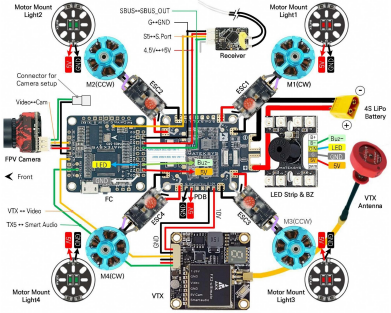
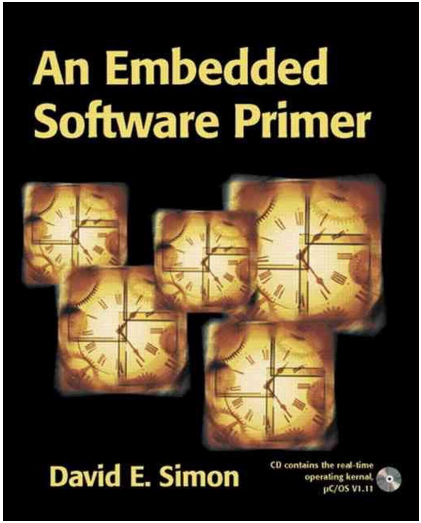



# Interrupts

Kizito NKURIKIYEYEZU, Ph.D.

# Readings

- Read Chap 4 of Simon, D. E. (1999). An Embedded Software Primer
- Topics
  - Assembly language
  - Saving and restoring context
  - Data-shared problem
  - Atomicity and critical section
  - The volatile keyword
  - Interrupts latency




[1] Readings are based on Simon, D. E. (1999). An Embedded Software Primer
[2] Bold reading section are mandatory. Other sections are suggested but not required readings

# Limitation of CPU timers

```
#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: ATMega128 code to blink an LED using a timer.**
Note that this program is wasteful since the CPU cannot do anything else until the overflow occurs.

# Limitation of the timer programs

- What if we are to generate two delays at the same time?
  - Example: Toggle bit PB5 every 1s and PB4 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?

- Interrupts are triggered when certain events occur in the hardware[1].
- e.g. when a serial chip has sent data to a microprocessor and wants it to read it from its pin, it sends an interrupt to the processor, usually by sending a signal to one of the processor's IRQ (interrupt request) pins.
- On receiving an interrupt, the microprocessor stops its current execution, saves the address of the next instruction on the stack and jumps to an interrupt service routine (ISR) or interrupt handler.
- The ISR is basically a subroutine written by the user to perform certain operations to handle the interrupt with a RETURN instruction at the end. It is a good practice to save register state and reset the interrupt in ISRs.
- ISRs are similar to a CALL except that the call to the ISR is automatically made.

[1] This lecture will not teach details of interrupts. You should have acquired this knowledge in your previous course. For a quick review, please see prof.Jonathan Valvano's lecture for details on interrupts https://users.ece.utexas.edu/~valvano/Volume1/E-Book/C12_Interrupts.htm

# 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

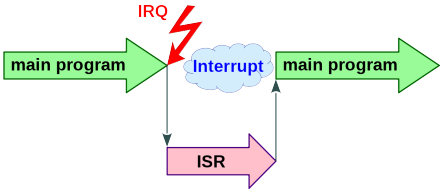



**FIG 1.** Principle of an interrupt

- 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 overflowing) 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

```
#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

# Interrupts vs. polling

```
#include<avr/io.h>
#include<avr/interrupt.h>
#define F_CPU (1000000UL * 16UL) //16MHz clock
ISR (TIMER1_OVF_vect){
  PORTD ^= (1 << PD0);
  TCNT1 = 63974;   // for 100ms at 16 MHz
}
int main(){
  DDRD = (1 << PD0);
  TCNT1 = 63974;   // for 100ms at 16 MHz
  TCCR1A = 0x00;
  TCCR1B = (1<<CS10)|(1<<CS12); // Timer mode with 1024 prescler
  TIMSK = (1 << TOIE1); // Enable timer1 overflow interrupt
  sei();    // Enable global interrupts
  while(1){/*Do nothing here! Everything is done via the ISR*/}
}
```

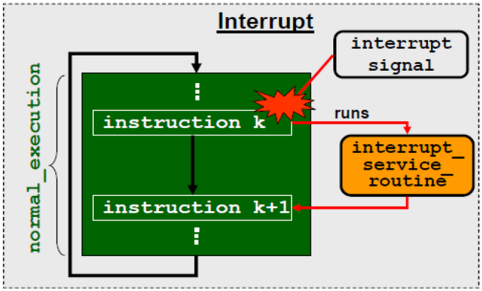
**LISTING 3: Blink an LED with an ISR**

# 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

# What causes an interrupt?

- 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

# Saving and Restoring the Context

- Due to limited number of registers, the ISRs and task codes usually have to work with same registers.
- As shown in Fig. 2, the task code has no idea of the changes taking place in registers like R1 or R2 in the ISR.

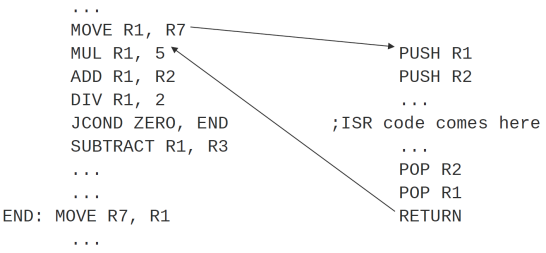



**FIG 2.** Example of assembly code of ISR

- Hence if R1 is modified by the ISR, we might get an incorrect final result.
- To solve this problem it is common practice to save the register contents onto the stack (saving the context) and restoring them at the end of the ISR

# Disabling Interrupts

- Most microprocessors allow programs to disable interrupts.
- In most cases the program can select which interrupts to disable during critical operations and which to keep enabled by writing corresponding values into a special register.
- Nonmaskable interrupts however cannot be disabled and are normally used to indicate power failures or other serious event.
- Certain processors assign priorities to interrupts, allowing programs to specify a threshold priority so that only interrupts having higher priorities than the threshold are enabled and the ones below it are disabled.

# The Shared-Data Problem

- In many cases the ISRs need to communicate with the task codes through shared variables.
- Listing 4 shows the classic shared-data problem. The code continuously monitors two temperatures and sets off an alarm if they are different.
- An ISR reads the temperatures from the hardware.The interrupt might be invoked through a timer or through the temperature sensing hardware itself.
- The code is buggy because it sets of the alarm when it should not.

```
1   static int temperatures [2];
2   void interrupt ReadTemperature(void){
3      temperatures[0] = ReadTemperatureSensor(0);
4      temperatures[1] = ReadTemperatureSesnor(1);
5   }
6   void main (void){
7     int room1Temperature, room2Temperature;
8     while(true){
9        reactor1Temperature = temperatures[0];
10       reactor2Temperature = temperatures[1];
11       if(room1Temperature != room2Temperature){
12               // Set off an alarm and alert plant operators
13       }
14    }
15  }
```

**LISTING 4: Industrial temperature monitoring**

The code is part of an industrial plant control firmware. The software monitors temperatures of various reactors. The temperatures are supposed to be equal, otherwise there is mulfunction, and the software should send alerts to the plant operators. The code, however, is buggy. Why?

# The Shared-Data Problem

- Suppose all temperatures have been 20 °C for a while
- Suppose now that the CPU execute the following code

```
1                    reactor1Temperature = temperatures[0];
```

- Suppose now that an interrupt occurs and that both temperature have increased to 21 °C degrees
- Thus, the ISR will assign 21 to all elements of the temperatures array.
- When the ISR ends , the MCU will continue with the next line of code

```
1                    reactor2Temperature = temperatures[1];
```

- Since the ISR has set all elements of the temperatures array to 21, the reactor2Temperature=21.
- However, the variable reactor1Temperature is still 20 since the code has not yet update it (Remember it was last updated just before the ISR fired!).
- When the MCU compares the two temperatures, it erroneously set off an alarm

# The Shared-Data Problem

```
1   static int temperatures [2];
2   void interrupt ReadTemperature(void){
3      temperatures[0] = ReadTemperatureSensor(0);
4      temperatures[1] = ReadTemperatureSesnor(1);
5   }
6   void main (void){
7     int room1Temperature, room2Temperature;
8     while(true){
9        if(temperatures[0] != temperatures[1]){
10                // Set off an alarm and alert plant operators
11       }
12    }
13  }
```

**LISTING 5: Industrial temperature monitoring—Much harder to catch bug**
This code is similar to that in Listing 4 except that the program does not store the sensors data into local copies. Instead, it directely compares values in the temperatures array. Where is the bug?

- Remember, C statements are terse and hide lots of information
- The assembly code in Listing 6 shows if an interrupt occurs between the two MOVES, this translates into the same problem we had before and the alarm goes off when it shouldn't have.

```
1  // Other code ommitted fro brevity
2  MOVE R1, temperatures[0]
3  MOVE R1, temperatures[1]
4  SUBTRACT R1, R2
5  //
6  //Set off the alarm
7  //
8  JCOND 0, _IS_TEMPERATURE_OK:
9  //
10 //Other instruction if the temperature matches
11 //
```

**LISTING 6: Assembly language equivalent of the code in Listing 5**

The same bug would appear if an interrupt occurs between the line that load the temperatures0 and register R1 and the line of code that loads the value temperatures1 into register R2.

## Characteristics of the shared-data bug

- Very hard to find because they do not occur every time to code runs
- They, in fact, appear random

## Solving the shared-data problem

- The problem can be solved by disabling the interrupts during the instructions that use the shared variable and re-enabling them later

```
1  while (true){
2    // Disable interrupts
3    interrupt_disable();
4    reactor1Temperature = temperatures[0];
5    reactor2Temperature = temperatures[1];
6    // Re-enable interrupts
7    interrupt_enable();
8    ...
9    // Continue with the remaining code
10   ...
11 }
```

**LISTING 7: Solution to the shared-data problem**

## Atomic operations

- An operation (or set of operations) is atomic, linearizable, indivisible or uninterruptible if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from concurrent processes.
- Hence the code between disable() and enable() above is atomic

Consider an ISR that code in Listing 9 that updates*hours*, *minutes*, and *seconds* every second through a hardware timer interrupt. Is this code atomic?

```
1 long calculateSecondsSinceMidnight (void){
2   long result =0;
3   interrupt_disable();
4   result = (long) (hours * 3600 + minutes * 60 + seconds);
5   interrupt_enable();
6   return result;
7 }
```

**LISTING 8: Function to compute the number of seconds since midnight**

# Atomic operations

```
1  static int seconds, minutes, hours;
2  void interrupt updateTime(void){
3    ++seconds;
4    if(seconds >= 60){
5      seconds = 0;
6      ++minutes;
7      if(minutes >= 60){
8        minutes = 0;
9        ++hours;
10       if(hours >= 24){
11         hours = 0;
12       }
13     }
14   }
15 }
```

**LISTING 9: Interrupt to update the elapsed time**

# Atomic operations

- What would happen calculateSecondsSinceMidnight (void) function is called somewhere else from a critical section that has disabled interrupts?
- In this case, the function will incorrectly enable interrupts on return.
- Listing 11 provides a better solution to this problem

```
1  long calculateSecondsSinceMidnight (void){
2    long result =0;
3    bool isInterruptAlreadyDisabled = false;
4    interrupt_disable();
5    isInterruptAlreadyDisabled = true;
6    result = (long) (hours * 3600 + minutes * 60 + seconds);
7    if(isInterruptAlreadyDisabled){
8      interrupt_enable();
9    }
10 return result;
11 }
```

**LISTING 10: A bug-free function to compute the number of seconds since midnight**

# How about this solution?

```
1   static long int seconds;
2   int SECONDS_IN_DAY =60*60*24;
3   void interrupt updateTime(void){
4     // Code omitted for brevity
5     ++seconds;
6     if(seconds == SECONDS_IN_DAY){
7     seconds = 0L;
8     }
9     // Code omitted for brevity
10  }
11  long calculateSecondsSinceMidnight (void){
12    return seconds;
13  }
```

**LISTING 11: A risky function to compute the number of seconds since midnight**

# How about this solution?

- This is a risky, irresponsible and dangerous solution
- if the CPU has registers (e.g., 32-bit registers) that are large enough to hold the up to the total number of seconds in a day (i.e., 86400), then the code would be atomic. The generated assembly would be atomic as shown below

```
1    MOVE R1, seconds
2    RETURN
```

- Otherwise, the generate assembly would be non-atomic

```
1    // Get the first byte or word
2    MOVE R1, seconds
3    // Get the second byte or workd
4    MOVE R2, (seconds +1)
5    ...
6    // Code omitted for brevity
7    ...
8    RETURN
```

# The volatile Keyword

- Compilers assume that the value in a variable stays in memory unless the program changes it, and they use this assumption to optimize the code.
- In embedded systems, this can lead to serious problem[2]
- For example, in Listing 12, the compiler might read *seconds* in one or more registers and instead of updating the value before saving it to *result*, it will read the value from the register every time instead of from memory.
- Some compilers might also remove the while-loop during optimization causing the same bug we were trying to avoid
- This is prevented by declaring *seconds*as volatile. This warns the C compiler that the variable might change due to interrupts or other routines and not to optimize code pertaining to it.

```
1 static volatile long int seconds;
```

[2]This problem may not arise during firmware development when usually the compile's optimization is with turned off.

# The volatile Keyword

```
1  static long int seconds;
2  void interrupt updateTime(void){
3    ...
4    seconds++;
5    if(seconds == 60L*60L*24L){
6      seconds = 0L;
7      }
8    ...
9  }
10 long calculateSecondsSinceMidnight (void){
11   long result =0;
12   //When we read the same value twice, it must be good
13   while(result != seconds)
14     result = seconds;
15   return result;
16 }
```

**LISTING 12: Function to compute the number of seconds since midnightAssembly**

# Interrupt Latency

- Interrupt latency is the amount of time taken to respond to an interrupt.
- It depends on several factors:
  - Longest period during which the interrupt is disabled
  - Time taken to execute ISRs of higher priority interrupts
  - Time taken for the microprocessor to stop the current execution, do the necessary ‘bookkeeping’ and start executing the ISR
  - Time taken for the ISR to save context and start executing instructions that count as a “response”
  - The third factor is measured by knowing the instruction execution times from the processor manual, when instructions are not cached.
- Disabling interrupts increases interrupt latency so this period should be kept as short as possible

# Keep ISR short and fast

- ISRs affect the normal execution of program and can block handling of other interrupts. Thus, interrupt service routine (ISR) should be short and to the point so that the primary application can resume execution.
- The real point of an interrupt is to handle an urgent event that requires the system's attention. To keep the routine short, only do the minimum of what really needs to be done at that moment. For example, if communication data is triggering the interrupt, stuff the data into a buffer, set a flag and let the main program process the data. Don't try to process it in the interrupt!
- Don't call a function from within your interrupt (unless they are inline functions). The function call overhead will kill your timing.
- Any processor intensive activity such as processing a data buffer, performing a calculation, etc. should instead set a flag and let the main application do the processing.
- Avoid delay functions, loops or any time intensive logic such as for loops, division or modulus operations

# Understand ISR response time

- Not taking into consideration the interrupt response time can lead to serious bugs that are hard to find and troubleshoot.
- For example, a simple context switch into the interrupt can take between four clock cycles. For a microcontroller running at 8 MHz this can be a delay of 500 nanoseconds. This is just for one context switch into the interrupt! There is another one when leaving the interrupt!
- It is very important to understand how often your periodic and asynchronous interrupts are occurring so that you can ensure the rest of the program has an opportunity to run.
- It's also important to understand if there is an opportunity for interrupts to occur at the same time and if they do understand the latency between one finishing and the next running.

# The end