the function **SysTick_Wait10ms** calls the function **SysTick_Wait** repeatedly. Notice that 800,000*12.5ns is 10ms.

Interrupts

- Appreciate the need to perform multiple tasks concurrently.
- Understand perform measures of a real-time system such as bandwidth and latency
- Learn how interrupts can be used to minimize latency.
- Study the basics of interrupt programming: arm, enable, trigger, vector, priority, acknowledge.
- Understand how to use SysTick to create periodic interrupts
- Use SysTick to create sounds.

Why do we need interrupts?



Interaction with the environment

- An embedded system uses its input/output devices to interact with the external world.
  - Input devices allow the computer to gather information, and output devices can display information.
  - Output devices also allow the computer to manipulate its environment.
- The tight-coupling between the computer and external world distinguishes an embedded system from a regular computer system.

Sync between hardware and software

- Software executes much faster than the hardware.
- E.g., it might take the software only 1 us to ask the hardware to clear the LCD, but the hardware might take 1 ms to complete the command.
  - During this time, the software could execute tens of thousands of instructions.
  - Synchronization between the executing software and its external environment is critical

What is an interrupt?

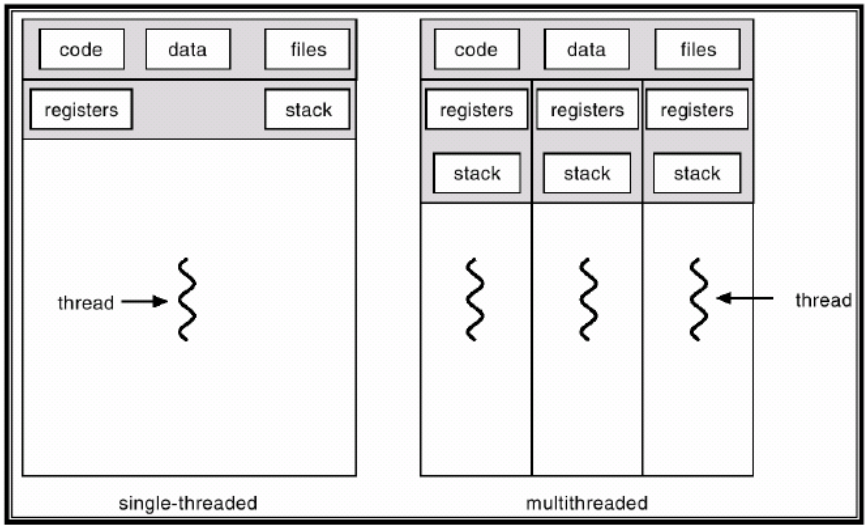
- **Interrupt** is the automatic transfer of software execution in response to a hardware event called **trigger**.
- The hardware event can either be
  - a busy to ready transition in an external I/O device (like the UART input/output)
  - an internal event (like bus fault, memory fault, or a periodic timer).
- When the hardware needs service, it will request an interrupt by setting its trigger flag.

Thread

- A **thread** is defined as the path of action of software as it executes.
  - A new thread is created for each interrupt request.
  - Created by the hardware interrupt request and is killed when the interrupt service routine returns from interrupt.
  - Each individual request is a separate thread.
    - local variables and registers used in the interrupt service routine are unique and separate from one interrupt event to the next interrupt.

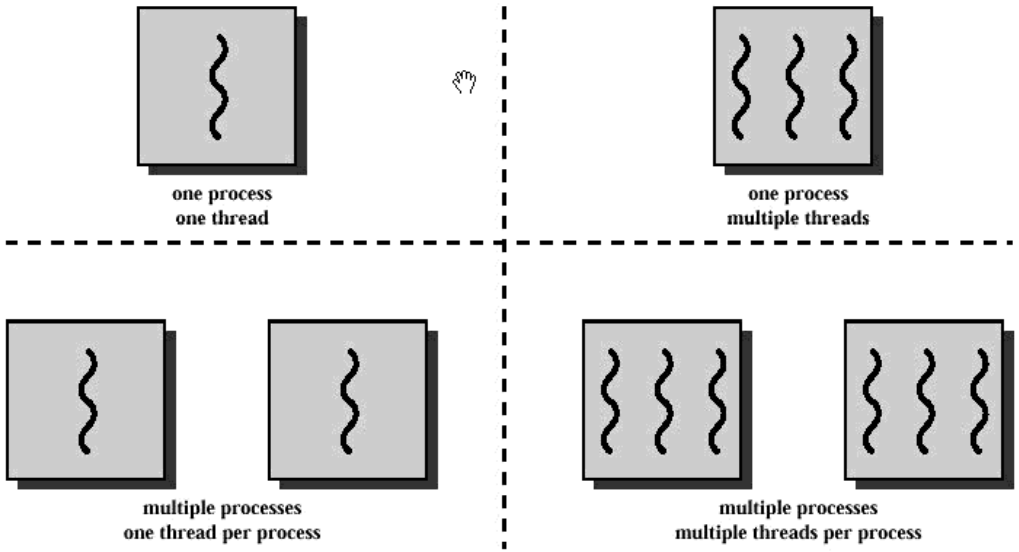
Multi-threading

- In a **multi-threaded** system, we consider the threads as cooperating to perform an overall task.
- Consequently we will develop ways for the threads to communicate (e.g., FIFO) and to synchronize with each other.
- Most embedded systems have a single common overall goal.
- On the other hand, general-purpose computers can have multiple unrelated functions to perform.

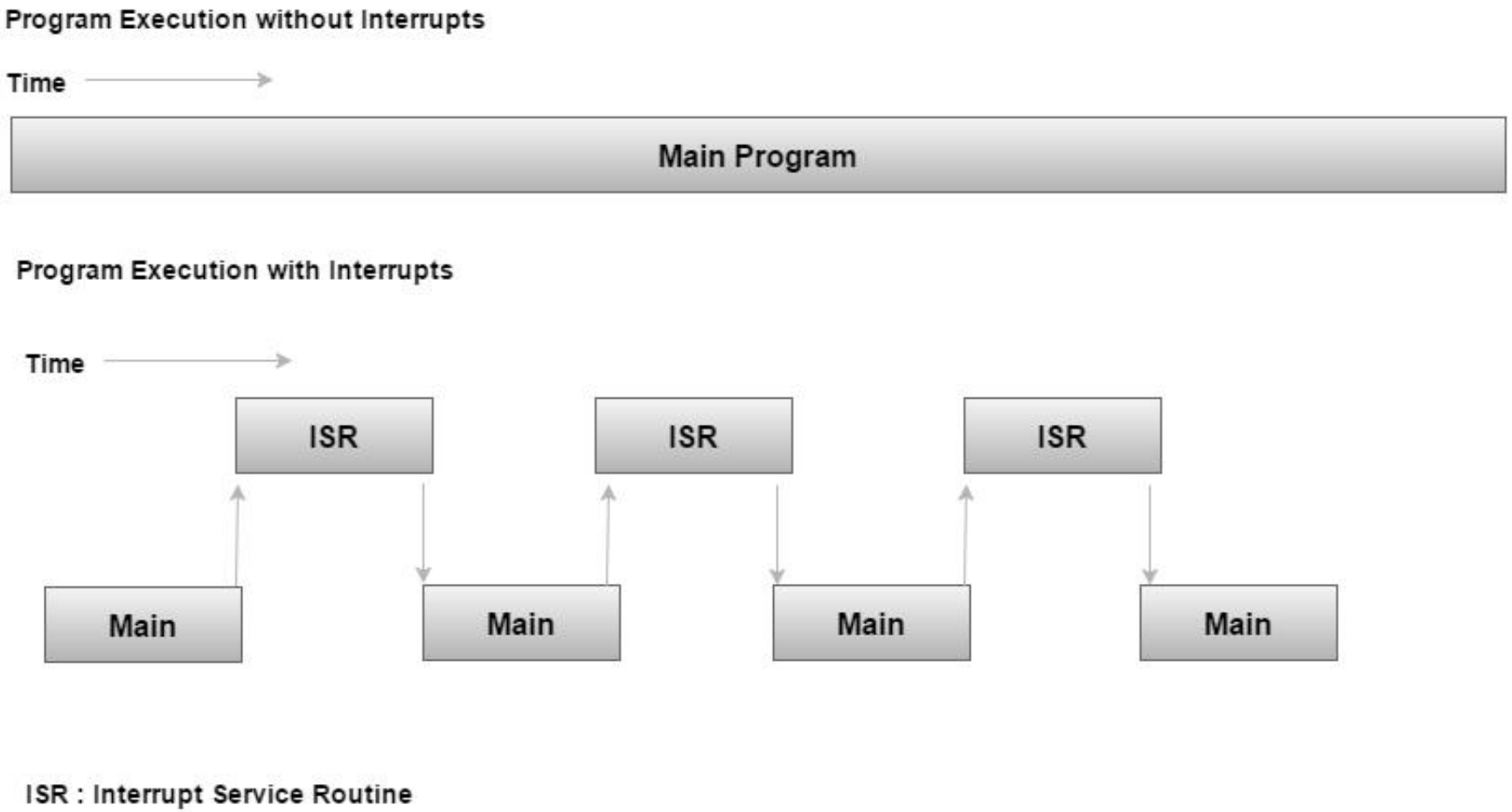


Processes

- A **process** is also defined as the action of software as it executes.
- Processes do not necessarily cooperate towards a common shared goal.
  - Threads share access to I/O devices, system resources, and global variables,
  - Processes have separate global variables and system resources.



Program execution with interrupts



Definitions: arm, disarm

- To **arm** a device means to allow the hardware trigger to interrupt.
  - One arms a trigger if one is interested in interrupts from this source.
- Conversely, to **disarm** a device means to shut off or disconnect the hardware trigger from the interrupts.
  - One disarms a trigger if one is not interested in interrupts from this source.
- Each potential interrupting trigger has a separate arm bit.

Software control on interrupts: Arm bits

- Each potential interrupt trigger has a separate **arm** bit that the software can activate or deactivate.
- The software
  - will set the arm bits for those devices from which it wishes to accept interrupts
  - will deactivate the arm bits within those devices from which interrupts are not to be allowed.

Definitions: Enable, disable

- To **enable** means to allow interrupts at this time.
- To **disable** means to postpone interrupts until a later time.
- On the ARM Cortex-M processor there is one interrupt enable bit for the entire interrupt system.
- We disable interrupts if it is currently not convenient to accept interrupts.
- In C, we enable and disable interrupts by calling the functions **EnableInterrupts()** and **DisableInterrupts()** res pectively.

Software control on interrupts: Enable bit

- For most devices there is a enable bit in the NVIC that must be set (periodic SysTick interrupts are an exception, having no NVIC enable).
- Specifically, bit 0 of the special register **PRIMASK** is the interrupt mask bit, **I**.
- If this bit is 1 most interrupts and exceptions are not allowed, which we will define as **disabled**.
- If the bit is 0, then interrupts are allowed, which we will define as **enabled**.

Software control on interrupts: Priority

- The **BASEPRI** register prevents interrupts with lower priority interrupts, but allows higher priority interrupts.
- For example: if the software sets the **BASEPRI** to 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed.
- The software can also specify the priority level of each interrupt request.
- If **BASEPRI** is zero, then the priority feature is disabled and all interrupts are allowed.

Hardware triggers

- Example: **Count** flag in the **NVIC_ST_CTRL_R** register which is set periodically by SysTick.
- Another example: bits in the **GPIO_PORTF_RIS_R** register that are set on rising or falling edges of digital input pins.

Five conditions of an interrupt

- 1) device arm,
  2) NVIC enable,
  3) global enable,
  4) interrupt priority level must be higher than current level executing, and
  5) hardware event trigger.
- For an interrupt to occur, these five conditions must be simultaneously true but can occur in any order.

Context Switch

- The current instruction is finished.
- The execution of the currently running program is suspended.
- A **context switch** occurs automatically in hardware as the context is switched from a foreground thread to a background thread.
- We can also have a context switch from a lower priority ISR to a higher priority ISR.
- Next, the software executes the ISR.

What happens if interrupts are disabled?

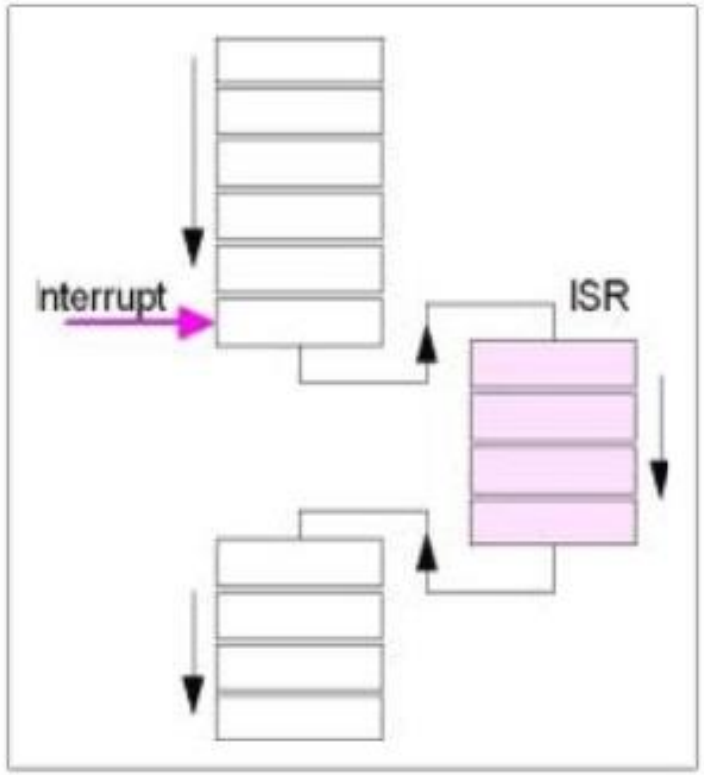
- If a trigger flag is set, but the interrupts are disabled, the interrupt level is not high enough, or the flag is disarmed, the request is not dismissed.
  - The request is held **pending**, postponed until a later time, when the system deems it convenient to handle the requests.
  - Once the trigger flag is set, under most cases it remains set until the software clears it.
- For example: The software can disable interrupts, run some code that needs to run to completion, and then enable the interrupt again.
- A trigger occurring while running with the interrupts are disabled is postponed until the time the I bit is cleared again.

Clearing the trigger flag

- Clearing a trigger flag is called **acknowledgement**, which occurs only by specific software action.
  - Each trigger flag has a specific action software must perform to clear that flag.
- The SysTick periodic interrupt will be the only example of an automatic acknowledgement.
  - For SysTick, the periodic timer requests an interrupt, but the trigger flag will be automatically cleared when the ISR runs.
  - For all the other trigger flags, the ISR must explicitly execute code that clears the flag.

Clearing the trigger flag

- The **interrupt service routine** (ISR) is the software module that is executed when the hardware requests an interrupt.
- There may be one large ISR that handles all requests (polled interrupts), or many small ISRs specific for each potential source of interrupt (vectored interrupts).
- The ISR software must clear the trigger flag that caused the interrupt (acknowledge). (Except SysTick interrupt)



After the interrupt

- Execution of the previous thread will then continue with the exact stack and register values that existed before the interrupt.
- Parameter passing between threads must be implemented using shared global memory variables.
- A private global variable can be used if an interrupt thread wishes to pass information to itself, e.g., from one interrupt instance to another.
- The execution of the main program is called the foreground thread, and the executions of the various interrupt service routines are called background threads.

Interrupts should be handled as fast as possible.

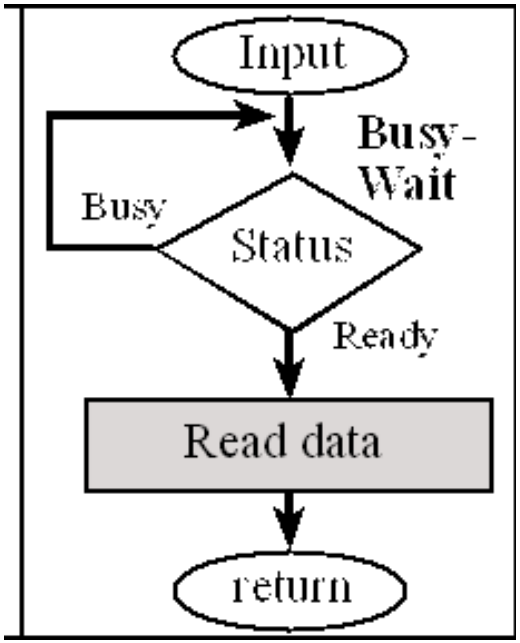
- ISR should execute as fast as possible.
  - The interrupt should occur when it is time to perform a needed function, and the interrupt service routine should perform that function, and return right away.
- Placing backward branches (busy-wait loops, iterations) in the interrupt software should be avoided if possible.
- The percentage of time spent executing interrupt software should be small when compared to the time between interrupt triggers.

Performance measures: latency and bandwidth.

- For an input device, the **interface latency** is the time between when new input is available, and the time when the software reads the input data.
- For an output device, the interface latency is the time between when the output device is idle, and the time when the software writes new data.
- We can also define **device latency** as the response time of the external I/O device.
  - If we request that a certain sector be read from a disk, then the device latency is the time it take to find the correct track and spin the disk (seek) so the proper sector is positioned under the read head.
- A **real-time** system is one that can guarantee a worst case interface latency.
- **Bandwidth** is defined as the amount of data/sec being processed.

Interrupts or busy-wait?

- One should not always use busy wait because one is too lazy to implement the complexities of interrupts.
- On the other hand, one should not always use interrupts because they are fun and exciting.
- Busy-wait synchronization is appropriate when the I/O timing is predictable and when the I/O structure is simple and fixed.
- Busy wait should be used for dedicated single thread systems where there is nothing else to do while the I/O is busy.



Interrupts or busy-wait?

- Interrupt synchronization is appropriate when the I/O timing is variable, and when the I/O structure is complex.
- Interrupts are efficient when there are I/O devices with different speeds.
- Interrupts allow for quick response times to important events.
- Using interrupts is one mechanism to design real-time systems, where the interface latency must be short and bounded.
- **Bounded** means it is always less than a specified value.
- **Short** means the specified value is acceptable to our consumers.

Periodic interrupts

- Interrupts can also be used for infrequent but critical events like power failure, memory faults, and machine errors.
- Periodic interrupts will be useful for real-time clocks, data acquisition systems, and control systems.

Interrupt requesting

- All interrupting systems must have the **ability for the hardware to request action from computer**.
- In general, the interrupt requests can be generated using a separate connection to the processor for each device.
- The TM4C microcontrollers use separate connections to request interrupts.

Determining the source of the interrupt

- All interrupting systems must have the **ability for the computer to determine the source**.
- A vectored interrupt system employs separate connections for each device so that the computer can give automatic resolution.
- You can recognize a vectored system because each device has a separate interrupt vector address.
- With a polled interrupt system, the interrupt software must poll each device, looking for the device that requested the interrupt.

Determining the source of the interrupt

- Most interrupts on the TM4C microcontrollers are vectored, but there are some triggers that share the same vector.
- For these interrupts the ISR must poll to see which trigger caused the interrupt.
  - For example, all input pins on one GPIO port can trigger an interrupt, but the trigger flags share the same vector.
- So if multiple pins on one GPIO port are armed, the shared ISR must poll to determine which one(s) requested service.

Acknowledging the interrupt

- The third necessary component of the interface is the **ability for the computer to acknowledge the interrupt.**
- Normally there is a trigger flag in the interface that is set on the busy to ready state transition.
- In essence, this trigger flag is the cause of the interrupt.
- Acknowledging the interrupt involves clearing this flag.
- It is important to shut off the request, so that the computer will not mistakenly request a second (and inappropriate) interrupt service for the same condition.

Software acknowledge

- Except for periodic SysTick interrupts, TM4C microcontrollers use software acknowledge.
- So when designing an interrupting interface, it will be important to know exactly
  - what hardware condition sets the trigger flag (and request an interrupt)
  - how the software will clear it (acknowledge) in the ISR.

Inter-thread Communication and Synchronization

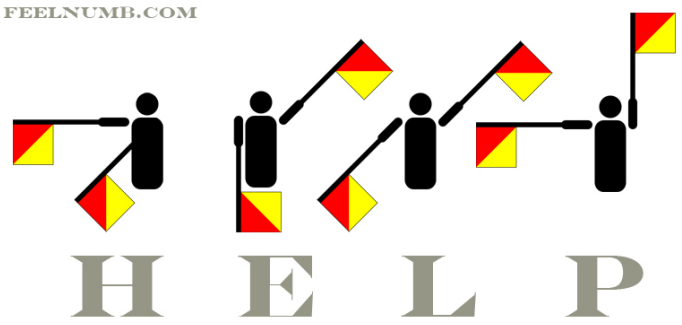
- For regular function calls we use the registers and stack to pass parameters, but interrupt threads have logically separate registers and stack.
- Registers are automatically saved by the processor as it switches from main program (foreground thread) to interrupt service routine (background thread).
- Exiting an ISR will restore the registers back to their previous values.
- **Thus, all parameter passing must occur through global memory.**
- One cannot pass data from the main program to the interrupt service routine using registers or the stack.

Synchronizing threads

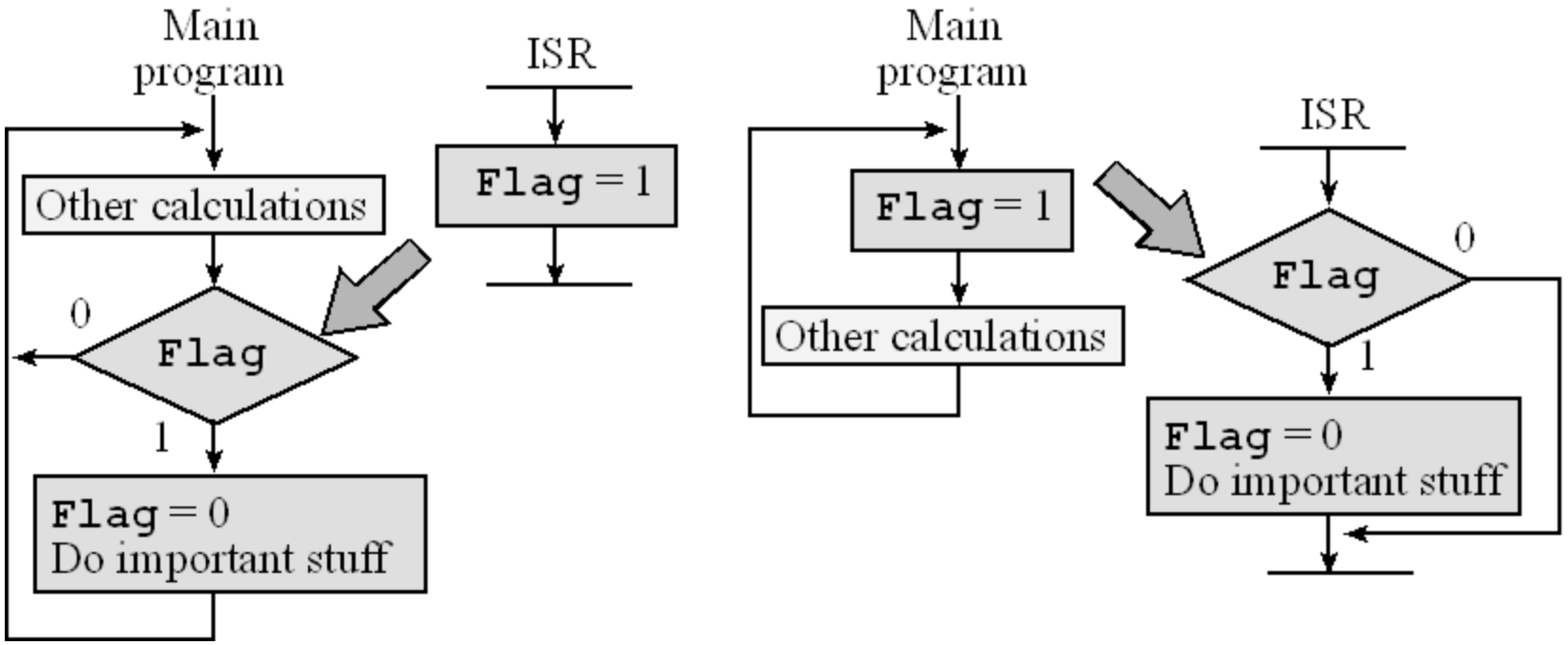
- In this chapter, multi-threading means one main program (foreground thread) and multiple ISRs (background threads).
- An operating system allows multiple foreground threads.
- Synchronizing threads is a critical task affecting efficiency and effectiveness of systems using interrupts.
- In this section, we will present in general form three constructs to synchronize threads:
  - binary semaphore
  - mailbox
  - FIFO queue.

Binary semaphore

- A **binary semaphore** is simply a shared flag.
- There are two operations one can perform on a semaphore.
- **Signal** is the action that sets the flag.
- **Wait** is the action that checks the flag, and if the flag is set, the flag is cleared and important stuff is performed.
- This flag must exist as a private global variable with restricted access to only these two code pieces.
- In C, we add the qualifier **static** to a global variable to restrict access to software within the same file.

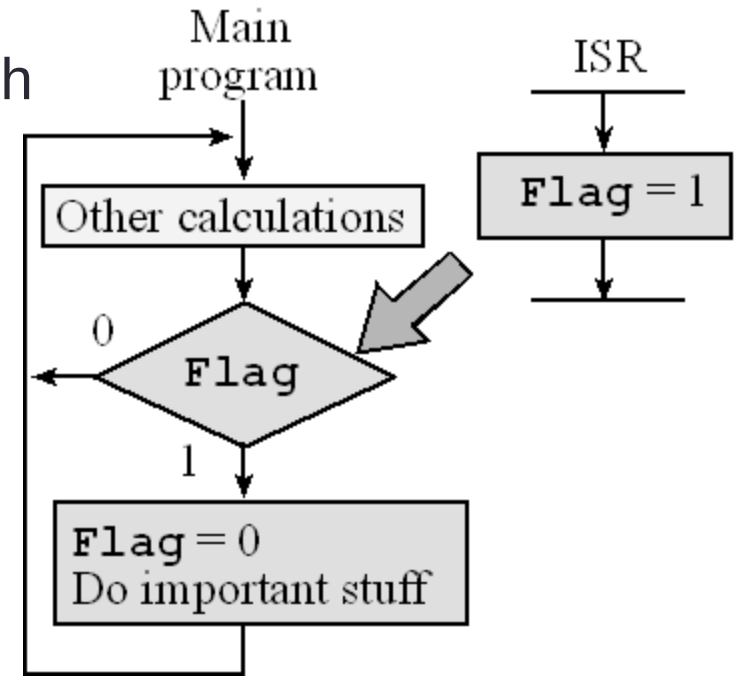


Binary semaphore



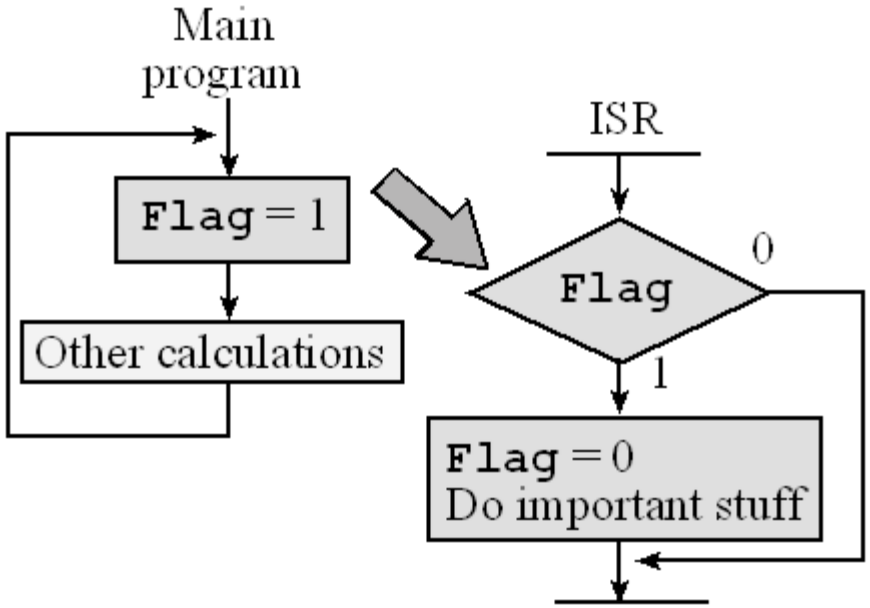
Binary semaphore

- A flag has two states: 0 and 1.
- For example, 0 might mean the switch has not been pressed, and 1 might mean the switch has been pressed.
- The big arrows in this figure signify synchronization links between the threads.
- In the example on the left, the ISR signals the semaphore and the main program waits on the semaphore.
- Notice the “important stuff” is run in the foreground once per execution of the ISR.



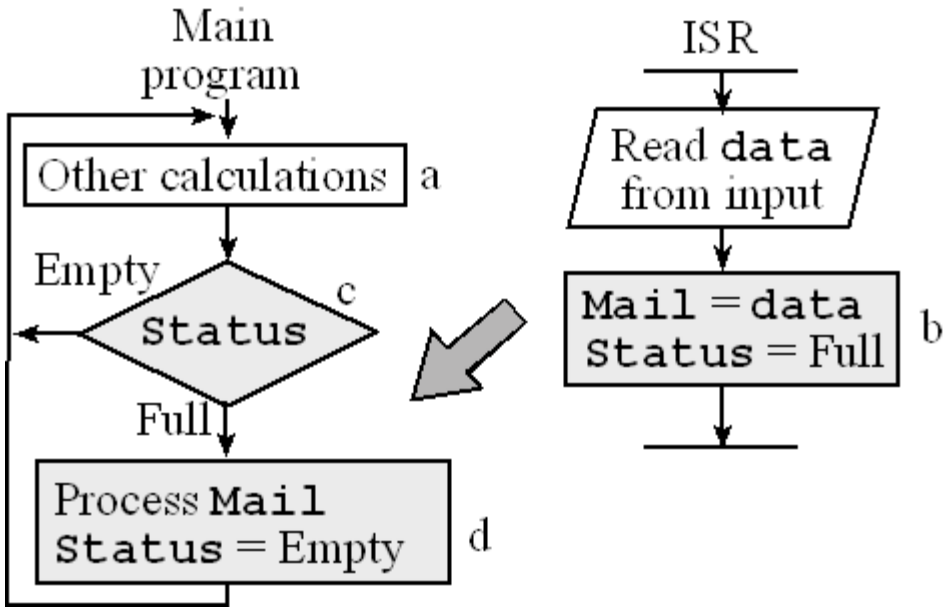
Binary semaphore

- In the example on the right, the main program signals the semaphore and the ISR waits.
- It is good design to have NO backwards jumps in an ISR.
- In this particular application, if the ISR is running and the semaphore is 0, the action is just skipped and the computer returns from the interrupt.



Mailbox

- The mailbox is a binary semaphore with associated data variable.
- The mailbox structure is implemented with two shared global variables.
- **Mail** contains data, and **Status** is a semaphore flag specifying whether the mailbox is full or empty.



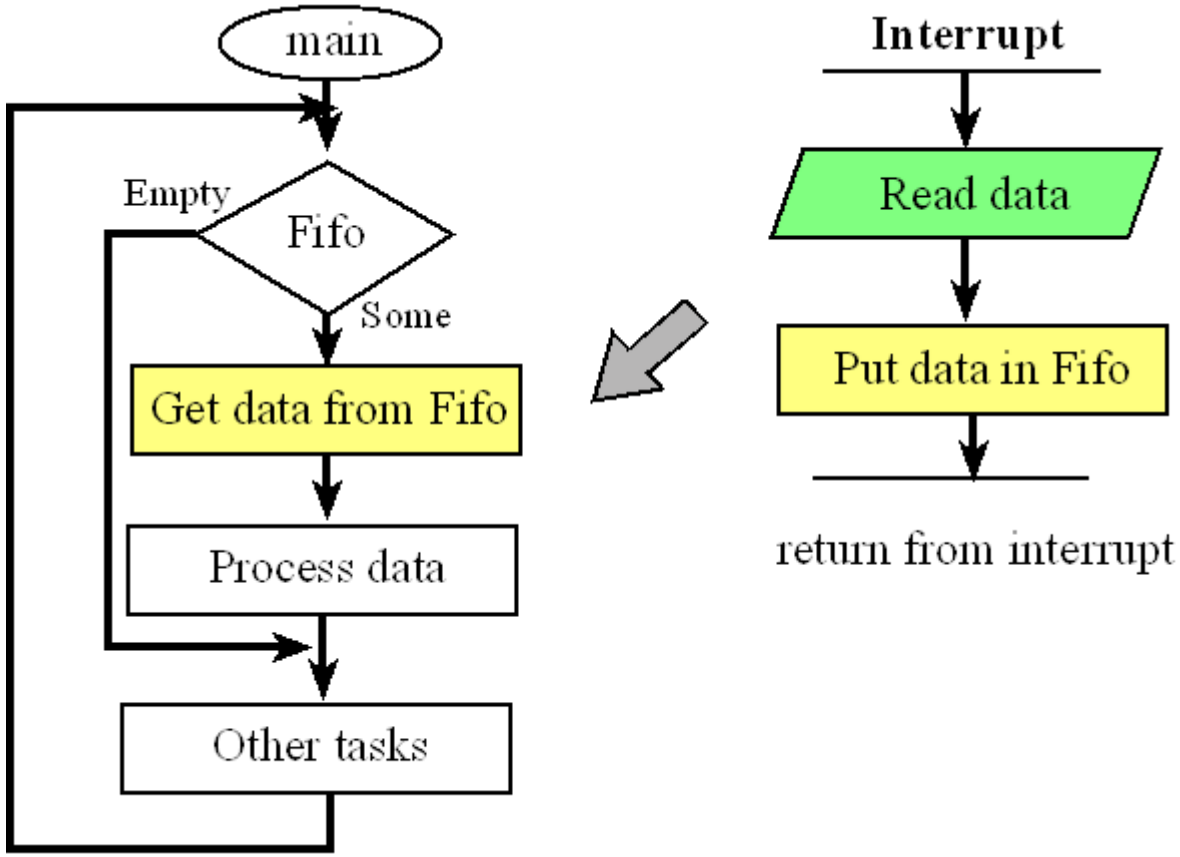
Mailbox

- The interrupt is requested when its trigger flag is set, signifying new data are ready from the input device.
- The ISR will read the data from the input device and store it in the shared global variable **Mail**, then update its **Status** to full.
- The main program will perform other calculations, while occasionally checking the status of the mailbox.
- When the mailbox has data, the main program will process it. This approach is adequate for situations where the input bandwidth is slow compared to the software processing speed.
- One way to visualize the interrupt synchronization is to draw a state versus time plot of the activities of the hardware, the mailbox, and the two software threads.

FIFO – input device

- The third synchronization technique is the **FIFO queue**.
- The use of a FIFO is similar to the mailbox, but allows buffering, which is storing data in a first come first served manner.
- For an input device, an interrupt occurs when new input data are available, the ISR reads the data from the input device, and puts the data in the FIFO.
- Whenever the main program is idle, it will attempt to get data from the FIFO.
- If data were to exist, that data will be processed.

Input device

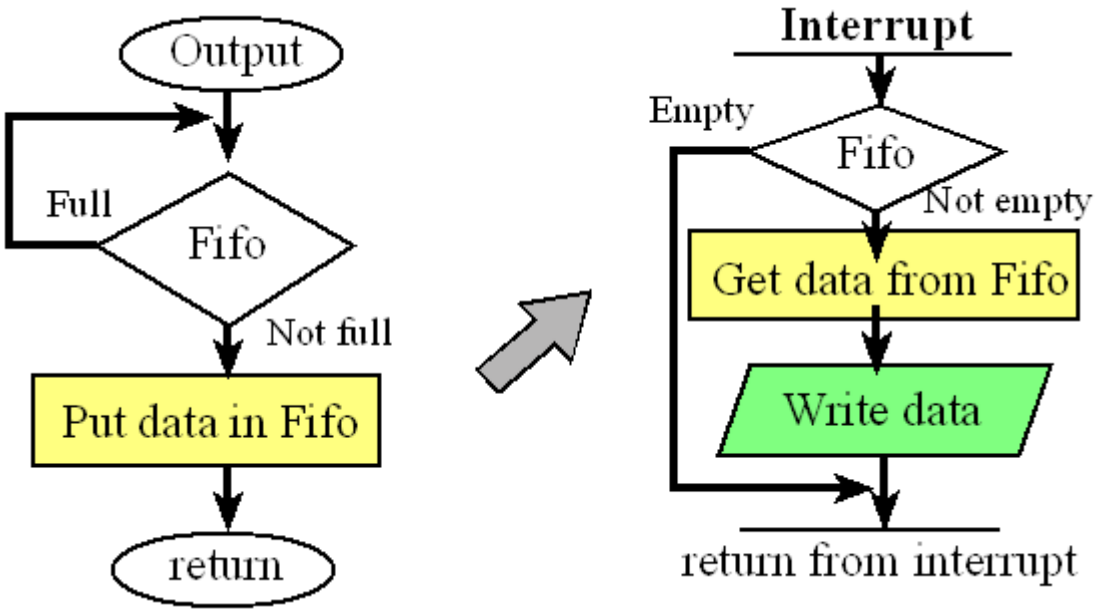


For an input device we can use a FIFO to pass data from the ISR to the main program.

FIFO – output device

- For an output device, the main program puts data into the FIFO whenever it wishes to perform output.
- This data is buffered, and if the system is properly configured, the FIFO never becomes full and the main program never actually waits for the output to occur.
- An interrupt occurs when the output device is idle, the ISR gets from the FIFO and write the data to the output device.
- Whenever the ISR sees the FIFO is empty, it could cause the output device to become idle.
- The direction of the big arrows in Figures 12.2 and 12.3 signify the direction of data flow in these buffered I/O examples.

Output device



Periodic interrupts

- There are other types of interrupt that are not an input or output.
- For example we will configure the computer to request an interrupt on a periodic basis.
- This means an interrupt handler will be executed at fixed time intervals.
- This periodic interrupt will be essential for the implementation of real-time data acquisition and real-time control systems.
- For example if we are implementing a digital controller that executes a control algorithm 100 times a second, then we will set up the internal timer hardware to request an interrupt every 10 ms.

Periodic interrupts

- The interrupt service routine will execute the digital control algorithm and then return to the main thread.
- In a similar fashion, we will use periodic interrupts to perform analog input and/or analog output.
- For example if we wish to sample the ADC 100 times a second, then we will set up the internal timer hardware to request an interrupt every 10 ms.
- The interrupt service routine will sample the ADC, process (or save) the data, and then return to the main thread.

NVIC on the ARM Cortex-M Processor

- On the ARM Cortex-M processor, **exceptions** include resets, software interrupts and hardware interrupts.
- Interrupts on the Cortex-M are controlled by the Nested Vectored Interrupt Controller (NVIC).
- Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located.
- Vectors are stored in ROM at the beginning of memory.

Interrupt sources

- Program 12.1 shows the first few vectors as defined in the **Startup.s** file.
- **DCD** is an assembler pseudo-op that defines a 32-bit constant.
- ROM location 0x0000.0000 has the initial stack pointer, and location 0x0000.0004 contains the initial program counter, which is called the reset vector.
- It points to a function called the reset handler, which is the first thing executed following reset.
- There are up to 240 possible interrupt sources and their 32-bit vectors are listed in order starting with location 0x0000.0008.
- We can attach ISRs to interrupts by writing the ISRs as regular assembly subroutines or C functions with no input or output parameters and editing the **Startup.s** file to specify those functions for the appropriate interrupt.

Some interrupt sources

```
EXPORT  __Vectors

__Vectors                       ; address   ISR
        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
        DCD     GPIOPortA_Handler   ; 0x00000040 GPIO Port A
        DCD     GPIOPortB_Handler   ; 0x00000044 GPIO Port B
        DCD     GPIOPortC_Handler   ; 0x00000048 GPIO Port C
        DCD     GPIOPortD_Handler   ; 0x0000004C GPIO Port D
        DCD     GPIOPortE_Handler   ; 0x00000050 GPIO Port E
        DCD     UART0_Handler       ; 0x00000054 UART0
        DCD     UART1_Handler       ; 0x00000058 UART1
        DCD     SSI0_Handler        ; 0x0000005C SSI
        DCD     I2C0_Handler        ; 0x00000060 I2C
```

Typical interrupt service routine.

- Program shows that the syntax for an ISR looks like a function with no parameters.
- Notice that each ISR (except for SysTick) must acknowledge the interrupt in software by clearing the flag that caused the interrupt.
- In the program, we assume the interrupt was caused by an edge on PF4, so writing to the ICR register will clear trigger flag 4.

```
void GPIOPortF_Handler(void){
  GPIO_PORTF_ICR_R = 0x10; // ack, clear interrupt flag4
  // stuff
}
```

Activating an interrupt source

- To activate an interrupt source we need to set its priority and enable that source in the NVIC.
- This activation is in addition to the arm and enable steps.
- Table 12.1 lists some of the interrupt sources available on the TM4C family of microcontrollers.
- Interrupt numbers 0 to 15 contain the faults, software interrupt and SysTick; these interrupts will be handled differently from interrupts 16 and up.

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	**PWM0Fault_Handler**	**NVIC_PRI2_R**	15 – 13
0x00000068	26	10	**PWM0_Handler**	**NVIC_PRI2_R**	23 – 21
0x0000006C	27	11	**PWM1_Handler**	**NVIC_PRI2_R**	31 – 29
0x00000070	28	12	**PWM2_Handler**	**NVIC_PRI3_R**	7 – 5
0x00000074	29	13	**Quadrature0_Handler**	**NVIC_PRI3_R**	15 – 13
0x00000078	30	14	**ADC0_Handler**	**NVIC_PRI3_R**	23 – 21
0x0000007C	31	15	**ADC1_Handler**	**NVIC_PRI3_R**	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

Priority registers

- Table shows some of the priority registers on the NVIC.
- Each register contains an 8-bit priority field for four devices.
- On the TM4C microcontrollers, only the top three bits of the 8-bit field are used.
- This allows us to specify the interrupt priority level for each device from 0 to 7, with 0 being the highest priority.
- If a request of equal or lower priority is generated while an ISR is being executed, that request is postponed until the ISR is completed.
- In particular, those devices that need prompt service should be given high priority.

Priority registers

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

The LM3S/TM4C NVIC registers. Each register is 32 bits wide. Bits not shown are zero.

Enabling UART0

- There are five enable registers NVIC_EN0_R through NVIC_EN4_R.
- The 32 bits in register NVIC_EN0_R control the IRQ numbers 0 to 31 (interrupt numbers 16 – 47).
- In Table 12.1 we see UART0 is IRQ=5.
- To enable UART0 interrupts we set bit 5 in NVIC_EN0_R, see Table 12.3.

Interrupt enable registers

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

Table 12.3. Some of the TM4C NVIC interrupt enable registers. There are five such registers defining 139 interrupt enable bits.

Enabling UART1

- The 32 bits in NVIC_EN1_R control the IRQ numbers 32 to 63 (interrupt numbers 48 – 79).
- In Table 12.1 we see UART2 is IRQ=33.
- To enable UART interrupts we set bit 1 (33-32=1) in NVIC_EN1_R, see Table 12.3.
- Not every interrupt source is available on every TM4C microcontroller, so you will need to refer to the data sheet for your microcontroller when designing I/O interfaces.
- Writing zeros to the NVIC_EN0_R through NVIC_EN4_R registers has no effect.
- To disable interrupts we write ones to the corresponding bit in the NVIC_DIS0_R through NVIC_DIS4_R register.

SysTick interrupt

- When the SysTick counter goes from 1 to 0,
  the **Count** flag in the **NVIC_ST_CTRL_R** register is set,
  triggering an interrupt.

Nested interrupt

- A **nested interrupt** occurs when a higher priority interrupt suspends an ISR.
- The lower priority interrupt will finish after the higher priority ISR completes.

Priority

- **Priority** determines the order of service when two or more requests are made simultaneously.
- Priority also allows a higher priority request to suspend a lower priority request currently being processed.
- Usually, if two requests have the same priority, we do not allow them to interrupt each other.
- NVIC assigns a priority level to each interrupt trigger.
- This mechanism allows a higher priority trigger to interrupt the ISR of a lower priority request.
- Conversely, if a lower priority request occurs while running an ISR of a higher priority trigger, it will be postponed until the higher priority service is complete.

Edge-triggered Interrupts

- Synchronizing software to hardware events requires the software to recognize when the hardware changes states from busy to done.
- Many times the busy to done state transition is signified by a rising (or falling) edge on a status signal in the hardware.
- For these situations, we connect this status signal to an input of the microcontroller, and we use edge-triggered interfacing to configure the interface to set a flag on the rising (or falling) edge of the input.
- Using edge-triggered interfacing allows the software to respond quickly to changes in the external world.

Edge-triggered Interrupts

- If we are using busy-wait synchronization, the software waits for the flag.
- If we are using interrupt synchronization, we configure the flag to request an interrupt when set.
- Each of the digital I/O pins on the TM4C family can be configured for edge triggering.
- Table 12.4 shows the registers needed to set up edge triggering for Port A.

Some TM4C port A registers.

Address	7	6	5	4	3	2	1	0	Name
$4000.43FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTA_DATA_R
$4000.4400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTA_DIR_R
$4000.4404	IS	IS	IS	IS	IS	IS	IS	IS	GPIO_PORTA_IS_R
$4000.4408	IBE	IBE	IBE	IBE	IBE	IBE	IBE	IBE	GPIO_PORTA_IBE_R
$4000.440C	IEV	IEV	IEV	IEV	IEV	IEV	IEV	IEV	GPIO_PORTA_IEV_R
$4000.4410	IME	IME	IME	IME	IME	IME	IME	IME	GPIO_PORTA_IM_R
$4000.4414	RIS	RIS	RIS	RIS	RIS	RIS	RIS	RIS	GPIO_PORTA_RIS_R
$4000.4418	MIS	MIS	MIS	MIS	MIS	MIS	MIS	MIS	GPIO_PORTA_MIS_R
$4000.441C	ICR	ICR	ICR	ICR	ICR	ICR	ICR	ICR	GPIO_PORTA_ICR_R
$4000.4420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTA_AFSEL_R
$4000.4500	DRV2	DRV2	DRV2	DRV2	DRV2	DRV2	DRV2	DRV2	GPIO_PORTA_DR2R_R
$4000.4504	DRV4	DRV4	DRV4	DRV4	DRV4	DRV4	DRV4	DRV4	GPIO_PORTA_DR4R_R
$4000.4508	DRV8	DRV8	DRV8	DRV8	DRV8	DRV8	DRV8	DRV8	GPIO_PORTA_DR8R_R
$4000.450C	ODE	ODE	ODE	ODE	ODE	ODE	ODE	ODE	GPIO_PORTA_ODR_R
$4000.4510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTA_PUR_R
$4000.4514	PDE	PDE	PDE	PDE	PDE	PDE	PDE	PDE	GPIO_PORTA_PDR_R
$4000.4518	SLR	SLR	SLR	SLR	SLR	SLR	SLR	SLR	GPIO_PORTA_SLR_R
$4000.451C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTA_DEN_R
$4000.4524	CR	CR	CR	CR	CR	CR	CR	CR	GPIO_PORTA_CR_R
$4000.4528	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	GPIO_PORTA_AMSEL_R
	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0	
$4000.452C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTA_PCTL_R
$4000.4520	LOCK (32 bits)								GPIO_PORTA_LOCK_R

Table 12.4. Some TM4C port A registers. We will clear PMC bits to use edge triggered interrupts.

Configuration for interrupts

- All of digital I/O pins can be configured as an edge-triggered input.
- To use any of the features for a digital I/O port, we first enable its clock in the Run Mode Clock Gating Control Register 2 (RCGC2).
- For each bit we wish to use we must set the corresponding **DEN** (Digital Enable) bit.
- To use edge triggered interrupts we will clear the corresponding bits in the **PCTL** register, and we will clear bits in the **AFSEL** (Alternate Function Select) register.
- We clear **DIR** (Direction) bits to make them input.
- On the TM4C123, only pins PD7 and PF0 need to be unlocked.
- We clear bits in the **AMSEL** register to disable analog function.

Configuration for interrupts

- To configure an edge-triggered pin, we first enable the clock on the port and configure the pin as a regular digital input.
- Clearing the **IS** (Interrupt Sense) bit configures the bit for edge triggering.
- If the **IS** bit were to be set, the trigger occurs on the level of the pin.
- Since most busy to done conditions are signified by edges, we typically trigger on edges rather than levels.
- Next we write to the **IBE** (Interrupt Both Edges) and **IEV** (Interrupt Event) bits to define the active edge.
- We can trigger on the rising, falling, or both edges, as listed in Table 12.5.

DIR	AFSEL	PMC	IS	IBE	IEV	IME	Port mode
0	0	0000	0	0	0	0	Input, falling edge trigger, busy wait
0	0	0000	0	0	1	0	Input, rising edge trigger, busy wait
0	0	0000	0	1	-	0	Input, both edges trigger, busy wait
0	0	0000	0	0	0	1	Input, falling edge trigger, interrupt
0	0	0000	0	0	1	1	Input, rising edge trigger, interrupt
0	0	0000	0	1	-	1	Input, both edges trigger, interrupt

Table 12.5. Edge-triggered modes.

Raw interrupt status

- The hardware sets an **RIS** (Raw Interrupt Status) bit (called the trigger) and the software clears it (called the acknowledgement).
- The triggering event listed in Table 12.5 will set the corresponding **RIS** bit in the **GPIO_PORTA_RIS_R** register regardless of whether or not that bit is allowed to request an interrupt.
- In other words, clearing an **IM** bit disables the corresponding pin's interrupt, but it will still set the corresponding **RIS** bit when the interrupt would have occurred.

Raw interrupt status

- The software can acknowledge the event by writing ones to the corresponding **IC** (Interrupt Clear) bit in the **GPIO_PORTA_IC_R** register.
- The **RIS** bits are read only, meaning if the software were to write to this register, it would have no effect.
- For example, to clear bits 2, 1, and 0 in the **GPIO_PORTA_RIS_R** register, we write a 0x07 to the **GPIO_PORTA_IC_R** register.
- Writing zeros into **IC** bits will not affect the **RIS** bits.

Pull up and pull down resistors

- For input signals we have the option of adding either a pull-up resistor or a pull-down resistor.
- If we set the corresponding **PUE** (Pull-Up Enable) bit on an input pin, the equivalent of a 13 kΩ to 30 kΩ resistor to +3.3 V power is internally connected to the pin.
- Similarly, if we set the corresponding **PDE** (Pull-Down Enable) bit on an input pin, the equivalent of a 13 kΩ to 35 kΩ resistor to ground is internally connected to the pin.
- We cannot have both pull-up and a pull-down resistor, so setting a bit in one register automatically clears the corresponding bit in the other register.

Pull up and pull down resistors

- A typical application of pull-up and pull-down mode is the interface of simple switches.
- Using these modes eliminates the need for an external resistor when interfacing a switch.
- Compare the interfaces on Port A to the interfaces on Port B illustrated in Figure 12.4.
- The PA2 and PA3 interfaces will use software-configured internal resistors, while the PB2 and PB3 interfaces use actual resistors.
- The PA2 and PB2 interfaces in Figure 12.4a) implement negative logic switch inputs, and the PA3 and PB3 interfaces in Figure 12.4b) implement positive logic switch inputs.

Example

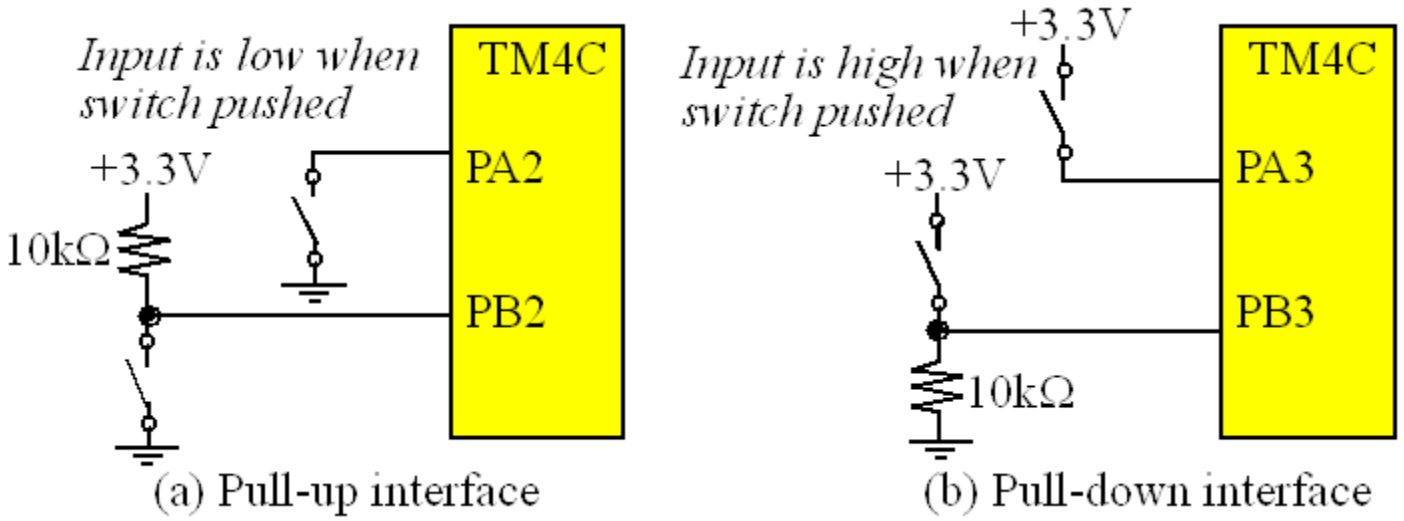


Figure 12.4. Edge-triggered interfaces can generate interrupts on a switch touch.

Busy-wait interface

- Using edge triggering to synchronize software to hardware centers around the operation of the trigger flags, **RIS**.
- A busy-wait interface will read the appropriate **RIS** bit over and over, until it is set.
- When the **RIS** bit is set, the software will clear the **RIS** bit (by writing a one to the corresponding **IC** bit) and perform the desired function.

Interrupt Synchronization

- With interrupt synchronization, the initialization phase will arm the trigger flag by setting the corresponding **IM** bit.
- In this way, the active edge of the pin will set the **RIS** and request an interrupt.
- The interrupt will suspend the main program and run a special interrupt service routine (ISR).
- This ISR will clear the **RIS** bit and perform the desired function.
- At the end of the ISR it will return, causing the main program to resume.

Interrupt conditions

- In particular, five conditions must be simultaneously true for an edge-triggered interrupt to be requested:
- The trigger flag bit is set (RIS)
- The arm bit is set (IME)
- The level of the edge-triggered interrupt must be less than BASEPRI
- The edge-triggered interrupt must be enabled in the NVIC_EN0_R
- The I bit, bit 0 of the special register PRIMASK, is 0

Configuring the pin for interrupts

- We will begin with a simple example that counts the number of rising edges on Port F bit 4 (Program 12.4).
- The initialization requires many steps.
- (a) The clock for the port must be enabled.
- (b) The global variables should be initialized.
- (c) The appropriate pins must be enabled as inputs.

Configuring the pin for interrupts

- (d) We must specify whether to trigger on the rise, the fall, or both edges. In this case we will trigger on the rise of PF4.
- (e) It is good design to clear the trigger flag during initialization so that the first interrupt occurs due to the first rising edge after the initialization has been run. We do not wish to count a rising edge that might have occurred during the power up phase of the system.
- (f) We arm the edge-trigger by setting the corresponding bits in the **IM** register.
- (g) We establish the priority of Port F by setting bits 23 – 21 in the **NVIC_PRI7_R** register as listed in Table 9.2.
- (h) We activate Port F interrupts in the NVIC by setting bit 30 in the **NVIC_EN0_R** register, Table 12.3. There is no need to unlock PF4.

Configuration code

```
void EdgeCounter_Init(void){
  SYSCTL_RCGC2_R |= 0x00000020; // (a) activate clock for port F
  FallingEdges = 0;             // (b) initialize count and wait for clock
  GPIO_PORTF_DIR_R &= ~0x10;    // (c) make PF4 in (built-in button)
  GPIO_PORTF_AFSEL_R &= ~0x10;  //     disable alt funct on PF4
  GPIO_PORTF_DEN_R |= 0x10;     //     enable digital I/O on PF4
  GPIO_PORTF_PCTL_R &= ~0x000F0000; //  configure PF4 as GPIO
  GPIO_PORTF_AMSEL_R &= ~0x10;  //    disable analog functionality on PF4
  GPIO_PORTF_PUR_R |= 0x10;     //     enable weak pull-up on PF4
  GPIO_PORTF_IS_R &= ~0x10;     // (d) PF4 is edge-sensitive
  GPIO_PORTF_IBE_R &= ~0x10;    //     PF4 is not both edges
  GPIO_PORTF_IEV_R &= ~0x10;    //     PF4 falling edge event
  GPIO_PORTF_ICR_R = 0x10;      // (e) clear flag4
  GPIO_PORTF_IM_R |= 0x10;      // (f) arm interrupt on PF4
  NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|0x00A00000; // (g) priority 5
  NVIC_EN0_R = 0x40000000;      // (h) enable interrupt 30 in NVIC
  EnableInterrupts();           // (i) Enable global Interrupt flag (I)
}
```

DIR	AFSEL	PMC	IS	IBE	IEV	IME	Port mode
0	0	0000	0	0	0	0	Input, falling edge trigger, busy wait
0	0	0000	0	0	1	0	Input, rising edge trigger, busy wait
0	0	0000	0	1	-	0	Input, both edges trigger, busy wait
0	0	0000	0	0	0	1	Input, falling edge trigger, interrupt
0	0	0000	0	0	1	1	Input, rising edge trigger, interrupt
0	0	0000	0	1	-	1	Input, both edges trigger, interrupt

Interrupt routine

```
void GPIOPortF_Handler(void){
  GPIO_PORTF_ICR_R = 0x10;      // acknowledge flag4
  FallingEdges = FallingEdges + 1;
}
int main(void){
   EdgeCounter_Init(); // initialize GPIO Port F interrupt
   while(1){
      WaitForInterrupt();
   }
}
```

Acknowledge the interrupt

- All ISRs must acknowledge the interrupt by clearing the trigger flag that requested the interrupt.
- For edge-triggered PF4, the trigger flag is bit 4 of the **GPIO_PORTF_RIS_R** register.
- This flag can be cleared by writing a 0x10 to **GPIO_PORTF_ICR_R**.

Polled and vectored interrupts

- If two or more triggers share the same vector, these requests are called **polled interrupts**, and the ISR must determine which trigger generated the interrupt.
- If the requests have separate vectors, then these requests are called **vectored interrupts** and the ISR knows which trigger caused the interrupt.

Switch bounce

- One of the problems with switches is called **switch bounce**.
- Many inexpensive switches will mechanically oscillate for up to a few milliseconds when touched or released.
- It behaves like an underdamped oscillator.
- These mechanical oscillations cause electrical oscillations such that a port pin will oscillate high/low during the bounce.
- In some cases this bounce should be removed.
- Recognize a switch transition, disarm interrupts for 10ms, and then rearm after 10 ms.
- Alternatively, we could record the time of the switch transition.
- If the time between this transition and the previous transition is less than 10ms, ignore it.
- If the time is more than 10 ms, then accept and process the input as a real event.

SysTick Periodic Interrupts

- One application of periodic interrupts is called “intermittent polling” or “periodic polling”.
- Figure 12.5 shows busy wait side by side with periodic polling.
- In busy-wait synchronization, the main program polls the I/O devices continuously.
- With periodic polling, the I/O devices are polled on a regular basis (established by the periodic interrupt.)
- If no device needs service, then the interrupt simply returns.

SysTick Periodic Interrupts

- If the polling period is Δt, then on average the interface latency will be ½Δt, and the worst case latency will be Δt.
- Periodic polling is appropriate for low bandwidth devices where real-time response is not necessary.
- This method frees the main program to perform other functions.
- We use periodic polling if the following two conditions apply:
- 1. The I/O hardware cannot generate interrupts directly
- 2. We wish to perform the I/O functions in the background

Accurate sampling

- For a data acquisition system, it is important to establish an accurate sampling rate.
- The time in between ADC samples must be equal and known in order for the digital signal processing to function properly.
- Similarly for microcontroller-based control systems, it is important to maintain a periodic rate for reading data from the sensors and outputting commands to the actuators.

SysTick timer

- The SysTick timer is a simple way to create periodic interrupts.
- A periodic interrupt is one that is requested on a fixed time basis.
- This interfacing technique is required for data acquisition and control systems, because software servicing must be performed at accurate time intervals.

SysTick timer configuration

- Table 12.6 shows the SysTick registers used to create a periodic interrupt.
- SysTick has a 24-bit counter that decrements at the bus clock frequency.
- Let f_{BUS} be the frequency of the bus clock, and let n be the value of the **RELOAD** register.
- The frequency of the periodic interrupt will be $f_{BUS}/(n+1)$.
- First, we clear the **ENABLE** bit to turn off SysTick during initialization.
- Second, we set the **RELOAD** register.
- Third, we write any value to **NVIC_ST_CURRENT_R** to clear the counter.
- Lastly, we write the desired mode to the control register, **NVIC_ST_CTRL_R**.

SysTick registers

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	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
$E000ED20	TICK	0	PENDSV	0	DEBUG	0	NVIC_SYS_PRI3_R

SysTick interrupt configuration

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

SysTick interrupt configuration

- In this way, the SysTick counter (**CURRENT**) is continuously decrementing.
- If the **RELOAD** value is *n*, then the SysTick counter operates at modulo *n*+1 (…*n*, *n*-1, *n*-2 … 1, 0, *n*, *n*-1, …).
- In other words, it rolls over every *n*+1 counts.
- Thus, the **COUNT** flag will be set every *n*+1 counts.
- Program 12.5 shows a simple example of SysTick.
- 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.

Systick initialization code

```
volatile unsigned long Counts=0;
void SysTick_Init(unsigned long period){
  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 SysTick with core clock
and interrupts
// enable interrupts after all initialization is finished
}
```

SysTick registers

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	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
$E000ED20	TICK	0	PENDSV	0	DEBUG	0	NVIC_SYS_PRI3_R

Systick handler

```
void SysTick_Handler(void){
  GPIO_PORTF_DATA_R ^= 0x04;       // toggle PF2
  Counts = Counts + 1;
}
```

Systick main code

```
int main(void){ // running at 16 MHz
  SYSCTL_RCGC2_R |= 0x00000020; // activate port F
  Counts = 0;
  GPIO_PORTF_DIR_R |= 0x04;   // make PF2 output (PF2 built-in
LED)
  GPIO_PORTF_AFSEL_R &= ~0x04;// disable alt funct on PF2
  GPIO_PORTF_DEN_R |= 0x04;   // enable digital I/O on PF2
  GPIO_PORTF_PCTL_R =
(GPIO_PORTF_PCTL_R&0xFFFFF0FF)+0x00000000;
  GPIO_PORTF_AMSEL_R = 0;     // disable analog functionality on PF
  SysTick_Init(16000);        // initialize SysTick timer, every 1ms
  EnableInterrupts();      // enable after everything initialized
  while(1){                // interrupts every 1ms, 500 Hz flash
    WaitForInterrupt();
  }
}
```

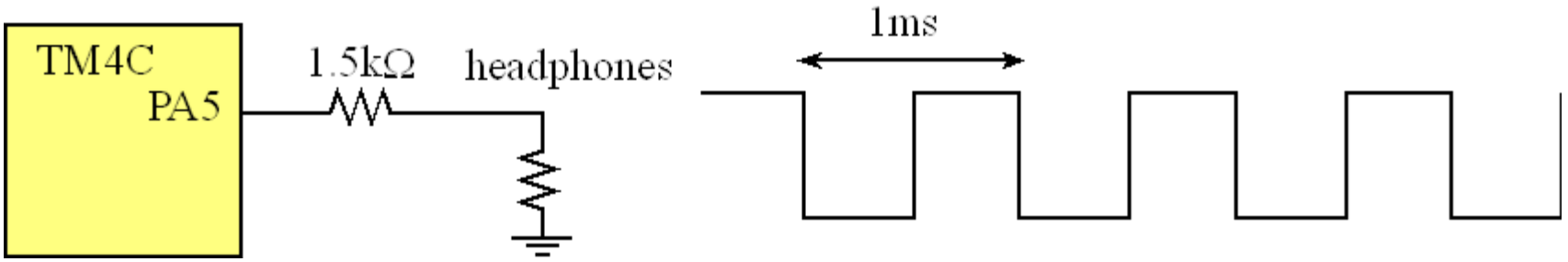
Design an interface 32 Ω speaker and use it to generate a soft 1 kHz sound.

- To make sound we need to create an oscillating wave.
- In this example, the wave will be a simple square wave.
- At 3.3V, a 32 Ω speaker will require a current of about 100 mA.
- The maximum the TM4C123 can produce on an output pin is 8 mA.
- If we place a resistor in series with the head phones, then the current will only be 3.3V/(1500+32Ω) = 2.2mA.
- To generate the 1 kHz sound we need a 1 kHz square wave.
- There are many good methods to generate square waves.
- In this example we will implement one of the simplest methods: period interrupt and toggle an output pin in the ISR.

Sound interface

- To generate a 1 kHz wave we will toggle the PA5 pin every 500 µs.
- We will assume the PLL is active and the system is running at 80 MHz.
- We wish to initialize the SysTick to interrupt with a period of 500 µs.
- The correct value for reload is 39999 ((500µs/12.5ns)-1).
- If the bus frequency were to be 16 MHz, we would set the reload value to be 7999 ((500µs/62.5ns)-1).
- Since this sound wave output is a real time signal, we set its priority to highest level, which is 0.

Figure



Code

```
void Sound_Init(void){ unsigned long volatile delay;
  SYSCTL_RCGC2_R |= 0x00000001; // activate port A
  delay = SYSCTL_RCGC2_R;
  GPIO_PORTA_AMSEL_R &= ~0x20;      // no analog
  GPIO_PORTA_PCTL_R &= ~0x00F00000; // regular function
  GPIO_PORTA_DIR_R |= 0x20;     // make PA5 out
  GPIO_PORTA_DR8R_R |= 0x20;    // can drive up to 8mA out
  GPIO_PORTA_AFSEL_R &= ~0x20;  // disable alt funct on PA5
  GPIO_PORTA_DEN_R |= 0x20;     // enable digital I/O on PA5
  NVIC_ST_CTRL_R = 0;           // disable SysTick during setup
  NVIC_ST_RELOAD_R = 39999;     // reload value for 500us (assuming 80MHz)
  NVIC_ST_CURRENT_R = 0;        // any write to current clears it
  NVIC_SYS_PRI3_R = NVIC_SYS_PRI3_R&0x00FFFFFF; // priority 0
  NVIC_ST_CTRL_R = 0x00000007;  // enable with core clock and interrupts
  EnableInterrupts();
}
void SysTick_Handler(void){
  GPIO_PORTA_DATA_R ^= 0x20;     // toggle PA5
}
```