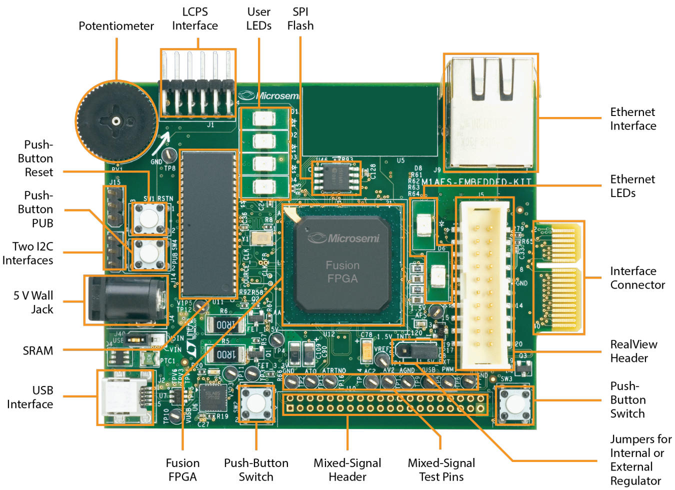



# AVR Interrupts

Kizito NKURIKIYEYEZU, Ph.D.

# Limitation of our timer programs

```c
#include <avr/io.h>
int main(){
  uint8_t count=0;
  DDRB    |= (1<<PB1)
  ASSR   |= (1<<AS0); //use ext oscillator
  TCCR0 |= (1<<CS00); //normal mode, no prescaling
  while(1) {
    while (! (TIFR & (1<<TOV0))){/*Wait until overflow occurs*/}
      TIFR |= (1<<TOV0); //clear by writing a one to TOV0
      count++;  //extend counter
      if((count % 64) == 0){//toggle PB0 every 64 overflows
        PORTB ^= (1<<PB1);
        }
   }
}
```

**LISTING 1:** This program waste resources by waiting overflow to occur

# Limitation of our timer programs

- What if we are to generate two delays at the same time?
  - Example: Toggle bit PB.5 every 1s and PB.4 every 0.5s
- What if there are some task to be done simultaneously with the timers?
  - Example: (1) read the contents of port A, process the data, and send them to port D continuously, (2) toggle bit PB.5 every 1s, and (3) PB.4 every 0.5s.

# What is an interrupt?

- An interrupt is a way for an external (or, sometimes, internal) event to pause the current processor's activity, so that it can complete a brief task before resuming execution where it left

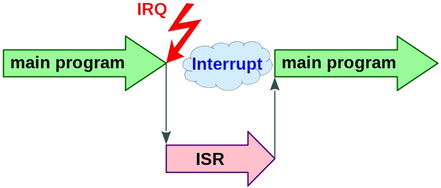



**FIG 1.** Principle of an interrupt

- For example, one can set up the processor so that it is looking for a specific external event (like a pin going high or a timer over owing) to become true, while it goes on and performs other tasks.
- When these even occur, we stop the current task, handle the event, and resume back the previous tasks.

# What is an interrupt?

- An interrupt is an exception, a change of the normal progression, or interruption in the normal flow of program execution.
- An interrupt is essentially a hardware generated function call.
- Interrupts are caused by both internal and external sources.
- An interrupt causes the normal program execution to halt and for the interrupt service routine (ISR) to be executed.
- At the conclusion of the ISR, normal program execution is resumed at the point where it was last.

In short, with an interrupt , there is no need for the processor to monitor the status of the devices and events. Instead, the events notify the processor when they occur by sending an interrupt signal to processor

# Interrupts vs. polling

```c
#include <avr/io.h>
int main(void){
    // Initialization code left out for clarity
    while (1) {
        if ((PINB & (1 << SWITCH_PIN)) == NOT_PRESSED ) {
          // Turn off the Led
          PORTB |= (1<<LED_PIN); // Set PB1 to HIGH
        }
        else {
          // Turn on the led
          PORTB &= ~(1<<LED_PIN); // Set PB1 to LOW
        }
    }
  return 0;
}
```

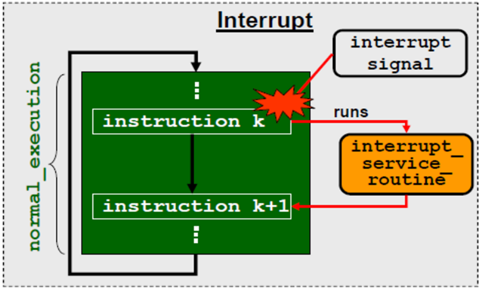
**LISTING 2:** Polling keeps check if the switch is pressed

# Interrupt vs. polling

- Using polling, the CPU must continually check the device's status
- Using interrupt:
  - A device will send an interrupt signal when needed.
  - In response, the CPU will perform an interrupt service routine, and then resume its normal execution.
  - Allows low response latency
  - Determinism (in many cases anyways!). Determinism is the consistency of the response time

**Polling**

```
while (1){
    get_device_status;
    if (service_required){
        service_routine;
    }
    normal_execution;
}
```



# Interrupt vs polling

- Polling uses a lot of CPU horsepower
  - checking whether the peripheral is ready or not
  - Wait until the peripheral is ready (but wait for how long?)
  - interrupts use the CPU only when work is to be done
- Polled code is generally messy and unstructured
  - big loop with often multiple calls to check and see if peripheral is ready
  - necessary to keep peripheral from waiting
  - ISRs concentrate all peripheral code in one place (encapsulation)
- Polled code leads to variable latency in servicing peripherals
  - whether if branches are taken or not, timing can vary
  - interrupts give highly predictable servicing latencies

# POLLING VERSUS INTERRUPT

| Criterion | Polling | Interrupt |
|---|---|---|
| Background | Checking at regular intervals | Processor is called if needed |
| Mechanism | Protocol | Mechanism |
| Servicing | CPU | Interrupt handler |
| CPU | On hold | Called if needed |
| Appearance | On regular interval | Anytime |
| Advantages | Simple program, transmission reliability, no need for additional chips | Serves multiple devices, flexible, efficient |
| Disadvantages | Standby time, time waste | More complex, time consuming |

DifferenceBetween.net

# Interrupt service routine

- Each interrupt is associated with an interrupt service routine (ISR)
- When an interrupt is invoked, the microcontroller runs the interrupt service routine.
- Generally, for every interrupt there is a fixed location in memory that holds the address of its ISR.
- The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector
- The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector
- You can find the list of all interrupts vectors of an ATmega128 on its datasheet on pages 59-60
- The datasheet also shows the priority levels of the different interrupts. The lower the address the higher is the priority level. RESET has the highest priority, and next is INT0 – the External Interrupt Request 0.

**TAB 1.** Example—Interrupts in ATmega16

| Vector No. | Program Address | Interrupt vector name | Description |
|---|---|---|---|
| 1 | $000 | RESET_vect | Reset |
| 2 | $002 | INT0_vect | External Interrupt Request 0 |
| 3 | $004 | INT1_vect | External Interrupt Request 1 |
| 4 | $006 | TIMER2_COMP_vect | Timer/Counter2 Compare Match |
| 5 | $008 | TIMER2_OVF_vect | Timer/Counter2 Overflow |
| 6 | $00A | TIMER1_CAPT_vect | Timer/Counter1 Capture Event |
| 7 | $00C | TIMER1_COMPA_vect | Timer/Counter1 Compare Match A |
| 8 | $00E | TIMER1_COMPB_vect | Timer/Counter1 Compare Match B |
| 9 | $010 | TIMER1_OVF_vect | Timer/Counter1 Overflow |
| 10 | $012 | TIMER0_OVF_vect | Timer/Counter0 Overflow |
| 11 | $014 | SPI_STC_vect | Serial Transfer Complete |
| 12 | $016 | USART_RXC_vect | USART, Rx Complete |
| 13 | $018 | USART_UDRE_vect | USART Data Register Empty |
| 14 | $01A | USART_TXC_vect | USART, Tx Complete |
| 15 | $01C | ADC_vect | ADC Conversion Complete |
| 16 | $01E | EE_RDY_vect | EEPROM Ready |
| 17 | $020 | ANA_COMP_vect | Analog Comparator |
| 18 | $022 | TWI_vect | 2-wire Serial Interface |
| 19 | $024 | INT2_vect | External Interrupt Request 2 |
| 20 | $026 | TIMER0_COMP_vect | Timer/Counter0 Compare Match |
| 21 | $028 | SPM_RDY_vect | Store Program Memory Ready |

# Types of interrupts

- Hardware interrupts
  - externally generated
  - frees up CPU from polling
- Software interrupts
  - generated by CPU instruction
  - on AVR: writing to a pin change interrupt pin configured as output triggers interrupt used to implement system calls

# What causes an interrupt an AVR MCU?

- Timers —there are at least two interrupts for each time: one for an overflow and another for the compare match
- Interrupts set for external hardware interrupts. For the ATmega128, the external interrupts are triggered by the INT7:0 pins.
- Serial communication interrupts
- Serial Peripheral Interface (SPI) interrupts
- Analog-to-digital converter (ADC) interrupts
- etc

# Why use an interrupt?

- To detect pin changes (eg. rotary encoders, button presses)
- Watchdog timer (eg. if nothing happens after 8 seconds, interrupt me)
- Timer interrupts - used for comparing/overflowing timers
- ADC conversions (analog to digital)
- EEPROM ready for use
- Flash memory ready