EE 3171 Lecture 9

I/O Synchronization

How is I/O Synchronized?

Polled I/O

- Processor enters a *polling loop* to read the ready flag
- When the flag is set, then processor accesses port
- Key idea: processor waits for the device

Interrupt-Driven I/O

- Ready flag is connected to an Interrupt pin on processor
- When Pin = 1, the processor
  - interrupts the current program
  - calls a handler routine to access the port
  - then returns to the previous program

BROOKINS 5/19
Thanks to Esther Dossett Evansville, Indiana
HE SHOULD BE COMING INTO VIEW ANYTIME NOW.
Write to: Pluggers P. O. Box 29347 Henrico, VA 23242
©2012 Tribune Media Services, Inc. Brookins Art, LLC
The highlight of a plugger's day is the arrival of the mailman.

Lecture 9 Concepts

I/O Synchronization Schemes

- Polling
- Interrupts
- DMA

Implementing I/O Synchronization on the Tiva C.

How is I/O Synchronized?

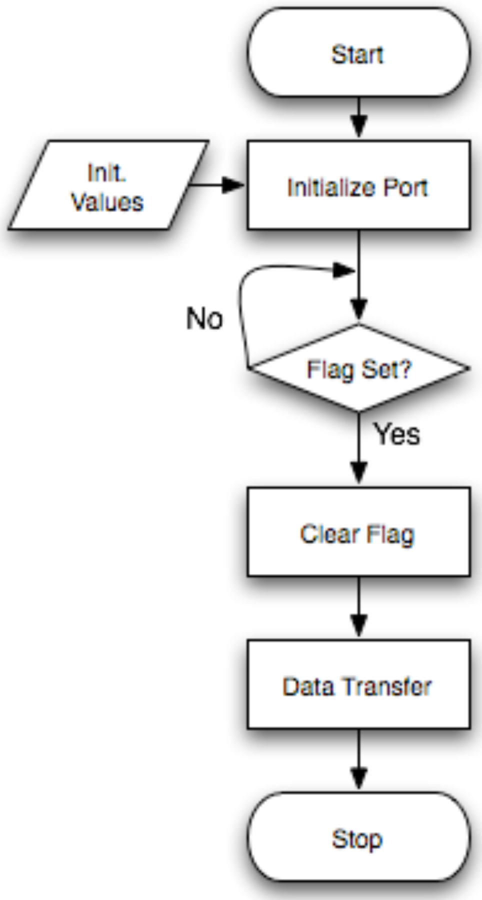
Polled I/O

- Processor enters a *polling loop* to read the ready flag
- When the flag is set, then processor accesses port
- Key idea: processor waits for the device

Interrupt-Driven I/O

- Ready flag is connected to an Interrupt pin on processor
- When Pin = 1, the processor
  - interrupts the current program
  - calls a handler routine to access the port
  - then returns to the previous program.

Polled I/O


Start
Init. Values
Initialize Port
No
Flag Set?
Yes
Clear Flag
Data Transfer
Stop

A Simplified I/O Port

Before we dig into the details of the GPIO ports on the Tiva, let's start with a simplified port (shown at right).

The `DIR` register controls the direction of the tri-state pins. 1=output, 0=input.

The `DATA` register provides read/write access to the pins.

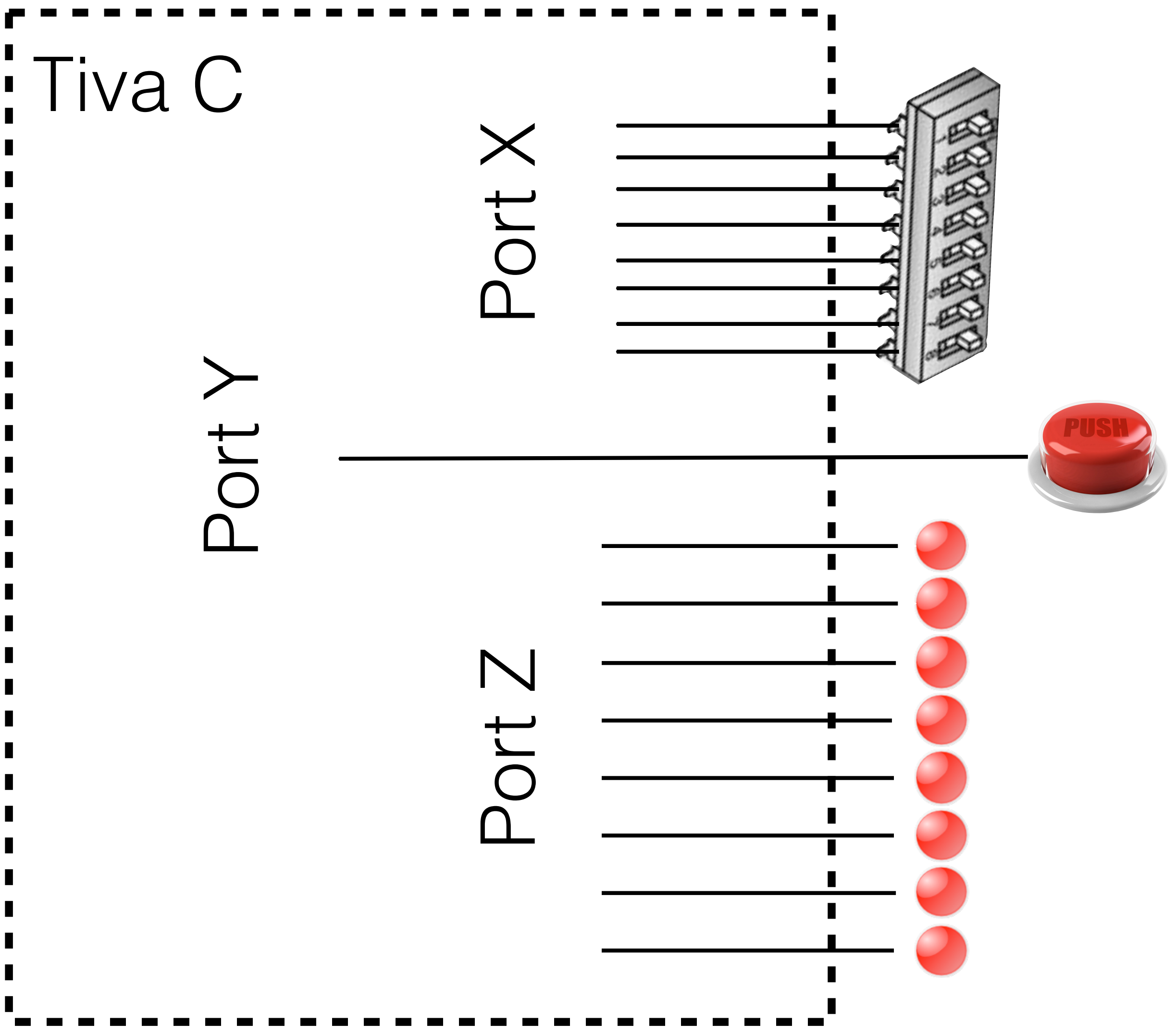
The `FLAG` register indicates if a rising edge was detected on an input pin of the data register.

For example, `F5` corresponds to pin 5 of the data register.

Write a 1 to a flag bit to clear it.

DIR	I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O
DATA	D	D	D	D	D	D	D	D
FLAG	F7	F6	F5	F4	F3	F2	F1	F0

Polled I/O Example



At the top are DIP switches.

Down below is a bank of 8 LEDs.

Polled I/O Example

Program Function

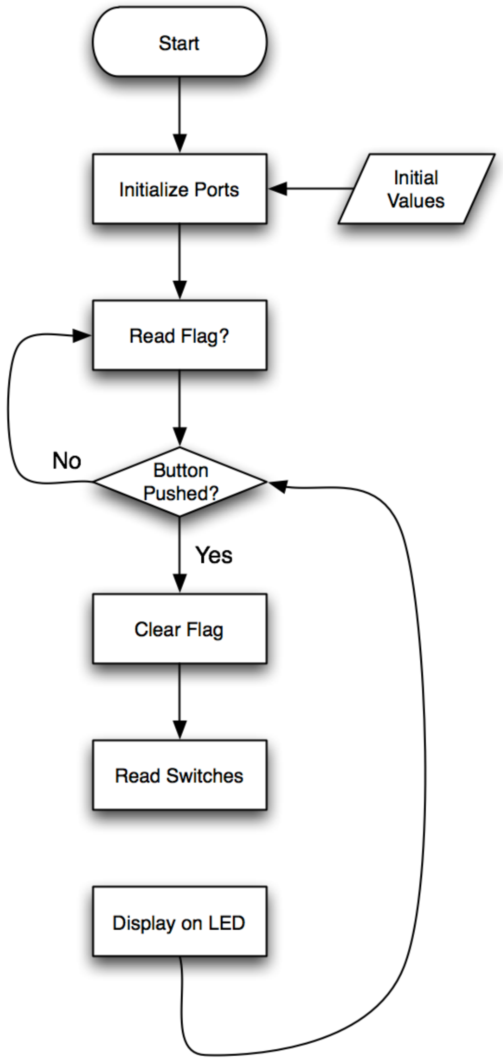
User sets a value on 8 of the DIP switches.

Then uses a button connected to Port Y (Bit 7) to indicate the processor should read the switches.

The program then displays the same value from the DIP switches on the LEDs.

Lather, rinse and repeat (forever).

Polled I/O Example



Polling Loop - Simple Port

There's almost nothing to talk about!

Configure the ports to be the proper direction.

Read flag bit **F7** of Port Y over and over until it is set.

Clear the flag.

Read the data and display on the LEDs.

Repeat forever.

```
#include <stdint.h>
#include <stdbool.h>
#include "registers.h"

int main()
{
    // Configuration
    PORTX_DIR_R &= 0x00;
    PORTY_DIR_R &= 0x7F;
    PORTZ_DIR_R |= 0xFF;

    while (1)
    {
      // Polling loop
      while (0x80 != (0x80 & PORTY_FLAG_R)) ;

      // Clear the flag
      PORTY_FLAG_R |= 0x80;

      // Read data & display on LEDs
      PORTZ_DATA_R = PORTX_DATA_R;
    }
}
```

Now to the TM4C123

In order to show this example for our processor, we need to review a few important registers.

The same basic registers from our “simple” port exist on the TM4C123, but are much more complicated because they are much more powerful and flexible.

Equivalencies:

> Simple DATA — `GPIODATA`
>
> Simple DIR — `GPIODIR`
>
> Simple FLAG — `GPIOIS`, `GPIOIBE`, `GPIOIEV`, `GPIOIM`, `GPIORIS`, `GPIOMIS`, `GPIOICR`

You might be thinking, “What’s all this interrupt stuff? I thought we were polling!” It’s just unfortunate naming conventions.

Let’s look back at these registers to remember what they do.

GPIO Registers

`GPIODATA`

Read and write data from/to the pins

`GPIODIR`

Change the direction of the tristate pins (1 = output, 0 = input)

`GPIOIS`

Change the trigger for interrupts to either edge-sensitive (0) or level-sensitive (1)

`GPIOIBE`

Configure the port to interrupt on both edges (1) or a single edge (0)

`GPIOIEV`

Interrupts on rising edge (1) or falling edge (0) on pins

GPIO Registers

`GPIOIM`

Ignore potential interrupts (0) or process the interrupt (1)

`GPIORIS`

Indicates if an interrupt has been triggered on a pin (1) or not (0)

`GPIOMIS`

Indicates if an interrupt has been masked (0) or has occurred (1)

`GPIOICR`

Write a 1 to clear an interrupt (declare it serviced)

Configuration

What values go into the Tiva GPIO registers to yield the same behavior as our "simple" port?

`DIR` register is the same.

To get the `FLAG` behavior:

```
GPIOIS = 0x00; // Interrupt on edges, not levels

GPIOIBE = 0x00; // Only one kind of edge

GPIOIEV = 0xFF; // Rising edges, by the way

GPIOIM = 0x00; // Don’t give me interrupts
```

What about the others?

They aren't for configuration, but runtime monitoring.

Equivalent Configuration

The old configuration is commented out.

The new configuration makes the Tiva port work just like our simple port.

Great.

Now let's write a polling loop.

```c
#include <stdint.h>
#include <stdbool.h>
#include "tm4c123gh6pm.h"

int main()
{
    // Configuration
    // PORTX_DIR_R &= 0x00;
    GPIO_PORTX_DIR_R &= 0x00;
    // PORTY_DIR_R &= 0x7F;
    GPIO_PORTY_DIR_R &= 0x7F;
    // PORTZ_DIR_R |= 0xFF;
    GPIO_PORTZ_DIR_R |= 0xFF;

    // Only configure Port Y, since it's the
    // only flags we look at
    GPIO_PORTY_IS_R  = 0x00; // Interrupt on edges, not levels
    GPIO_PORTY_IBE_R = 0x00; // Only one kind of edge
    GPIO_PORTY_IEV_R = 0xFF; // Rising edges, by the way
    GPIO_PORTY_IM_R  = 0x00;
```

The Tiva Polling Loop

Again, the old values are commented out so you can see them side-by-side.

```
while (1)
{
  // Polling loop
  // while (0x80 != (0x80 & PORTY_FLAG_R)) ;
  while (0x80 != (0x80 & GPIO_PORTY_RIS_R)) ;

  // Clear the flag
  // PORTY_FLAG_R |= 0x80;
  GPIO_PORTY_ICR_R |= 0x80;

  // Read data & display on LEDs
  // PORTZ_DATA_R = PORTX_DATA_R;
  GPIO_PORTZ_DATA_R = GPIO_PORTX_DATA_R;
}
```

Really Important Note

All of the clock initializations are missing from this sample code. You would need to add those in order to make this code work.

Of course, you would also have to magically create ports X, Y and Z too.

TivaWare?

Could you do this example using the TivaWare functions too?

Yes, but I frankly find it really tedious. That being said, here we go…

TivaWare Slide 1

```
// Clocks
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOX);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOY);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOZ);

SysCtlDelay(1);

// Configuration
// GPIO_PORTX_DIR_R &= 0x00;
GPIOPinTypeGPIOInput(GPIO_PORTX_BASE, GPIO_PIN_7 | GPIO_PIN_6 |
                                      GPIO_PIN_5 | GPIO_PIN_4 |
                                      GPIO_PIN_3 | GPIO_PIN_2 |
                                      GPIO_PIN_1 | GPIO_PIN_0);
// GPIO_PORTY_DIR_R &= 0x7F;
GPIOPinTypeGPIOInput(GPIO_PORTY_BASE, GPIO_PIN_7);

// GPIO_PORTZ_DIR_R |= 0xFF;
GPIOPinTypeGPIOOutput(GPIO_PORTZ_BASE, GPIO_PIN_7 | GPIO_PIN_6 |
                                       GPIO_PIN_5 | GPIO_PIN_4 |
                                       GPIO_PIN_3 | GPIO_PIN_2 |
                                       GPIO_PIN_1 | GPIO_PIN_0);


// Only configure Port Y, since it's the
// only flags we look at
// GPIO_PORTY_IS_R  = 0x00; // Interrupt on edges, not levels
// GPIO_PORTY_IBE_R = 0x00; // Only one kind of edge
// GPIO_PORTY_IEV_R = 0xFF; // Rising edges, by the way
// GPIO_PORTY_IM_R  = 0x00;
GPIOIntTypeSet(GPIO_PORTY_BASE, GPIO_PIN_7, GPIO_RISING_EDGE);
```

Full Disclosure: The TivaWare code is NOT a direct translation. There are some subtle differences, but nothing that changes the functionality.

TivaWare Slide 2

```c
while (1)
{
  // Polling loop
  // while (0x80 != (0x80 & GPIO_PORTY_RIS_R)) ;
  while (GPIO_INT_PIN_7 != GPIOIntStatus(GPIO_PORTY_BASE, false)) ;

  // Clear the flag
  // GPIO_PORTY_ICR_R |= 0x80;
  GPIOIntClear(GPIO_PORTY_BASE, GPIO_INT_PIN_7);

  // Read data & display on LEDs
  // GPIO_PORTZ_DATA_R = GPIO_PORTX_DATA_R;
  uint32_t dataIn = GPIOPinRead(GPIO_PORTZ_BASE, GPIO_PIN_7 | GPIO_PIN_6 |
                                                 GPIO_PIN_5 | GPIO_PIN_4 |
                                                 GPIO_PIN_3 | GPIO_PIN_2 |
                                                 GPIO_PIN_1 | GPIO_PIN_0);

  GPIOPinWrite(GPIO_PORTX_BASE, GPIO_PIN_7 | GPIO_PIN_6 |
                                GPIO_PIN_5 | GPIO_PIN_4 |
                                GPIO_PIN_3 | GPIO_PIN_2 |
                                GPIO_PIN_1 | GPIO_PIN_0,
                                (uint8_t) dataIn);
}
```

The “Problem”

Processor spends enormous amounts of time waiting for the I/O device to be ready

- Human = seconds or tenths of seconds
- Disk ≈ 10 ms for 1st word
    - @ 1GHz, 10 ms = 10 million clock cycles
- Disk ≈ 0.1 ms between words
    - @ 1GHz, 0.1 ms = 100,000 clock cycles

CPU spends all that time in the polling loop

- It is not doing useful work

Interrupt-Driven I/O

Problem with polling:

- CPU spends most of its time waiting for the device.

Solution:

- Let device interrupt CPU when device is ready.
- CPU drops what it was doing and services the port.
- CPU returns to its original task.

Interrupt Service Routine (ISR) routine that services the I/O port

- Resembles a subroutine call
- Done when device is ready
- Also called an “interrupt handler”

Interrupts - The View from 10,000 Feet

The processor changes the Fetch-Decode-Execute cycle to become Fetch-Decode-Execute-CheckPendingInterrupts, looking at some kind of central repository of pending interrupts before fetching the next instruction.

You must take all of the code that you had for doing the data transfer *after* the polling loop and drop it in a (slightly) specially-formatted function.

Replace the polling loop entirely with an infinite “do nothing” loop (`while(1) {}`).

> Or, put the processor to “sleep” (if available on your device).

When the interrupt is pending, the processor will automatically call your function (the *ISR*).

> You should never, ever, ever, ever directly call the ISR in your code.

The Four Steps to Functioning Interrupts

1. Write your subroutine as an ISR.
2. Tell the processor how to find the ISR.
3. Enable the specific interrupt.
4. Globally enable interrupts.

It is EXTREMELY important that you remember these 4 steps!

Writing a Proper ISR

What does it mean to write a “proper” ISR?

Bottom Line: An ISR is just a C function.

There are just a few slight differences.

Here’s the short list:

Returns `void` and takes a `void` parameter list.

Clears the flag that caused the interrupt.

Performs the specific actions required to deal with the interrupt and very little (or nothing) else.

Communicates with the main program via global variables.

Processor Responsibilities

Specifically, the Cortex-M4 looks at several registers in the NVIC (Nested Vectored Interrupt Controller) module. If they are non-zero, there is a pending interrupt.

> These bits are set automatically by the modules in the processor when specific events occur.

When an interrupt is pending, the processor must:

> Identify which interrupt is pending
>
> Prepare to call the ISR
>
> Actually call the appropriate ISR
>
> Recover from the ISR

Preparing to Invoke the ISR

- Identify the originator of the interrupt
- Find the address of the ISR
- Push the *process state* on the stack
  - Program Counter
  - Program Status - e.g. **PSR**, Registers
  - Most CPUs don't save general purpose registers
    - Takes a lot of time
    - Let the ISR push any registers it needs

The Tiva only pushes a handful of registers: R0—R3, R12, LR, PSR **and** PC.

The idea here is to be able to service the interrupt and return to the main program seamlessly.

Determining the Interrupter

Generally, each bit in the pending interrupt repository is assigned to a particular interrupt source.

Usually this is part of the processor’s design.

> For example, the interrupts for the GPIO ports on the Tiva are interrupts 0—4.

Often tightly coupled with this is an identifier that helps the processor know how to find the appropriate ISR.

> The GPIO vector numbers are 16—20.

Getting the Address of the ISR

- Method 0: Single, fixed address (Nios II approach)
  - Let the global exception handler figure it out.
  - Simple and flexible, but slow.
- Method 1: Simple static Vectored interrupts (Tiva C approach)
  - Vector Table = List of all handler addresses
    - Each entry points to one handler
    - The actual handler can be anywhere in memory
  - Use IRQ pin as index into vector table
    - Fetch table entry specified by pin number
    - Load that value into PC
    - Fetch (should be first instruction of ISR)
  - Problem: Devices sharing a pin share a handler (solved by method 0)

More Getting the Address of the ISR

Method 2: Direct Bus Vectored Interrupts

- Does not use a vector table
- The device sends its handler address to the CPU
- The processor asserts `INTA` (interrupt ACK) and the device sends its address
- CPU latches the address and then starts executing there.

Problems:

- The device needs to know the address of its ISR
- So you have to get it there somehow.
- Takes an extra bus cycle.

Last Getting the Address

Method 3: Indirect Vectored Interrupts

- Uses a vector table
- Device send offset into the vector table.
    - Not the actual address.
- CPU gets the ISR address from the vector table.
- Advantages:
    - Allows OS flexibility is hander placement
    - Does not require the device to know an address.
    - Clean merging of internal and external exceptions
    - Flexible priority
- Popular for bus-based interrupts.

Static Vector Table

Put *address* of ISR in a fixed memory location

- Tiva C Interrupt Vectors are somewhere in ROM
- At power-on, the table is at address `0x0000.0000`
- Can be relocated by putting a new address into the `VTABLE` register.

IRQ vector number points to one entry in the Vector Table

Vector Table entry points to ISR (the address of the first instruction)

When the interrupt “fires”, this entry from the table is moved into the PC.

The table is always there, but it is the programmer’s responsibility to populate it.

Step 2!

Handling Multiple Interrupts

- Multiple Sequential Interrupts: Temporarily disable interrupts
    - Most processors will ignore further interrupts whilst processing one interrupt
        - Other interrupts remain pending and are rechecked after first interrupt has been processed
    - Other Interrupts handled in sequence as they occur
- Multiple Concurrent Interrupts: Define priorities
    - Low priority interrupts can be interrupted by higher priority IRQs
        - When higher priority interrupt has been processed, processor returns to previous interrupt

Multiple Interrupts - Sequential

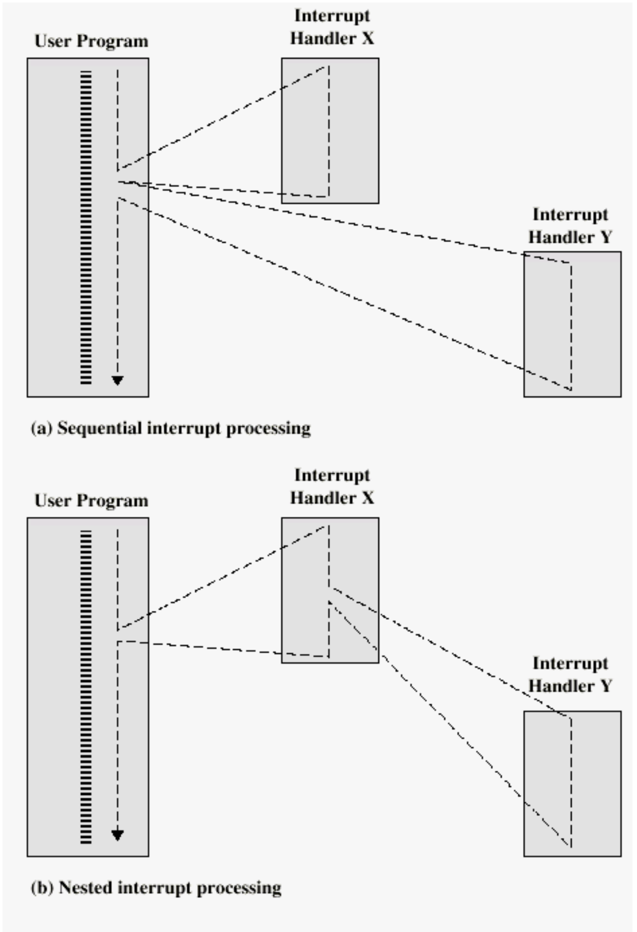


Figure 3.13 Transfer of Control with Multiple Interrupts

Multiple Interrupts – Nested

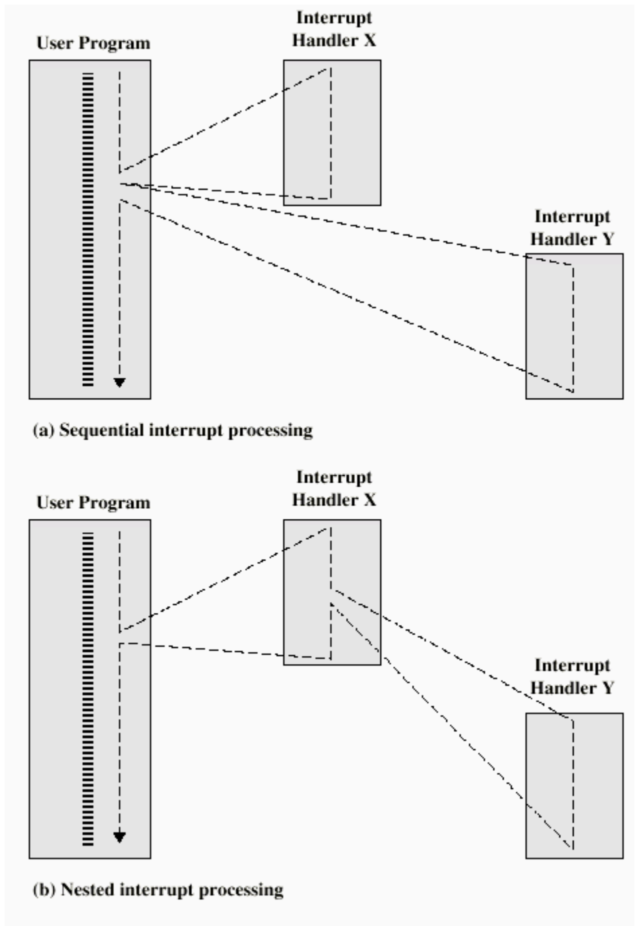


Figure 3.13 Transfer of Control with Multiple Interrupts

The Tiva handles interrupts through an NVIC — Nested Vector Interrupt Controller. It even has special hooks for making this process seamless.

Recovering from an IRQ

- Initiated by Return from Interrupt instruction
  - Last instruction in Interrupt Service Routine
    - Tiva C = `BX LR` (in ASM)
    - Still just a `return;` in C.
  - Undo all stack ops done by the interrupt
  - The next instruction the CPU fetches will be the next instruction of the previous task.

This instruction isn't exclusively for returning from ISRs.

A Terminology Note

The ARM Cortex-M4 differentiates between *exceptions* and *interrupts*.

> But only sort of. All asynchronous events are considered *exceptions*.
>
> Asynchronous events generated by external hardware are called *interrupts*.

There are two non-maskable external interrupts (NMI).

All the GPIO devices can cause interrupts.

The internal devices (ADC, SSI, UART, PWM and so on) can cause interrupts under a variety of conditions.

Specific Enables

Recall three tightly related registers in the GPIO module:

```
GPIOIM, GPIORIS, GPIOMIS
```

This is a pattern that exists in all of the I/O modules of the Tiva.

The relationship:

`RIS` is just a flag register. When a thing happens, a bit gets set here.

If you want a bit set in the pending interrupt repository, you need to set the appropriate bit in `IM`.

`MIS` is just the logical AND of `RIS` and `IM`.

Global Enables

Global IRQ Enable

Bit 0 of `FAULTMASK` register = "`FAULTMASK`" bit

Two instructions control this bit

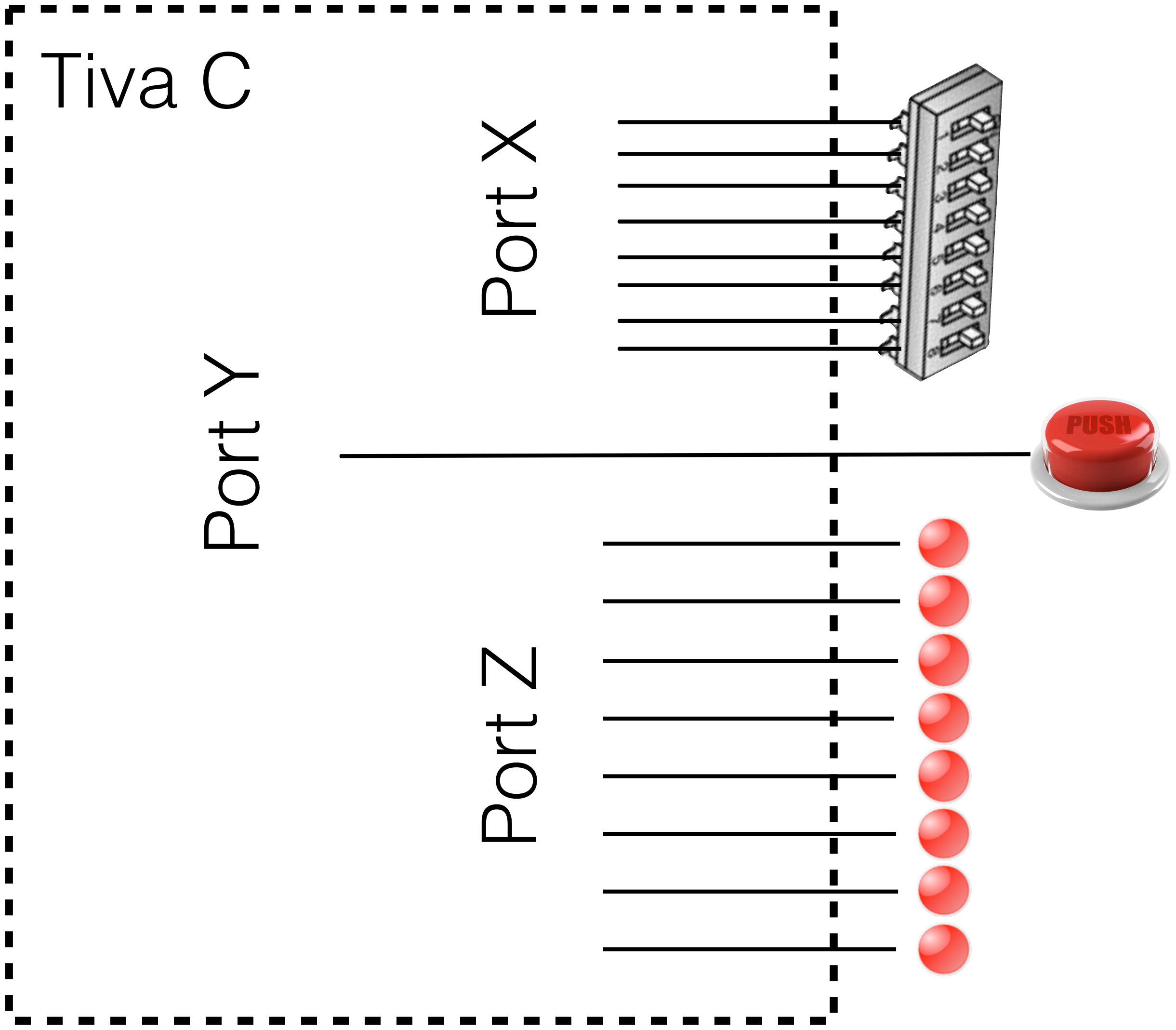
`CPSIE f` ⇒ make `FAULTMASK` = `0` ⇒ enable IRQ

`CPSID f` ⇒ make `FAULTMASK` = `1` ⇒ disable IRQ

Using the TivaWare, we use the `IntMasterDisable()` and `IntMasterEnable()` functions.

There is no DRA function to do this. All you can do is use *inline assembly*, a method for writing raw assembly language in the middle of C programs.

IRQ-Driven I/O Example



IRQ-Driven I/O Example

Same Problem, Different Synchronization

User sets a value on 8 of the DIP switches.

Then uses a button connected to Port Y (Bit 7) to indicate the processor should read the switches.

The program then displays the same value from the DIP switches on the LEDs.

Lather, rinse and repeat (forever).

Simple I/O with Interrupts

We need to make our simple I/O port a little more complex by adding a specific enable for interrupts.

Setting some bit `I`*`x`* means that when the flag `F`*`x`* is set, it should cause an interrupt.

We need a global enable too, so let's invent a magic function `GlobalIntEnable()` (and `GlobalIntDisable()` too).

Let us also declare that there is an array called `VectorTable[]` that holds function pointers as the static vector table.

DIR	I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O
DATA	D	D	D	D	D	D	D	D
FLAG	F7	F6	F5	F4	F3	F2	F1	F0
INT	I7	I6	I5	I4	I3	I2	I1	I0

Simple Interrupts

Step 1: Write the ISR.

It is short and sweet.

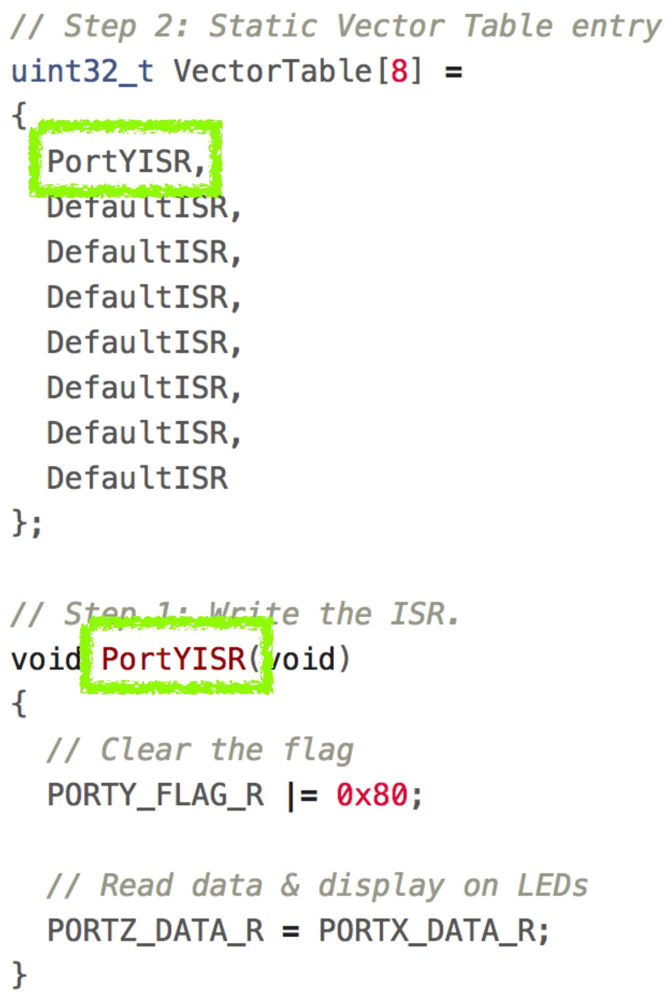
It accepts and returns `void`.

It clears the flag.

Step 2: Vector table entry.

Don't get hung up on the syntax for this. It's not important.

However, notice that the entry in the table has the same name as the ISR. This ***IS*** important.



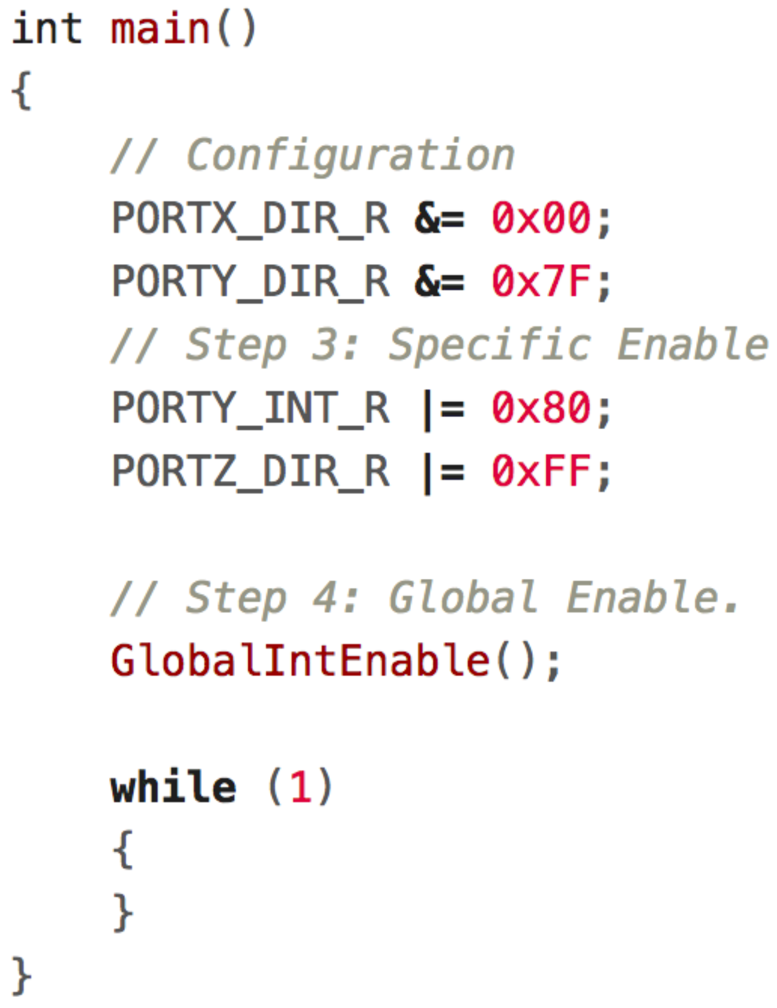
```
// Step 2: Static Vector Table entry
uint32_t VectorTable[8] =
{
  PortYISR,
  DefaultISR,
  DefaultISR,
  DefaultISR,
  DefaultISR,
  DefaultISR,
  DefaultISR,
  DefaultISR
};

// Step 1: Write the ISR.
void PortYISR(void)
{
  // Clear the flag
  PORTY_FLAG_R |= 0x80;

  // Read data & display on LEDs
  PORTZ_DATA_R = PORTX_DATA_R;
}
```

More Simple Interrupts

Notice that there really isn’t much left in the `main()`.

Configure the port and then wait.

<u>**VERY IMPORTANT:**</u> We never directly call `PortYISR()`.



```c
int main()
{
    // Configuration
    PORTX_DIR_R &= 0x00;
    PORTY_DIR_R &= 0x7F;
    // Step 3: Specific Enable
    PORTY_INT_R |= 0x80;
    PORTZ_DIR_R |= 0xFF;

    // Step 4: Global Enable.
    GlobalIntEnable();

    while (1)
    {
    }
}
```

Moving to the TM4C

The screenshot at right is code in the `startup_rvmdk.s` file provided by TI.

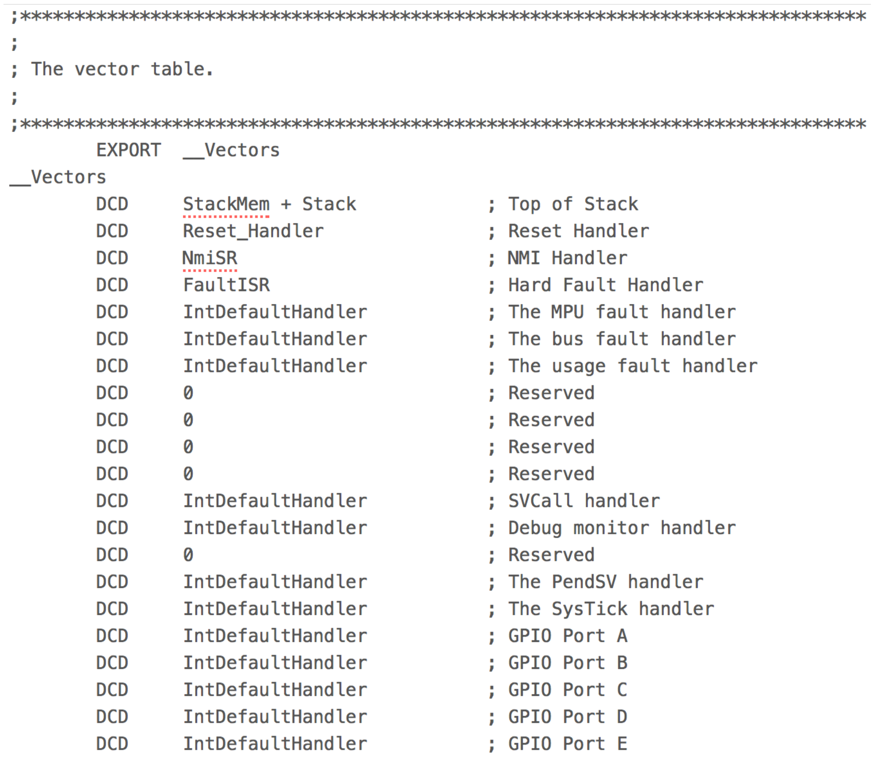
There is an entry for every single interrupt source.

Most of the them are `IntDefaultHandler`, which is just an empty infinite loop.

When you write an interrupt-based program in the lab, you must:

Find the entry labeled with the interrupt you're using and…

Replace `DefaultIntHandler` with the name of your ISR.



```
;******************************************************************************
;
; The vector table.
;
;******************************************************************************
        EXPORT  __Vectors
__Vectors
        DCD     StackMem + Stack            ; Top of Stack
        DCD     Reset_Handler               ; Reset Handler
        DCD     NmiSR                       ; NMI Handler
        DCD     FaultISR                    ; Hard Fault Handler
        DCD     IntDefaultHandler           ; The MPU fault handler
        DCD     IntDefaultHandler           ; The bus fault handler
        DCD     IntDefaultHandler           ; The usage fault handler
        DCD     0                           ; Reserved
        DCD     0                           ; Reserved
        DCD     0                           ; Reserved
        DCD     0                           ; Reserved
        DCD     IntDefaultHandler           ; SVCall handler
        DCD     IntDefaultHandler           ; Debug monitor handler
        DCD     0                           ; Reserved
        DCD     IntDefaultHandler           ; The PendSV handler
        DCD     IntDefaultHandler           ; The SysTick handler
        DCD     IntDefaultHandler           ; GPIO Port A
        DCD     IntDefaultHandler           ; GPIO Port B
        DCD     IntDefaultHandler           ; GPIO Port C
        DCD     IntDefaultHandler           ; GPIO Port D
        DCD     IntDefaultHandler           ; GPIO Port E
```

Tiva Interrupts - Plain C Version

Not really anything different here. We're just using the correct register names.

> "Correct" if there was such a port as Port Y.

Notice that Step 2 is done outside of the C file.

```c
// Step 2: In startup file

// Step 1: Write the ISR.
void PortYISR(void)
{
  // Clear the flag
  GPIO_PORTY_ICR_R |= 0x80;

  // Read data & display on LEDs
  GPIO_PORTZ_DATA_R = GPIO_PORTX_DATA_R;
}
```

Tiva Interrupts - Plain C Version

```
int main()
{
    // Configuration
    GPIO_PORTX_DIR_R &= 0x00;
    GPIO_PORTY_DIR_R &= 0x7F;
    GPIO_PORTZ_DIR_R |= 0xFF;

    // Only configure Port Y, since it's the
    // only flags we look at
    GPIO_PORTY_IS_R  = 0x00; // Interrupt on edges, not levels
    GPIO_PORTY_IBE_R = 0x00; // Only one kind of edge
    GPIO_PORTY_IEV_R = 0xFF; // Rising edges, by the way
    GPIO_PORTY_IM_R  = 0x80; // Specific Enable for PY7 interrupts.

    // Step 4: Global Enable
    __asm
    {
      MRS R0, PRIMASK;
      CPSIE i;
    }
    while (1)
    {
    }
}
```

Please do not do this. Pretty please?

I'm just being pedantic and not using ANY TivaWare functions.

How To *Really* Do It

This hasn't changed much.

Use the `GPIOIntClear()` function instead of directly accessing the `GPIOICR` register.

Skip the `GPIOPinRead()` and `GPIOPinWrite()` functions because they're a pain.

```c
// Step 2: In startup file

// Step 1: Write the ISR.
void PortYISR(void)
{
  // Clear the flag
  GPIOIntClear(GPIO_PORTY_BASE, GPIO_INT_PIN_7);

  // Read data & display on LEDs
  GPIO_PORTZ_DATA_R = GPIO_PORTX_DATA_R;
}
```

How To *Really* Do It

Actually included the clock enables.

Use DRA to do directions because the `GPIOPinTypeGPIOInput()` and `GPIOPinTypeGPIOOutput()` functions are both stupidly named and inefficient to write.

Leverage the TivaWare functions for configuring the interrupts.

<u>STILL DON’T CALL THE ISR DIRECTLY.</u>

```c
int main()
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOX);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOY);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOZ);

    SysCtlDelay(1);

    // Configuration
    GPIO_PORTX_DIR_R &= 0x00;
    GPIO_PORTY_DIR_R &= 0x7F;
    GPIO_PORTZ_DIR_R |= 0xFF;

    GPIOIntTypeSet(GPIO_PORTY_BASE, GPIO_PIN_7, GPIO_RISING_EDGE);

    // Step 3: Specific Enable
    // Oddly, BOTH of these are needed.
    GPIOIntEnable(GPIO_PORTY_BASE, GPIO_INT_PIN_7);
    IntEnable(INT_GPIOY);

    // Step 4: Global Enable
    IntMasterEnable();

    while (1)
    {
    }
}
```

Summary

Polling loops are simple, but inefficient.

Interrupts add complexity, but free up the processor to do whatever else it needs to do.

DMA is even better (the processor is uninvolved), but we'll talk more about that soon.