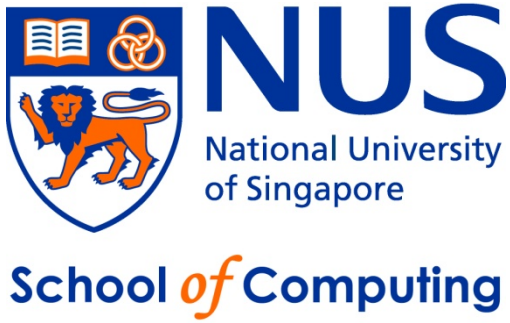
CG1112

Engineering Principles and Practices II for CEG

Week 5 Studio 1

Interrupts

colintan@nus.edu.sg



NUS National University of Singapore
School of Computing

Dealing With I/O

- **In Week 4 you learnt about programming the GPIO ports on the AVR microcontroller unit (MCU) used on the Arduino.**
- **In this lecture we will look at a related topic: Interrupts.**

Dealing With I/O

- **Presumably you are currently using your laptop/PC to view this set of slides, and maybe listen to my lecture.**
- **So your PC is busy:**
  - Rendering the gorgeous colors and wise words of these slides.
  - Playing my recording of these notes.
- **Presumably, your computer still responds to your keyboard!**
  - You can try hitting ESC or ALT-TAB to see if you can escape these notes!

Dealing With I/O

- **Question:**
  - Since your PC is busy showing these slides or playing a video (or Goat Simulator, if you're inclined towards such things):
    - ✓**HOW DOES YOUR PC KNOW WHEN YOU HAVE PRESSED A KEY??**
  - Long story short, there are two possibilities:
    - ✓**Your computer (actually, your operating system – but we'll wait till CG2271 to talk about that) continuously checks if a key has been pressed. OR**
    - ✓**The keyboard tells the CPU when a key has been pressed, and the CPU goes and reads the key.**

An Analogy

- **Consider this story:**
  - Elon the Magnificent, one time CEO of PayPal, founder of Tesla Motors and SpaceX, and Awesome Superhero ™, wants to employ you to the hedges in his Awesome Superhero ™ Secret Hideout ™ at the Basement of 1 Infinite Loop, Cupertino, CA 95014.
  - He will pay you US$25m a year, in perpetuity.
- **Would you take this job? (Say YES! C'mon trimming hedges in a basement for the Awesome Superhero ™ has to be better than reading these slides!)**

An Analogy

- **There a catch! (Aside from the fact that nobody grows hedges in basements):**
  - He will call you on your land-line phone to talk to you first (little known fact: Elon does not like mobile phones).
  - You must answer in three seconds, or the job goes to your worst enemy, and you have to pay your worst enemy US$2m.
  - You're busy with EPP2 readings and studios.
- **What would you do if:**
  - Your phone's ringer doesn't work?
  - Your phone's ringer works perfectly?

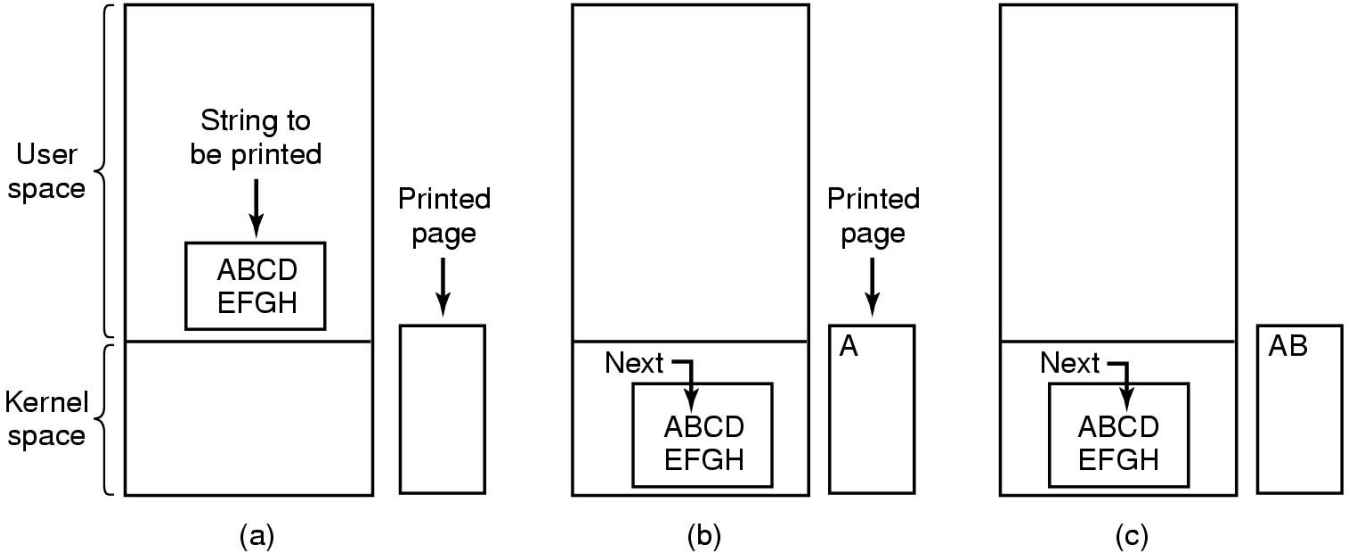
I/O Programming

- **There are 3 ways to access the I/O devices. The Arduino/AVR does not support the third method (direct memory access) so we will skip it.**
  - Polling (You with a broken ringer)
  - Interrupt-driven I/O (You with a fully functioning phone)
  - Direct Memory Access (Not you)

I/O Programming
Polling

- **This is the simplest form of I/O programming – The CPU does all the work.**
- **Consider a process that prints "ABCDEFGH" on the printer:**
  - User process acquires the printer by making a system call. This call returns an error or blocks if the printer is busy, until it is available.
  - There is an operating system (OS) that copies the string to be printed into its own buffer.

I/O Programming
Polling




```
copy_from_user(buffer, p, count);                  /* p is the kernel bufer */
for (i = 0; i < count; i++) {                      /* loop on every character */
        while (*printer_status_reg != READY) ;     /* loop until ready */
        *printer_data_register = p[i];             /* output one character */
}
return_to_user( );
```

I/O Programming
Polling

- The OS then copies character by character onto the printer's latch, and the printer prints it out.
  - **Copy the first character and advance the buffer's pointer.**
  - **Check that the printer is ready for the next character. If not, wait.**
    - *This is called "busy-waiting" or "polling".*
  - **Copy the next character. Repeat until buffer is empty.**

I/O Programming
Polling

- **Issue:**
  - It takes perhaps 10ms to print a character.
  - During this time, the CPU will be busy-waiting until the printer is done printing.
  - On a 16MHz processor like the AVR Atmega328P on the Arduino, this is equivalent to wasting 160,000 instruction cycles!

Polling Arduino Wiring Language Example

```
#define inputPin       2
…
void loop()
{
  while(1)
  {
    // When pin 2 is HIGH we know
    // data is available at
    // analog input A0.
    while(!digitalRead(inputPin))
        ;
    data=analogRead(0);
    // process data
    ...
}
```

I/O Programming
Interrupt I/O

- **Since 160,000 instructions is a large number, why not let the CPU do other stuff while the printer is busy?**
  - After the string is copied, the OS will send a character to the printer, then switch to a task.
  - When the printer is done, it will interrupt the CPU by asserting one of the "interrupt request" (IRQ) lines on the CPU.
  - This triggers a software routine called an "interrupt handler" , "interrupt service routine" or "interrupt service procedure" which will then load the next character. We will use the term "ISR" – short for "Interrupt Service Routine".

NUS National University of Singapore
School of Computing

I/O Programming
Interrupt I/O

Main Routine

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a)

Interrupt Service Routine (ISR)

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
     i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

Interrupt Programming on the Arduino Wiring Language

- **You can use attachInterrupt to specify an ISR to process interrupts.**
- **The UNO can receive interrupts on pins 2 and 3 (INT0 and INT1).**

```
attachInterrupt(digitalPinToInterrupt(pinnum),
        function, mode)
```

  - pinnum: Either 2 (pin 2) or 3 (pin 3)
  - function: ISR to call when interrupt is triggered.
  - mode: LOW, CHANGE, RISING, FALLING.

Interrupt Programming on the Arduino Wiring Language

```
#define outputPin     13
// Interrupt 0 is on pin 2
#define inputInterrupt          0

volatile unsigned int data;
volatile unsigned char flag;
void setup()
{
    pinMode(outputPin, OUTPUT);
    attachInterrupt(digitalPinToInterrupt(
           inputInterrupt),  readAnalog,
            CHANGE);
}
void readAnalog()
{
    data=analogRead(0);
    flag=1;
}
```

```
void loop()
{
    while(1)
    {
        if(flag)
           {
               processData(data);
            flag=0;
           }
        // Do other stuff
    }
}

int main()
{
    setup();
    loop();
    return 0;
}
```

NUS National University of Singapore
School of Computing

Bare-metal AVR Interrupt Programming

- **The Atmega328P MCU can process up to 26 different interrupts, partially shown below:**

Vector No	Program Address(2)	Source	Interrupts definition
1	0x0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 0
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/Coutner2 Compare Match B
10	0x0012	TIMER2_OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA	Timer/Counter0 Compare Match A

NUS National University of Singapore
School of Computing

How AVR Interrupts Work
The Vector Table

- **When an interrupt is triggered (e.g. from the timer or an external interrupt), the AVR uses a vector table (shown here) to executed the correct handler routine.**

```
Address   Labels    Code                          Comments
0x0000              jmp     RESET                 ; Reset
0x0002              jmp     INT0                  ; IRQ0
0x0004              jmp     INT1                  ; IRQ1
0x0006              jmp     PCINT0                ; PCINT0
0x0008              jmp     PCINT1                ; PCINT1
0x000A              jmp     PCINT2                ; PCINT2
0x000C              jmp     WDT                   ; Watchdog Timeout
0x000E              jmp     TIM2_COMPA            ; Timer2 CompareA
0x0010              jmp     TIM2_COMPB            ; Timer2 CompareB
0x0012              jmp     TIM2_OVF              ; Timer2 Overflow
0x0014              jmp     TIM1_CAPT             ; Timer1 Capture
0x0016              jmp     TIM1_COMPA            ; Timer1 CompareA
0x0018              jmp     TIM1_COMPB            ; Timer1 CompareB
0x001A              jmp     TIM1_OVF              ; Timer1 Overflow
0x001C              jmp     TIM0_COMPA            ; Timer0 CompareA
0x001E              jmp     TIM0_COMPB            ; Timer0 CompareB
0x0020              jmp     TIM0_OVF              ; Timer0 Overflow
0x0022              jmp     SPI_STC               ; SPI Transfer Complete
0x0024              jmp     USART_RXC             ; USART RX Complete
0x0026              jmp     USART_UDRE            ; USART UDR Empty
0x0028              jmp     USART_TXC             ; USART TX Complete
0x002A              jmp     ADC                   ; ADC Conversion Complete
0x002C              jmp     EE_RDY                ; EEPROM Ready
0x002E              jmp     ANA_COMP              ; Analog Comparator
0x0030              jmp     TWI                   ; 2-wire Serial
0x0032              jmp     SPM_RDY               ; SPM Ready
;
0x0034    RESET:    ldi     r16,high(RAMEND)      ; Main program start
0x0035              out     SPH,r16               ; Set Stack Pointer to top of RAM
0x0036              ldi     r16,low(RAMEND)
0x0037              out     SPL,r16
0x0038              sei                           ; Enable interrupts
```

Bare-metal AVR Interrupt Programming

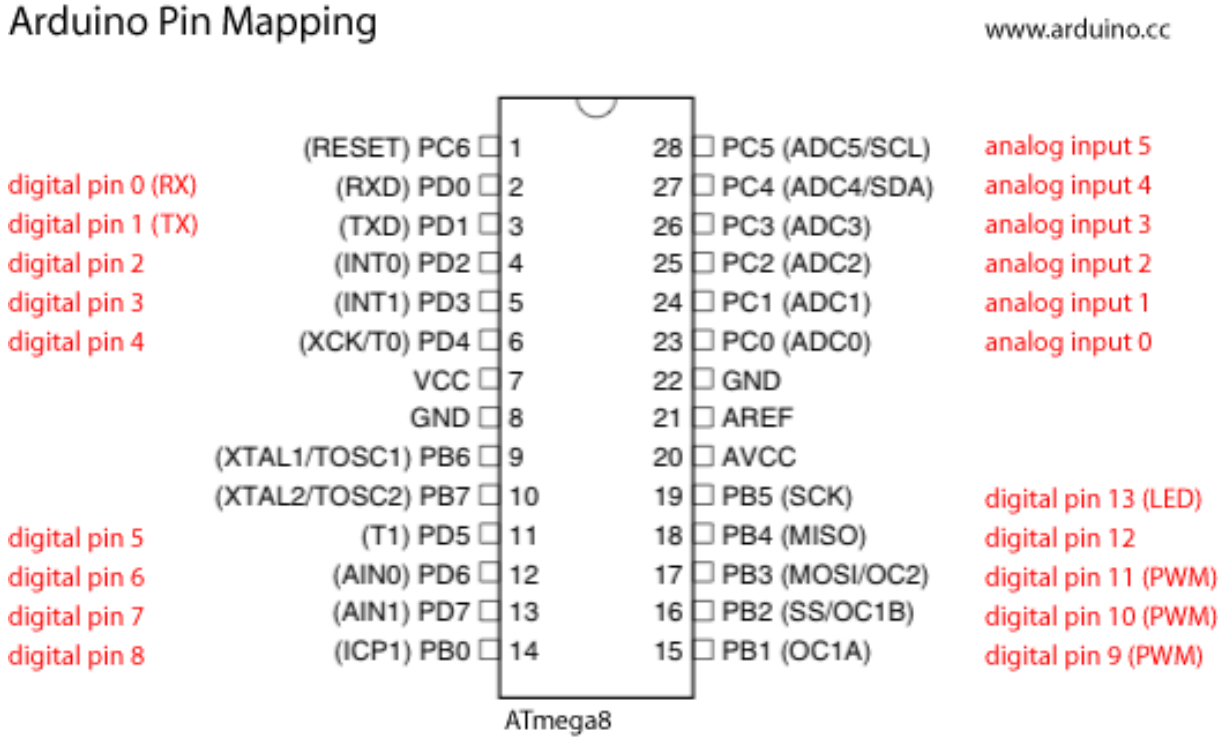
- **You can handle an interrupt request by using the ISR macro:**

```
ISR(vector)
{
    ISR Code
}
```

- **The *vector* parameter specifies the interrupt you want to process.**
  - It takes the form *interruptName_vect*
  - E.g. to handle the PCINT0 interrupt, you would specify PCINT0_vect in *vector*.

Bare-metal AVR Interrupt Programming

- **Arduino pins 2 and 3 are connected to INT0 and INT1, as shown in this diagram.**
- **In the previous table, we can see that INT0 and INT1 are external interrupts.**



NUS National University of Singapore

School of Computing

Bare-metal AVR Interrupt Programming

- **Both INT0 and INT1 can be configured in how they respond to interrupt requests:**
  - This is done by settings bits 3 and 2 for INT1, and 1 and 0 for INT0 of the EICRA register, according to the table on the right.

17.2.1. External Interrupt Control Register A

The External Interrupt Control Register A contains control bits for interrupt sense control.

Name: EICRA
Offset: 0x69
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
					ISC11	ISC10	ISC01	ISC00
Access					R/W	R/W	R/W	R/W
Reset					0	0	0	0

Bits 3:2 – ISC1n: Interrupt Sense Control 1 [n = 1:0]

The External Interrupt 1 is activated by the external pin INT1 if the SREG I-flag and the corresponding interrupt mask are set. The level and edges on the external INT1 pin that activate the interrupt are defined in the table below. The value on the INT1 pin is sampled before detecting edges. If edge or toggle interrupt is selected, pulses that last longer than one clock period will generate an interrupt. Shorter pulses are not guaranteed to generate an interrupt. If low level interrupt is selected, the low level must be held until the completion of the currently executing instruction to generate an interrupt.

Value	Description
00	The low level of INT1 generates an interrupt request.
01	Any logical change on INT1 generates an interrupt request.
10	The falling edge of INT1 generates an interrupt request.
11	The rising edge of INT1 generates an interrupt request.

Bits 1:0 – ISC0n: Interrupt Sense Control 0 [n = 1:0]

The External Interrupt 0 is activated by the external pin INT0 if the SREG I-flag and the corresponding interrupt mask are set. The level and edges on the external INT0 pin that activate the interrupt are defined in table below. The value on the INT0 pin is sampled before detecting edges. If edge or toggle interrupt is selected, pulses that last longer than one clock period will generate an interrupt. Shorter pulses are not guaranteed to generate an interrupt. If low level interrupt is selected, the low level must be held until the completion of the currently executing instruction to generate an interrupt.

Value	Description
00	The low level of INT0 generates an interrupt request.
01	Any logical change on INT0 generates an interrupt request.
10	The falling edge of INT0 generates an interrupt request.
11	The rising edge of INT0 generates an interrupt request.

Bare-metal AVR Interrupt Programming

- **By default, INT0 and INT1 are turned off. To turn them on, you need to write to bits 0 and 1 respectively of register EIMSK (External Interrupt Mask Register):**

Name: EIMSK
Offset: 0x3D
Reset: 0x00
Property: When addressing as I/O Register: address offset is 0x1D

Bit	7	6	5	4	3	2	1	0
							INT1	INT0
Access							R/W	R/W
Reset							0	0

Interrupt Masking

- **The MCU can be made to ignore almost all interrupts. This is called "interrupt masking".**
- **This is used in critical segments of code where any interruption can result in wrong execution.**
- **How to disable/enable interrupts:**
  - On Arduino Wiring Language:
    - ✓**noInterrupts() to disable all interrupts.**
    - ✓**interrupts() to enable all interrupts again.**
  - On bare-metal:
    - ✓**cli() to disable all interrupts.**
    - ✓**sei() to enable all interrupts again.**

NUS National University of Singapore
School of Computing

Bare-metal AVR Interrupt Programming

- **Example code in C: Using INT0 to switch and LED on and off.**

```
/*
 *
 * Created: 3/2/2018 3:01:16 AM
 * Author : dcstanc
 */

#include <avr/io.h>
#include <avr/interrupt.h>

static volatile int onOff=0;

/* Our setup routine */
void setup()
{
	// Set the INT0 interrupt to respond on the RISING edge.
	// We need to set bits 0 and 1
	// in EICRA to 1

	EICRA |= 0b00000011;

	// The green LED is on Pin 12, which is on PB4. Set it to OUTP
	DDRB |= 0b00010000;

	// Activate INT0
	EIMSK |= 0b00000001;

	// Ensure that interrupts are turned on.
	sei();
}
```

Bare-metal AVR Interrupt Programming

```
/* Here we declare the Interrupt Service Routine (ISR) for INT0_vect */
ISR(INT0_vect)
{
	onOff=1-onOff;
}

/* Our setup routine */
void loop()
{
	// Switch off the LED if onOff is 0, otherwise switch it on.
	if(onOff==0)
		PORTB &=0b11101111;
	else
		PORTB |= 0b00010000;
}
```

Being Volatile

- **For correct program operation, any global variable that is modified by an ISR but used by another function, must be declared to be "volatile".**
- **In our example onOff is set/cleared by the ISR but used in main, so we declare:**

```
static volatile int onOff;
```

- **The "static" keyword is optional, and it prevents other C modules from accessing/modifying onOff, which it could otherwise do simple by declaring:**

```
extern int onOff;
```

Why Volatility Matters

- **To understand volatility, we need to understand how C compiles a loop into assembly.**
- **Where we have:**

```
while(onoff == 0)
{
        … Loop Body …
}
```

- **On CPUs like the Sun SPARC and MCUs like the ARM and AVR, the CPU can only operate on small temporary storage units called "CPU registers", and not on variables directly.**

Why Volatility Matters

- **So in the code above, the CPU/MCU loads up onOff into a register using a lw (load-word) assembly instruction and generates the following code:**

```
        lw $r0, onOff ; Load onOff into register r0
loop:
        … Loop Body …
        beq $r0, 0, loop ; Jump back to loop if $ro is 0
```

- **We see that the compiler does not do lw $0, onOff on every iteration but only once.**
  - This is because memory is very slow and the lw instruction consumes a lot of CPU cycles.

Why Volatility Matters

- **The problem now is that if the ISR changes onOff, the code in the loop will never know because it never reloads onOff, and thus $r0 will never be updated.**
  - The code goes into an infinite loop.
- **By declaring the variable to be volatile, the C compiler moves the lw into the loop, like so:**

```
loop:
        lw $r0, onOff ; Load onOff into register r0
        … Loop Body …
        beq $r0, 0, loop ; Jump back to loop if $ro is 0
```

Why Volatility Matters

- **Since the code now reloads onOff into $r0 on every iteration, the loop will now always have the latest value for onOff, as set by the ISR.**
- **Downside:**
  - The loop now accesses memory every cycle, slowing down your execution greatly.
  - But that… as they say.. is life.
- **Moral:**
  - Always declare GLOBAL variables that are changed by ISRs to be volatile.

The Interrupts Studio

- **Monday / Tuesday 10/11 February 2020**
- **Four activities this week:**
  - Activity 1: Reading a switch through polling.
  - Activity 2: Reading a switch through interrupts, using the Arduino library.
  - Activity 3: Switch debouncing.
  - Activity 4: Doing bare-metal interrupt programming.