Sensor Processing

So far, our code looks something like this:

```
while(1) {
    <read some sensors>
    <respond to the sensor input>
    <read some other sensors>
    <respond to the sensor input>

}
```

Sensor Processing

- Sometimes, this is sufficient
- Other times:
  - We need to respond to certain events very quickly
  - We need to time events very carefully

Interrupts

- Hardware mechanism that allows some event to temporarily interrupt an ongoing task
- The processor then executes a small piece of code called: **interrupt handler** or **interrupt service routine** (ISR)
- Execution then continues with the original program

Some Sources of Interrupts (atmega2560)

External:

- An input pin changes state
- The UART receives a byte on a serial input

Internal:

- A clock
- Processor reset
- The on-board analog-to-digital converter completes its conversion

Interrupt Example

Suppose we are executing code from your main program:

LDS R1 (A) ← **PC**

LDS R2 (B)

CP R2, R1

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

An Example

Suppose we are executing code from your main program:

LDS R1 (A)

LDS R2 (B) ← **PC**

CP R2, R1

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

An Example

Suspect we are executing code from your main program:

LDS R1 (A)

LDS R2 (B)

CP R2, R1 ← **PC**

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

An Example

An interrupt occurs (EXT_INT1):

LDS R1 (A)

LDS R2 (B)

CP R2, R1 ← **PC**

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

An Example

Execute the interrupt handler

LDS R1 (A)

LDS R2 (B)

CP R2, R1

BRGE 3 ← remember this location

LDS R3 (D)

ADD R3, R1

STS (D), R3

An Example

Execute the interrupt handler

LDS R1 (A)

LDS R2 (B)

CP R2, R1

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

EXT_INT1:

PC

LDS R1 (G)

LDS R5 (L)

ADD R1, R2

:

RETI

An Example

Execute the interrupt handler

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
PC → LDS R5 (L)
ADD R1, R2
:
RETI

An Example

Execute the interrupt handler

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
PC → ADD R1, R2
:
RETI

An Example

Execute the interrupt handler

LDS R1 (A)
LDS R2 (B)
CP R2, R1
▶ BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
PC → :
RETI

An Example

Return from interrupt

LDS R1 (A)
LDS R2 (B)
CP R2, R1
▶ BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
:
PC → RETI

An Example

Return from interrupt

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3      <- PC
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

```
EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
   :
RETI
```

An Example

Continue execution with original

LDS R1 (A)

LDS R2 (B)

CP R2, R1

BRGE 3

LDS R3 (D) ← **PC**

ADD R3, R1

STS (D), R3

EXT_INT1:

LDS R1 (G)

LDS R5 (L)

ADD R1, R2

:

RETI

An Example

Continue execution with original

LDS R1 (A)

LDS R2 (B)

CP R2, R1

BRGE 3

LDS R3 (D)

ADD R3, R1 ← **PC**

STS (D), R3

EXT_INT1:

LDS R1 (G)

LDS R5 (L)

ADD R1, R2

:

RETI

Interrupt Routines

Generally a very small number of instructions

- We want a quick response so the processor can return to what it was originally doing
- No delays, waits, or floating point operations in the ISR…

Timer 0 Interrupt

We can configure the timer to generate an interrupt every time that the timer's counter "rolls over" from 0xFF to 0x00

Timer 0 Interrupt Example

Suppose:

- 16MHz clock
- Prescaler of 1024

How often is the interrupt generated?

Timer 0 Example

$$interval = \frac{1024 * 256}{16{,}000{,}000} = 16.384\ ms$$

Timer 0
Interrupt Service Routine (ISR)

An ISR is a type of function that is called when the interrupt is generated

```
ISR(TIMER0_OVF_vect) {
    // Toggle the LED attached to bit 0 of port B
    PORTB ^= 1;
};
```

What is the flash frequency?

Timer 0
Interrupt Service Routine (ISR)

```
ISR(TIMER0_OVF_vect) {
    // Toggle the LED attached to bit 0 of port B
    PORTB ^= 1;
};
```

What is the flash frequency?

$$frequency = \frac{16{,}000{,}000}{1024 * 256 * 2} = 30.5176\ Hz$$

Example I:
ISR Initialization in Main Program

```
// Interrupt occurs every (1024*256)/16000000 = .016384 seconds
timer0_config(TIMER0_PRE_1024);

// Enable the timer interrupt
timer0_enable();

// Enable global interrupts
sei();

 while(1) {
   // Do something else
 };
```

Timer 0 with Interrupts

This solution is particularly nice:

- “something else” does not have to worry about timing at all
- PB0 state is altered **asynchronously** from what is happening in the main program

Next Example: Timer 0 Example II

$$interval = \frac{1024 * 256}{16{,}000{,}000} = 16.384\ ms$$

How many interrupts do we need so that we toggle the state of PB0 every second?

Timer 0 Example II

How many interrupts do we need so that we toggle the state of PB0 every second?

$$counts = \frac{1000\ ms}{16.384\ ms} = 61.0352$$

We will assume 61 is close enough.

Example II: Interrupt Service Routine (ISR)

```
ISR(TIMER0_OVF_vect) {
    static uint8_t counter = 0;
    ++counter;
    if(counter == 61) {
        // Toggle output state every 61st interrupt:
        //  This means: on for ~1 second and then off for ~1 sec
        PORTB ^= 1;
        counter = 0;
    };
};
```

See Atmel HOWTO for example code (timer_demo.c)

Example II: Interrupt Service Routine (ISR)

```
uint8_t counter = 0;

ISR(TIMER0_OVF_vect) {
    ++counter;
    if(counter == 61) {
          // Toggle output state every 61st interrupt:
          //  This means: on for ~1 second and then off for ~1 sec
          PORTB ^= 1;
          counter = 0;
    };
};
```

See Atmel HOWTO for example code (timer_demo.c)

Example II: Initialization (same as before)

```
// Initialize counter
counter = 0;

// Interrupt occurs every (1024*256)/16000000 = .016384 seconds
timer0_config(TIMER0_PRE_1024);

// Enable the timer interrupt
timer0_enable();

// Enable global interrupts
sei();

 while(1) {
   // Do something else
 };
```

Timer 0 Example II

What is the flash frequency?

Timer 0 Example II

What is the flash frequency?

$$frequency = \frac{16{,}000{,}000}{1024 * 256 * 61 * 2} \approx 0.5\ Hz$$

Interrupts and Timers

Timing can often involve a cascade of multiple counters:

- prescaler (1 … 1024)
- Timer0 (256)
- Counter within an interrupt routine (any)

Each counter implements a frequency division

Generating a PWM Signal in Software

How would we do this?

Generating a PWM Signal in Software

We need:

- To produce a periodic behavior, and
- A way to specify the pulse width (or the duty cycle)

How do we implement this in code?

Generating a PWM Signal in Software

How do we implement this in code?

One way:

- Interrupt routine increments an 8-bit software counter
- When the counter is 0, turn the signal on
- When the counter reaches some "duration", turn the signal off

Our Implementation

```
volatile uint8_t duration = 42;

ISR(TIMER0_OVF_vect)
{
  static uint8_t counter = 0;

  if(counter < duration) PORTB |= 0x80;

  else PORTB &= ~0x80;

  ++counter;
}
(one bug when duration is changing)
```

Another Implementation

```
volatile uint8_t duration = 0;

ISR(TIMER0_OVF_vect)
{
  static uint8_t counter = 0;

  ++counter;
  if(counter >= duration)
      PORTB &= ~8;
  else if(counter == 0)
      PORTB |= 8;

}
```

Initialization Details

- Set up timer
- Enable interrupts
- Set duration in some way
  - In this case, we will slowly increase it

What does this implementation look like?

Initialization

```
int main(void) {
  DDRB = 0x08;
  PORTB = 0;

  duration = 0;

  // Interrupt configuration
  timer0_config(TIMER0_PRE8);  // Prescaler = 8

  // Enable the timer interrupt
  timer0_enable();

  // Enable global interrupts
  sei();
          :
```

PWM Implementation

What is the resolution (how long is one increment of “duration”)?

PWM Implementation

What is the resolution (how long is one increment of “duration”)?

- The timer0 counter (8 bits) expires every 256 clock cycles

$$t = \frac{8 \times 256}{16000000} = 0.128\, ms$$

(assuming a 16MHz clock)

PWM Implementation

What is the period of the pulse?

PWM Implementation

What is the period of the pulse?

- The 8-bit software counter expires every 256 interrupts

$$t = \frac{8 * 256 * 256}{16000000} \approx 32.77\ ms$$

Doing “Something Else”

```
        :
  unsigned int i;
  while(1) {
    for(i = 0; i < 256; ++i)
        duration = i;
        delay_ms(50);
    };
  };
}
```

NOTE: DON’T USE THIS SOFTWARE PWM FOR YOUR PROJECT

- Use hardware PWM instead

ISR Example III

```
ISR(TIMER0_OVF_vect) {
    // Toggle the LED attached to bit 0 of port B
     PORTB ^= 1;
};

int main(void){
     timer0_config(TIMER0_PRE_8);
     timer0_enable();
     sei();

  while(1) {
    // Do something else
  };
```

What is the flash frequency?

Timer 0 Example III

What is the flash frequency?

$$frequency = \frac{16{,}000{,}000}{8 * 256 * 2} \approx 3.9\ KHz$$

ISR Example III: How about this case?

```
ISR(TIMER0_OVF_vect) {
   // Toggle the LED attached to bit 0 of port B
   PORTB ^= 1;
   timer0_set(128);  // Set the timer0 counter to 128
};

int main(void){
   timer0_config(TIMER0_PRE_8);
   timer0_enable();
   sei();

 while(1) {
  // Do something else
 };
```

What is the flash frequency?

Timer 0 Example III

What is the flash frequency?

$$frequency = \frac{16{,}000{,}000}{8*128*2} \approx 7.8\ KHz$$

Different Timers

- Timer 0
- Timer 1, 3, 4, 5
- Timer 2

Interrupt Service Routines

- Should be **very** short
  - No “delays”
  - No busy waiting
  - Function calls from the ISR should be short also
  - Minimize looping
  - No “printf()”
- Communication with the main program using global variables

Interrupts, Shared Data and Compiler Optimizations

- Compilers (including ours) will often optimize code in order to minimize execution time
- These optimizations often pose no problems, but can be problematic in the face of interrupts and shared data

Shared Data and Compiler Optimizations

For example:

```
A = A + 1;
C = B + A
```

Will result in ‘A’ being fetched from memory once (into a general-purpose register) – even though ‘A’ is used twice

Shared Data and Compiler Optimizations

Now consider:

```
while(1) {
   PORTB = A;
}
```

What does the compiler do with this?

Shared Data and Compiler Optimizations

The compiler will assume that ‘A’ never changes.

This will result in assembly code that looks something like this:

```
R1 = A;  // Fetch value of A into register 1
while(1) {
   PORTB = R1;
}
```

The compiler only fetches A from memory once!

Shared Data and Compiler Optimizations

This optimization is generally fine – but consider the following interrupt routine:

```
ISR(TIMER0_OVF_vect){
  A = PIND;
}
```

Shared Data and Compiler Optimizations

This optimization is generally fine – but consider the following interrupt routine:

```
ISR(TIMER0_OVF_vect){
  A = PIND;
}
```

- The global variable ‘A’ is being changed!
- The compiler has no way to anticipate this

Shared Data and Compiler Optimizations

The fix: the programmer must tell the compiler that it is not allowed to assume that a memory location is not changing

- This is accomplished when we declare the global variable:

```
volatile uint8_t A;
```

Shared Data and Compiler Optimizations

volatile uint8_t A;

This will cause the compiler to do this:

```
while(1) {
    R1 = A;  // Fetch value of A into reg 1
    PORTB = R1;
}
```

The compiler only fetches A from memory every time it needs it!

Shared Data and Interrupts

- Recall: the data bus on the mega2560 is 8 bits wide
- A byte can be transferred between a general purpose register and memory in one cycle
- Any data structure larger than a byte requires multiple transfers

Shared Data and Interrupts

Any data structure larger than a byte requires multiple transfers

When there are interrupts: this can lead to subtle (but very real) problems

For example:

```
uint16_t a;
a = a + 5;
```

For example:

```
uint16_t a;
a = a + 5;
```

Steps:

- Transfer of the low byte from memory to a general purpose register
- Transfer of the high byte
- Addition operation (multiple steps)
- Transfer of the low byte from GP to mem
- Transfer of the high byte from GP to mem

Suppose that an ISR routine views and then modifies the variable ***a*** …

- Transfer of the low byte from memory to a general purpose register
- Transfer of the high byte
- Addition operation (multiple steps)
- Transfer of the low byte from GP to mem
- Transfer of the high byte from GP to mem

- Transfer of the low byte from memory to a general purpose register
- Transfer of the high byte
- Addition operation (multiple steps)
- Transfer of the low byte from GP to mem
- Transfer of the high byte from GP to mem

Interrupt occurs:

- ISR changes ***a***, but main program still uses old value

- Transfer of the low byte from memory to a general purpose register
- Transfer of the high byte
- Addition operation (multiple steps)
- Transfer of the low byte from GP to mem
- Transfer of the high byte from GP to mem

- Transfer of the low byte from memory to a general purpose register
- Transfer of the high byte
- Addition operation (multiple steps)
- Transfer of the low byte from GP to mem
- Transfer of the high byte from GP to mem

Interrupt occurs:

- The ISR “sees” the new value of the low byte and the old value of the high byte

Solution?

Solution?

One possibility:

- If the main program is working with a, then it can temporarily disable interrupts while it does this operation
- Note: it should not disable interrupts for very long

Turning off Interrupts

```
uint16_t a;
:
:

cli;             // Turn off interrupts
a = a + 5;
sei;             // Turn them back on
```

Shared Data Problems

- Any time that the main program and the ISR both view/operate on a global variable, the potential exists for these *shared data problems*
- Always a problem if the variable is larger than a single byte
- Some single byte variables are a problem, but not all are (it depends on how they are used)

Turning off Interrupts

- Always turn off for the shortest time possible
- There are some cases in which interrupts do not need to be turned off for things to work properly
  - E.g., our “flag” in project 4