MICROCONTROLLERS
INTERRUPTS and ACCURATE TIMING I

OBJECTIVE
We aim at becoming familiar with the concept of interrupt, and, through a specific example, learn how to implement an interrupt process with the Arduino board. Along this description we will become aware of the “NEC IR communication” protocol.
As a specific practical example, an infrared remote control unit is used to (upon punching a particular key) send a time-coded ON and OFF (i.e. modulated) infrared signal, which is detected at a distance by a demodulator unit (the latter containing an infrared detector and capability to demodulate the signal into a binary electrical information) as shown in Fig. 5. The output binary signal from the demodulator is used as interrupt of the Arduino microcontroller (Fig. 6). A subroutine (provided herein) is added to the Arduino program so that the decoded signal (the interrupt signal) is sequentially read and the full signal displayed in a monitor display.

1. INTRODUCTION
The concept of priority
For a scalar processor, only one process can occur at a time. Under normal operations, processes are executed one by one; if there is no priority, each process will be executed according to its order number in a queue. Even if a process were become urgent, it has to be executed in the queue when its turn comes.
In computer science or electrical engineering, PRIORITY means that some process needs to be executed immediately. The process that has priority status can stop everything else, and it cannot be stopped by anything else until it is over. For a simple processor, a priority scheme is implemented by INTERRUPT.

The concept of interrupt
Suppose the processor is running processes one by one now and, suddenly, one task with higher priority comes. The processor will stop the current task immediately, and run the processes of the new task. After the task with high priority is over, the processor will resume the former processes. This whole procedure is called interrupt.

More details about interrupt
Interrupts can be classified into internal interrupt, external interrupt, hardware interrupt, software interrupt, maskable interrupt and non-maskable interrupt. Here we focus on the external hardware interrupt. (Herein, instead of the term “process” we will use the word “instructions”.)
First, an interrupt signal is triggered by an external event (a change in the state of a dedicated pin), or by an internal event (a timer or a software signal).

Once triggered, the interrupt pauses the current activity and forces the program to execute a different function. This function is called an interrupt handler or an interrupt service routine (ISR), which is a special subroutine that has to be included in our microcontroller code program. (In the example implemented in this lab, the ISR subroutine is called “timing”, as shown in Fig. 8).

Once the function is completed, the program returns to what it was doing before the interrupt was triggered. See Fig. 1 below.

![Diagram](image)

**Fig. 1** Illustration of the concept of interrupt.

As shown in the table below, UNO boards support 2 interrupts, Leonardo boards support 4 interrupts. In order to achieve successful interrupt, one needs to

1. hook up the interrupt signal to the right pin (hardware), and
2. cite the right interrupt number within the code (software).

<table>
<thead>
<tr>
<th>Board</th>
<th>int.0</th>
<th>int.1</th>
<th>int.2</th>
<th>int.3</th>
<th>int.4</th>
<th>int.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uno, Ethernet</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mega2560</td>
<td>2</td>
<td>3</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>32u4 based (e.g. Leonardo, Micro)</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Due, Zero, MKR1000, 101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interrupt vector = ( int.0, int.1, int.2, int.3, int.4, int.5)
In this lab we will use only one interrupt (usually “int.0” if it were damaged use “int.1”).

According to the table above, when choosing interrupt 0:
the interrupt will arrive at pin 3 in the Leonardo board.
The interrupt will arrive at pin 2 in the UNO board.

**Accurate timing**

One major application of interrupt is accurate timing. For non-critical applications, timing is implemented via loops. The microcontroller will record WHEN the physical interrupt happens. (In this lab we will ignore how long the microcontroller does take to establish the time the interrupts occurs; i.e. we will assume it is instantaneous. However, since the processor is working at 16 MHz, we expect a few µs accuracy in this measurement).

2. **INFRARED (IR) REMOTE CONTROL, and THE NEC IR COMMUNICATION PROTOCOL**

In the NEC IR protocol, the IR signal consists of 2 parts. One part is the *head code*, the other part contains the *actual code*.

- The head code is 9 ms HIGH follows a 4.5 ms LOW.
- For the actual code, 0 is coded as 560 µs HIGH and 560 µs LOW; while 1 is coded as 560 µs HIGH and 1650 µs LOW.

Each transmission is 32 bits long.

Every time the IR signal is sent, the head signal is sent first and subsequently followed by the actual code.

![Fig. 2 The NEC IR communication protocol.](image)

But there so many influences in the environments. In order to keep the code integrity, we use the code to modulate a 38 KHz square wave.
Amplitude modulate 38 kHz signal. The protocol requires actual code of 32 bits.

Then the 38 KHz modulated signal is sent to an amplifier to drive the IR diode to emit the $3 \times 10^{14}$ Hz IR signals. From the remote side, when a button is pushed, the modulated $3 \times 10^{14}$ Hz IR signals are sent out from the IR diodes.

At the receiver side, an IR receiver diode gets the electromagnetic $3 \times 10^{14}$ Hz IR signals, and demodulates the IR signal into 38 KHz code modulated signals, i.e. replicates the ones sent to the amplifier of the remote. Then this 38 KHz modulated signals are furtherly demodulated into NEC codes. See the illustration show in figure 5.
**Figure 5.** Modulation and demodulation process. **Left:** Emission of IR light is modulated by the input code. **Right:** Photodiode receives the IR signal, which is then demodulated to obtain a replica of the digital input signal. (The remote control unit and the demodulator unit have to be compatible.)

The signal $V_{out}$ from the receiver will be used to implement the INTERRUPT process in our microcontroller. Each HIGH to LOW transition in the $V_{out}$ signal (arrows pointing down in Fig. 5) constitutes an interrupt. The objective in this lab is to familiarize with how to incorporate those interrupt signals into any microcontroller program, and, in particular (in this lab session) be able to read $V_{out}$ and display it in a computer monitor; i.e., we will see in the monitor the signal sent by the IR remote control.

**Figure 6.** Schematic hardware arrangement to demodulate the on and off (modulated) electromagnetic IR signal sent by the remote control unit. On the far left, each key in the remote control unit has an associated coded signal. Upon pressing a key, an ON and OFF encoded IR wave is sent out. A demodulator unit receives the signal and demodulates it into an electrically binary signal $V_{out}$. On the far right, a microcontroller is able to decode the electrical signal $V_{out}$ and display it to the user in the monitor.

In this lab, we are going to use the falling edges of the NEC codes to trigger the interrupt in the microcontroller program.
When the Arduino detects a HIGH to LOW (falling edge) transition at the designated pin (pin-3 for the case of the Leonardo board), the control is passed to the interrupt service routine (ISR). (In our example the ISR subroutine is called “timing”.)

- The ISR accomplishes an accurate timing of the arrival time of the interrupt signal.
- Then, following the NEC protocol, we will be able to decode the actual code sent by the remote control unit.

2. Experimental procedure.

2.1 Connect the circuit like the one shown below.

Find out the pin # your microcontroller board assigns to detect the interrupt signal. (The program in Fig. 8 has selected to use the 0-th component of the interrupt vector. Hence, according to the table on page- 2, find out which pin-# has been assigned to be the 0-th component). Connect the signal $V_{out}$ from the decoder unit to that pin.

- when choosing interrupt 0:
  - the interrupt will arrive at pin 3 in the Leonardo board.
  - The interrupt will arrive at pin 2 in the UNO board.
Run the program shown in Fig. 8.
Pay attention to the actual microcontroller board you are using (see the table on page 2) because they have different interrupt vector. Find the correct interrupt number and the related pin of the board to input the interrupt signals.

When the program is compiled, a “guardian” inside the board is assigned to watch whether the interrupt pin (pin 3 in the case of the Leonardo board) makes a HIGH to LOW transition. (This assigning task is transparent neither in the hardware-connection layer, nor in the programing-code layer. It is hidden for us in one of the many layers that constitute the functioning of the microcontroller). When the board detects such a transition, it records the time (in the program we use the function micros() to retrieve that information ) and the control of the program is “instantaneously” passed on to the ISR subroutine (the “timing” ISR subroutine in the program below).

Each HIGH to LOW transition in the signal $V_{out}$ (arrows pointing down in Fig. 5) constitutes an interrupt.

```c
unsigned long inputnm=0x00000000; // inputnm is a 32 bit array where the decoded binary
   // information will be saved, which be be displayed
   // later on in the monitor.
volatile int tintv=0, lastus=0;      // volatile tells the compiler that the value of the
volatile char lead=0, bt=0;          // variable may change at any time.
   // char is an 8 bit long variable.

// This is the ISR subroutine ( It goes into effect once the Arduino board detects a HIGH to LOW 
// transition at the interrupt pin. )
void timing()                          // The purpose is to record the time interval between
   // contiguous interrupts (since it has to be verified whether 
   // or not the upcoming signal fulfills the NEC protocol.)
{
    tintv=micros() - lastus;          // micros() is a system library to record time;
    lastus=micros();                 // here it records the time the ISR subroutine is
                                       // called ( because an interrupt has been activated)

    // The head code is 13,500 $\mu$s long (nominally). Suppose
    // we are allowing $\pm$ 1500 $\mu$s precision, then,
    // we choose a range from 12,000 to 15,000; if the
    // time interval between two interrupts fall within this
    // range, we will accept that it is fulfilling the NEC protocol.
    // as head-code information.

    if ( tintv>12000 && tintv<15000 )       // It verifies whether a head signal has arrived
    {
        bt=0, lead=1;
        // lead=1 if we got the correct head signal;
        return;
    }
}
```
if (lead==1)
{ if ( tintv>800 && tintv<1500) bitClear(inputnm,bt), bt++;
    // It checks if a 0 code has arrived.
    // bitClear is a system library. It writes a 0 to the component
    // (inputnm, bt) of the inputnm array.
    // A 0 code is 560+560 = 1120 $\mu$s long. Suppose we
    // allow $\pm$ 300 $\mu$s precision. If two interrupts were far
    // apart by any value between 800 $\mu$s and 1500 $\mu$s
    // it will then be considered that a 0 code information
    // has arrived.

    if ( tintv>1900 && tintv<2400) bitSet( inputnm, bt ), bt++;
    // It checks if a 1 code has arrived.
    // bitSet is a system library. It writes a 1 to the component
    // (inputnm, bt) of the inputnm array.
    // A 1 code is 560+1650 = 2210 $\mu$s long. Suppose we
    // allow $\pm$ 300 $\mu$s precision. If two interrupts are
    // far apart by any value between 1900 $\mu$s and 2400 $\mu$s
    // it will be considered that a 1 code information
    // has arrived.

    if ( bt==32) bt=0, lead=0;
    // It reads the pulse counters bt, to end the transmission.
    // inputnm stores only 32 bits.
}

void setup()
{
    pinMode(3, INPUT_PULLUP);
    // initializes pin 3 as an input with an internal
    // pull-up resistor enabled.
    // Thus, we are forcing pin-3 to be a negative logic input;
    // i.e. effective when pin-3 acquires a LOW level.\(^2\)
    // (Pin-3 will be forced to be low by an external signal,
    // which in our project is the interrupt signal).

    Serial.begin(9600);
    // Series BAUD rate of 9600 bytes/sec between the
    // Arduino and the computer

    attachInterrupt( 0, timing, FALLING);
    // This sentence allows the program to be interrupted.
    // AttachInterrupt is a system library.\(^3\) This interfaces
    // (or bridges) the hardware layer with the signal layer.

    // Here we are using int0 (the 0-th component
    // of the interrupt vector. (See table on page 2).

    // timing is the subroutine-ISR to call when the
    // interrupt occurs;
// FALLING defines that the interrupt should be triggered
// when the interrupt pin goes from HIGH to LOW

// Notice the program does not call for the pin# that
// is used for the interrupt; instead it calls for
// the i-th component of the interrupt vector. The
// table on page-2 indicates which pin of the board
// is associated with the i-th component of the
// interrupt vector.

void loop()
{
  Serial.println(inputnm, HEX), delay(200);
}

Figure 8.

TASK: Write down the HEX code of each button of the remote.

OPTIONAL EXPERIMENTS
2.2 IR remote control demo. Remote control of the 7 segment display

Connect LED display bus abcdefg to D4---D10 of the Arduino board
  LED display D1 to Arduino D12
  LED display D2 to Arduino D13
  LED display D3 to Arduino A0
  LED display D4 to Arduino A1

When choosing interrupt 0:
  The interrupt will arrive at pin 3 in the Leonardo board.
  The interrupt will arrive at pin 2 in the UNO board.
unsigned char inputNum=0;
volatile int tintval=0, lastus=0;
volatile char leadok=0, bt=0;
volatile int push=0;
int pushCheck=0;
int s=5000;
char inputNumLast=0;
#define D1 12
#define D2 13
#define D3 18  // If using UNO , use A0.  #define D3 14
#define D4 19  // If using UNO , use A1.  #define D4 15
#define BusL 4

void timing()
{		tintval=micros()-lastus;
		lastus=micros();
		if ( tintval>12000 && tintval<15000)
		{
			bt=0, leadok=1;
			push++;
return; // correct leading code.
}

if (leadok==1)
{
    if (tintval>800 && tintval<1500)
    {
        if(bt>=16) bitClear(inputNum,bt-16);
        bt++; }

    if (tintval>1900 && tintval<2400)
    {
        if(bt>=16) bitSet(inputNum,bt-16);
        bt++;}

    if (bt==24) bt=0, leadok=0;
}

void writePort8(char dat,int j) // write a byte to Dj~~~Dj+7
{
    for(int i=0;i<=7;i++)
        digitalWrite(j+i,bitRead(dat,i)==1? HIGH:LOW);
}

const char disp[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,
0x07,0x7f,0x6f,0x77,0x7c,0x39,0x5e,0x79,0x71,0x80};

void write4digits(int dat)
{
    int i4=dat%10;
    int i3=dat%100/10;
    int i2=dat%1000/100;
    int i1=dat/1000;

    for (int j=0;j<5;j++)
        {writePort8(disp[i1],BusL),digitalWrite(D1,LOW),delay(1),digitalWrite(D1,HIGH),delay(5),
        writePort8(disp[i2],BusL),digitalWrite(D2,LOW),delay(1),digitalWrite(D2,HIGH),delay(5),
        writePort8(disp[i3],BusL),digitalWrite(D3,LOW),delay(1),digitalWrite(D3,HIGH),delay(5),
        writePort8(disp[i4],BusL),digitalWrite(D4,LOW),delay(1),digitalWrite(D4,HIGH),delay(5); // delay(8);
        }
void setup() {
  pinMode(3,INPUT_PULLUP);
  attachInterrupt(0,timing,FALLING);
  for (int i=4;i<=19;i++)
    pinMode(i,OUTPUT);
}

void loop() {
  switch (inputNum) {
    case 0x44:
      s--;
      break;
    case 0x40:
      s++;
      break;
    case 0x07:
      if(pushCheck!=push)s--;
      break;
    case 0x15:
      if(pushCheck!=push)s++;
      break;
    case 0x43:
      // if(pushCheck!=push)inputNum=inputNumLast;
      break;
    default: inputNum=inputNumLast;
  }
  inputNumLast=inputNum;
2.3 Using the IR remote to only send control signals

In this case we do not have to implement the complete NEC protocol. Try the program shown in Fig. 11. It is a simplified version of the program in Fig. 8. [In layer 1,2,3, they do the same things. In the subroutine, the below program only stores 8 bits of the code. The above program stores the whole 32 bits. That is the major difference. Both programs read 0s and 1s. The program in Fig. 9 handles it by a different way; It is suitable for a larger project with IR control. The robustness of the system will be taken care of by the caller of this block not by this block itself.]

```c
volatile int tintval=0,lastus=0;
volatile char bt=0, inputNum=0;
void timing() // Simplified ISR subroutine
{
    tintval=micros() – lastus, lastus=micros();
    if(tintval>12000 && tintval<15000)
        { bt=-1; return; }
    else  bt++;
    if( bt>=16 ) { (tintval>800 && tintval<1500)?  bitClear(inputNum,bt-16):bitSet(inputNum,bt-16); }
}
void setup()
{
    pinMode(3, INPUT_PULLUP); // To setup pin-3 as the interrupt signal pin . By keeping it
    // in HIGH, we are setting it to be effective when forced to
    // go LOW. That is, it is ready for negative logic.
    Serial.begin(9600);
    attachInterrupt(0,timing,FALLING); }
void loop() {Serial.println(inputNum,HEX),delay(500);}
```

Figure 11.
2.4 IF you want to use IR control in the future (for example to control the stepper motor with the IR remote control) ADD the program suggested below to your new project program. Pay attention to the comments.

/* The detailed interrupt scheme and IR transmission is out of
 * the scope of this course. Here, you only need to get the
 * concept of interrupt. If you want to use wireless communication,
 * you only need to download the on-line library for the specific
 * communication protocol.
 */

// if you want to use the IR control in your project, you only need to
// copy and paste the section below, to your program.

//==========COPY From HERE!==========
volatile int tintval=0, lastus=0;
volatile char bt=0, inputNum=0;
void timing(){
  tintval=micros()-lastus, lastus=micros();
  if(tintval>12000 && tintval<15000)
  { bt=-1; return;}
  else bt++;
  if( bt>=16 ) {{tintval>800 && tintval<1500)?
    bitClear(inputNum, bt-16):bitSet(inputNum, bt-16);}
}
//==========COPY End at HERE======

//======Copy the below 2 lines into your setup() subroutine!
void setup()
{
  pinMode(3,INPUT_PULLUP);
  attachInterrupt(0,timing,FALLING);
  //================================
When a circuit requires logic 1 to operate, engineers may refer to this condition as *positive logic*. Thus, the more positive voltage causes the action to take place. On the other hand, if a circuit requires a logic 0 to cause action, this type circuit is referred to as *negative logic* [Ref. http://www.sealevel.com/support/article/AA-00509/0/What-is-the-difference-between-positive-and-negative-logic-in-digital-I-O-circuits.html]


```c
attachInterrupt( interrupt vector component, ISR, mode);
```

**interrupt vector component:**
It varies from board to board (see table on page 2).

**ISR:** It is the subroutine-ISR to call when the interrupt occurs; this function must take no parameters and return nothing.

**Mode:** It defines when the interrupt should be triggered.
Four constants are predefined as valid values:
- **LOW** to trigger the interrupt whenever the pin is low,
- **CHANGE** to trigger the interrupt whenever the pin changes value
- **RISING** to trigger when the pin goes from low to high,
- **FALLING** for when the pin goes from high to low.