EduCake: Interrupt and Event

1. Introduction to Interrupt and Event

In previous chapters, we talked about how the 86Duino EduCake interfaces to external peripherals, such as digital I/O, analog I/O, sensor and motor. This chapter talks about interrupt and event, which are important to develop responsive and efficient application. In the previous chapters, we had multiple exercises that use combination of digitalRead/Write(), analogRead/Write() and delay() function to change LED brightness, read analog input at predetermined time interval and etc. These are polling method. While polling method works, it's not the best and efficient method, where the code need to run continuously in loop with the delay() function to control timing in order to read and capture input from one or more input sources. While executing the program loop, it consumes processing and memory resources which add unnecessary loading to the controller.

Instead of polling, an efficient application should be written to respond to external events and executes relevant codes each time these external events happen. These events trigger one of the microcontroller's interrupts to alert the system there is pending task to handle, where an Interrupt Service routine (ISR) is used to handle the event. Interrupt and event are part of the common design for the PC we are using and many microprocessors in the current market, including the Vortex86EX which the 86Duino EduCake is built on. Even legacy serial port communication is heavily relying on interrupt and event to communication efficiently.

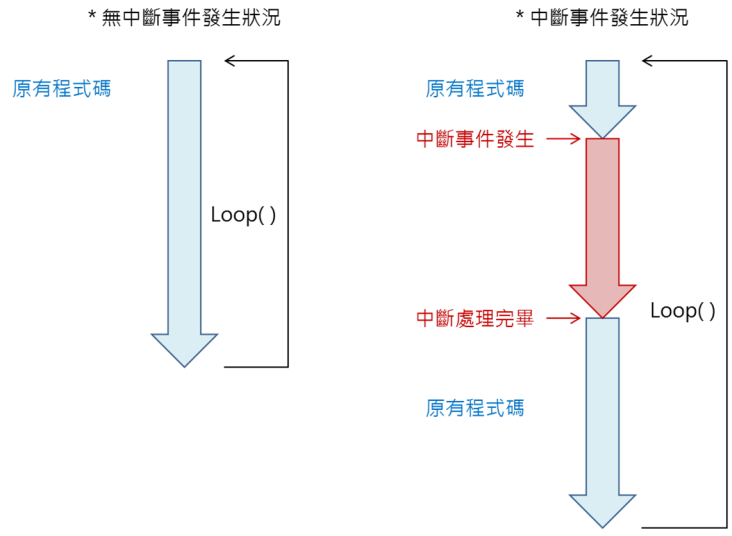


Figure-1: Typical interrupt processing operation for a processor

Instead of polling, an efficient application should be written to respond to external events and executes relevant codes each time these external events happen. These events trigger one of the microcontroller's interrupts to alert the system there is pending task to handle, where an Interrupt Service routine (ISR) is used to handle the event. Interrupt and event are part of the common design for the PC we are using and many microprocessors in the current market, including the Vortex86EX which the 86Duino EduCake is built on. Even legacy serial port communication is heavily relying on interrupt and event to communication efficiently.

As shown in figure-1, while not servicing any interrupt tasks, the processor goes about to perform other designated tasks. When an interrupt occurs, the processor will temporary halt existing tasks and jump into an interrupt handler and process tasks associate with the interrupt. After the interrupt tasks is handled, the processor will continue to process the task that was halted just before the interrupt occurred. Since the interrupt handling process affects the code that has to halted in order to process the interrupt event, the code that is part of the interrupt handler must be optimized and take minimal processing time to complete.

There are two different type of interrupts, "internal interrupt" and "external interrupt". Internal interrupts are part of the processor's built-in function, such as the built-in timer used to trigger functions that need to perform at a set time interval, such as reading sensor data at 30 seconds interval, which is different from the delay () function where the processor is not able to perform other task while waiting for the delay () function. By using timer interrupt to trigger the function to read sensor data at 30 second interval, the processor is able to perform other tasks in between the 30 seconds waiting period when the interrupt is not occupying processor resources. Using the delay () method in polling mode, the processor is occupied by this process and not able to perform other task. In addition, the timer triggered events generate a far more accurate and consistent timing period than the delay () function. External interrupt, as the name implied, is triggered by sources external to the processor, such as when one of the GPIO pin's voltage is changed from low to high or vice versa. With proper circuitry and implementation, external device can trigger interrupt and generating event for the processor to react to and create an efficient application development environment. External interrupt is covered in more detail later in the chapter.

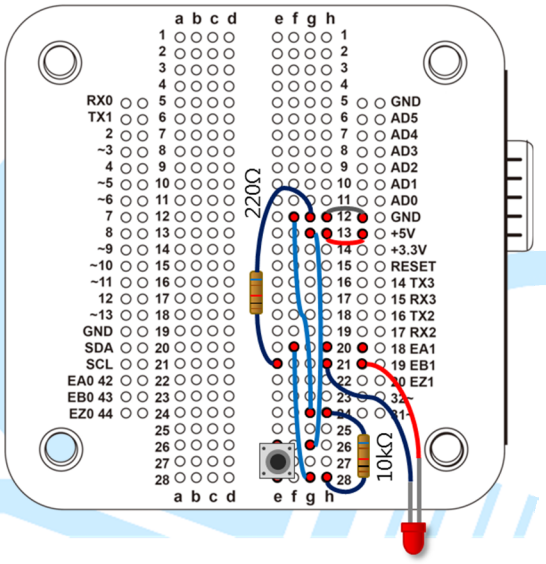
In addition to using interrupt, the 86Duino EduCake has other feature to detect changes from external devices using the pulseIn () function. This function can be used to detect change in voltage for a signal pin and the length of time the signal pin remains in

HIGH or LOW state. The pulseIn () function is used in the sample exercise later in this chapter, and compare the result with interrupt.

In the following section, we will work through different exercises to demonstrate how to use the interrupt and pulseIn () function.

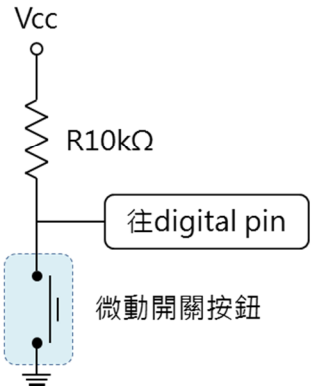
2. First exercise – attachInterrupt() and deattachInterrupt()

In this first exercise, we will work through an exercise to show how to use the 86Duino EduCake's interrupt, using the circuitry below:

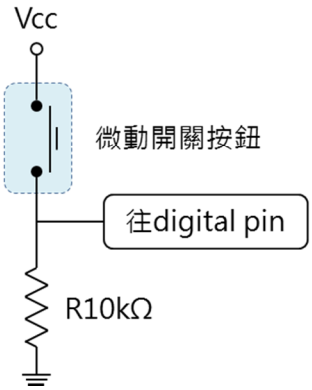


There are two different methods to implement the button circuitry, as shown in the following figures. These circuits are designed to be in voltage LOW condition, when the button is not pressed and will generate a HIGH voltage condition when the button is pressed.

按鈕一般未按下時，控制板偵測到HIGH
(Normal HIGH)



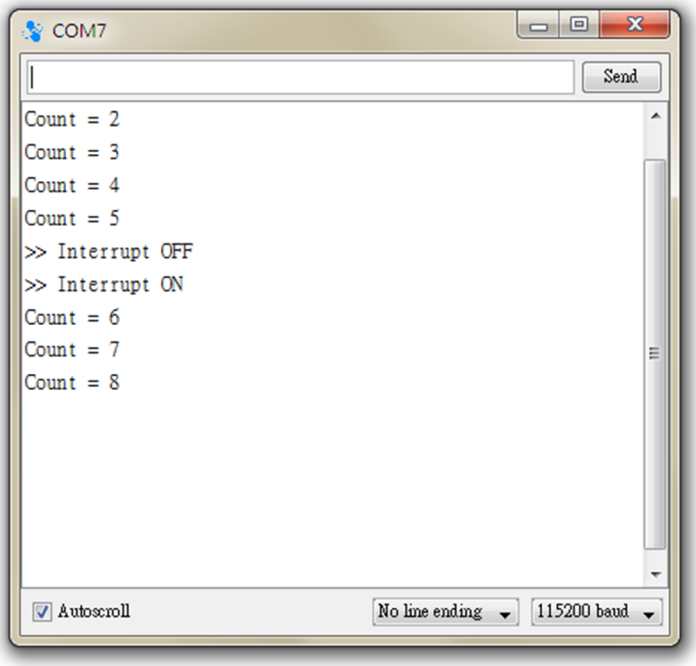
按鈕一般未按下時，控制板偵測到LOW
(Normal LOW)



Launch the 86Duino IDE and enter the following codes:

```
nt BTN_pin = 3;// Normal LOW, pin 18 = interrupt 3
int LED_pin = 19;
volatile int state = LOW;
int count = 0;

void setup()
{
  Serial.begin(115200);// Configure Serial port
  pinMode(LED_pin, OUTPUT);// Configure signal pin
  digitalWrite(LED_pin, LOW);// Initialize LED to OFF

  // attach interrupt to signal pin
  attachInterrupt(BTN_pin, InterruptHandler, RISING);
}

void loop()
{
  if(Serial.available()){// check Serial Port for data
    char data = Serial.read();
    if(data == 'A'){// when an "A" is detected

      // attach interrupt for the signal pin
      attachInterrupt(BTN_pin, InterruptHandler, RISING);
      Serial.println(">> Interrupt ON");
    }
    else if(data == 'B'){// when a "B" is detected

      // detach interrupt from signal pin
      detachInterrupt(BTN_pin);
      Serial.println(">> Interrupt OFF");
    }
  }

  digitalWrite(LED_pin, state);// turn LED ON
}

void InterruptHandler()

I{
state = !state;// Change LED state
  count++;// increment counter by 1
  Serial.print("Count = ");
  Serial.println(count);
}
```

For the above exercise, each time the button is pressed, it trigger the interrupt and associated processing function to change the LED state and increment the counter by 1. The Serial Monitor is used to display counter activities. When a character "B" is detected from the serial port, interrupt is disabled. When a character "A" is detected from the serial port, interrupt is enabled, as shown in the following figure.



The program begins with variables declaration, define signal pin used in the circuit, initialize LED status and counter value. Then, the attachInterrupt() function is called within the setup() to set the interrupt number, corresponding interrupt handler, interrupt detection mode and other parameters.

You may be wondering why BTN_pin is set to 3 in the code (BTN_pin = 3) while the actual button is connected to digital I/O pin #18. The 86Duino platform has multiple group of GPIO that can be associated with built-in interrupt, as shown in the following table (http://www.86duino.com/?p=1756):

中斷編號	int.0	int.1	int.2	int.3	int.4	int.5
EduCake	42	43	44	18	19	20

In the attachInterrupt(pin, ISR, MODE) and detachInterrupt(pin) functions, the pin parameter is referencing the interrupt number associated with the pin and not the physical pin number. Interrupt detection mode can be set to detect the following:

- Change in voltage from HIGH to LOW and from LOW to HIGH
- RISING – Voltage from LOW to High
- FALLING – Voltage from HIGHT to LOW

You can select one of the above interrupt detection method based on the type of circuitry is used. In this exercise, the pin associated with the button is in voltage LOW condition when the button is not pressed. When the button is pressed, the associated circuit causes the voltage on the pin to go HIGH. The code for this exercise uses the RISING detection mode to detect when the button is pressed.

Every time the interrupt handler is triggered, it change the LED status to the opposite, increment the COUNT by one and uses the Serial.print() function to output the result to the Serial Monitor.

Within the loop() function , there are codes that check inbound data from the Serial port. When a character “A” is detected, it triggers the attachInterrupt(pin, ISR, MODE) function, which will continuously detect the designated interrupt. When a character “B” is detected, it triggers the detachInterrupt(pin) function to stop the routine that is assigned to detect the designated interrupt.

Within the loop() function, the digitalWrite() function control the LED brightness based on the State variable.

In addition to processing the application code from the IDE, the 86Duino platform is actually handling other interrupt events, such as receiving incoming data for the Serial port.

In the above exercise, the attachInterrupt(0 and detachInterrupt() functions are used to start and stop interrupt detection for a designated I/O pin. In addition to these functions, the 86Duino platform has other interrupt handling functions, interrupts() and noInterrupts(). The interrupts() function is used to enable background interrupt handling. The noInterrupts() function is used to disable background interrupt handling. You can use these two interrupt function in the above exercise and compare the result.

When executing the code in the above exercise, you may notice a single button press event can trigger multiple interrupt events, which is caused by a condition refer to as bounce. When the button is pressed, the mechanical contact may create several intermittent contact as part of the process and trigger multiple interrupt events. Debounce is a common application issue for microcontroller which can be addressed as part of the hardware and/or software design. This is a common issue which you should search for addition and more detailed information to better understand the associated problem and resolution.

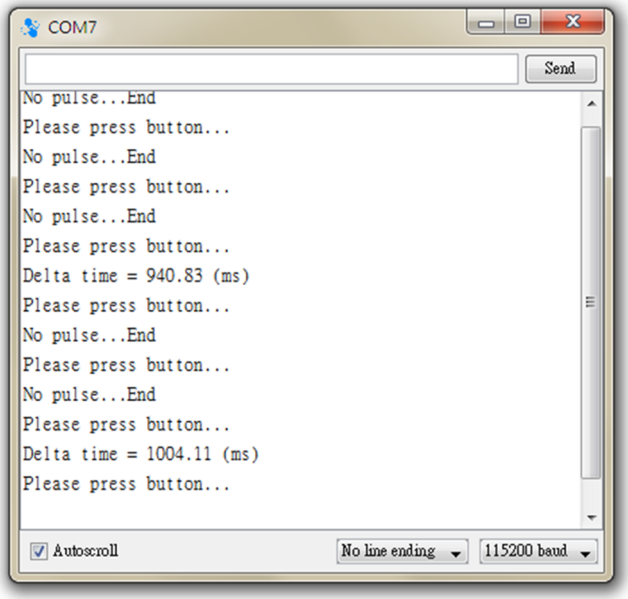
3. Second exercise – pulseIn()

The exercise in this section is similar to the previous one, using the puseIn() function

```
// pulseIn Timer
// I/O pin connected to button, normal low
//  pin 18 = interrupt 3
int btn_pin = 18;

// pulseIn timeout period, in micro-second
unsigned long max_duration = 2000000;

void setup() {
  Serial.begin(115200);// initialize Serial port
  pinMode(btn_pin, INPUT);// initialize I/O pin
}

void loop() {
  // Output message to serial monitor
  Serial.println("Please press button...");

  unsigned long duration = pulseIn(btn_pin, HIGH, max_duration);

  if(duration>0){
    Serial.print("Delta time = ");
    // output interval to serial monitor, in millisecond
    Serial.print(((float)duration)/1000);
    Serial.println(" (ms)");
  }
  else{
    Serial.println("No pulse...End");
  }

  delay(2000);
}
```

with the same circuitry. From the 86Duino IDE, enter the following code:

The sample application in this exercise detect the length of time the button is pressed. Prior to executing the application code, you need to launch Serial Monitor. When the "Please press button…" message is shown, you can press the button to see the result. The code detect whether the button has been pressed within 2 seconds. When button press event is not detected after 2 seconds, the message "No pulse…End" will be sent to the Serial monitor. After 2 seconds delay, the process to detect button press repeat again, as shown in the figure below:



In this exercise, the pulseIn() function does not use interrupt to detect button press event. Since the code is detecting the button event when the pulseIn() function is being executed, the code must run continuously inside the main program loop(). There are two calling conventions for the pulseIn() function:

- pulseIn(pin, value)
- pulseIn(pin, value, timeout)

The pulseIn() function operation is demonstrated in the figure below:

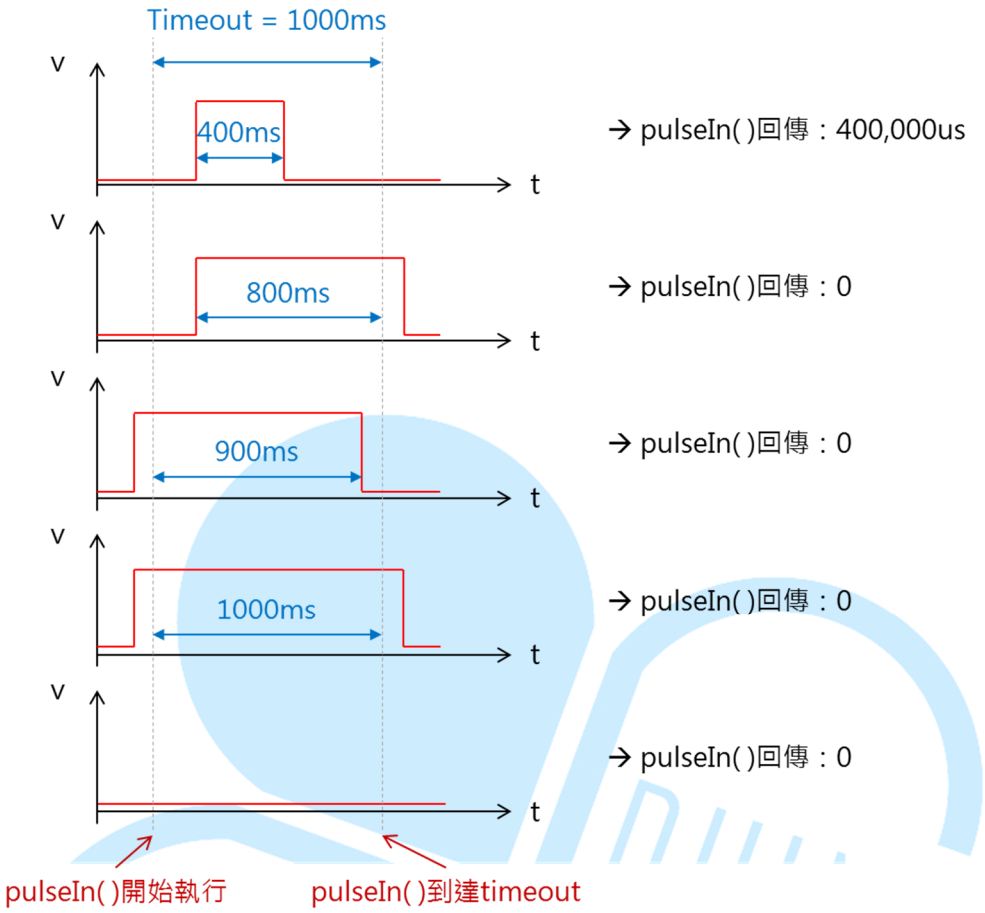


Figure: Return value from pulseIn() function based on 1000ms timeout value

In an example where the code is designed to detect voltage HIGH condition, when a full-wave is detected within the designated timeout period (both the rising and falling edge of the wave happen within the timeout period), the function return the detected value (length of time between the rising and falling edge). Otherwise, the function return 0 value. The pulseIn() function will detect the length of time from a rising-edge to falling-edge or vice versa from falling-edge to rising-edge, as long as both edge are within the timeout period. When change is not detected within the timeout period, the function return 0 value.

The above example start with declaring “unsigned long max_duration = 2000000” (the value is in micro second) as the timeout period.

The example for this exercise starts with declaring “unsigned long max_duration = 2000000” (the value is in micro second) as the timeout period. Using this timeout variable, the pulseIn() function will wait and attempt to detect change to the signal pin for 2 seconds and continue. When the pulseIn(pin value) function is used, the timeout period is set to 1 second. Within the setup() function, the pinMode(btn_pin, INPUT) function is called to initialize the signal pin linked to the button as input signal. The Serial.print() function inside the main program loop() output a message to the Serial Monitor to prompt the user to press the button. Then, the following line of code is used to detect button press event:

- unsigned long duration = pulseIn(btn_pin, HIGH, max_duration)

Since the signal pin for the button is in normal LOW condition, the pulseIn() function is set to detect voltage HIGH condition. If the signal pin for the button is wired as normal HIGH condition, you would set the pulseIn() function to detect voltage LOW condition. The return value from the pulseIn() function is in micro-second and need to be divided by 1000 to convert to millisecond. You can change the timeout value to a different range, such as to 5 seconds, to see different result.

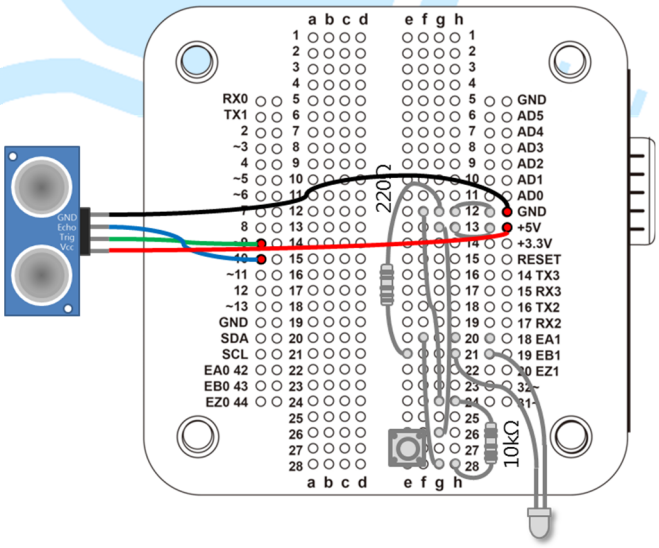
From this exercise, you can see the different between using the pulseIn() function and interrupt to detect change to the I/O pin, one of the methods is able to respond to external event immediately, the other needs to execute some code in order to detect the event.

Based on the code from this exercise, you can think about how to modify the code to detect reaction time from the time when the LED is turn on to the time when the button is pressed. How would you modify the code to accomplish this?

4. Third exercise

Although the pulseIn() function is not responsive, comparing with interrupt function, this function has its usefulness. In this exercise, we will use pulseIn() function to create interesting application. An ultrasonic sensor will be used for this exercise, using the HC-SR04 ultrasonic sensor which is widely available for purchase from many hardware vendor. Obviously, there are different variety of ultrasonic sensors available in the market where most of these sensors have similar function and behavior, with different sensing distance and accuracy. In general, most of the ultrasonic sensors in the market is built with an interface that has 3 to 4 wires, where 2 of these wires are 5V and GND, and the remaining wires are for sensor value.

The circuitry for this exercise is built on top of the existing circuit from the previous exercise. Using he existing circuit from the previous exercise, add additional components and wiring to the circuit, as shown in the figure below.



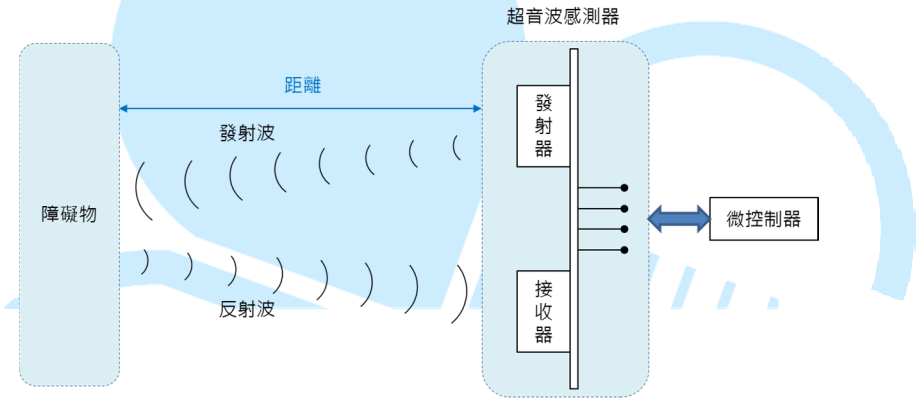
From the 86Duino IDE, enter the following code:

```
#define trigPin_1 9 // set pin number for trigPin
#define echoPin_1 10 // set pin number for echoPin
#define intervaltime 100 // set measurement interval in ms
#define LED_Pin 19// set output pin for LED
boolean LED_ON = false;// LED status
unsigned int LED_ON_count = 0;// LED on counter
```

```
unsigned int LED_ON_count_max = 1;// LED on max duration
unsigned int LED_OFF_count = 0;// LED off duration
int timeout = 12000; // set timeout period for pulseIn

// speed of sound in cm/micro second
float Sound_speed = 343.0f * 100 / 1000000;

void setup()
{
  Serial.begin(115200);
  pinMode(trigPin_1, OUTPUT);// set trigPin to output mode
  pinMode(echoPin_1, INPUT);// set echoPin to input mode
  pinMode(LED_Pin, OUTPUT);// set LED_pin to output mode
  digitalWrite(trigPin_1, LOW);// initialize trigpin to LOW
  delay(1);
}

void loop() {
  // Read sensor value from ultrasonic sensor
  // Convert sensor value to float
  float distance = Get_US();
  if(distance>0){
    Serial.print(", Dis= ");
    Serial.print(distance);

    if(LED_ON) {    // LED On
      LED_ON_count++;
      if(LED_ON_count >= LED_ON_count_max) {
        LED_ON_count = 0;
        LED_ON = false;
      }
      digitalWrite(LED_Pin,HIGH);
      Serial.print(", LED ON");
    }
    else {        // LED Off

LED_OFF_count++;
      if(LED_OFF_count >= int(distance/10)) {
        LED_OFF_count = 0;
        LED_ON = true;
      }
      digitalWrite(LED_Pin,LOW);
    }
    Serial.println();
  }
  else {
    Serial.println("Out of range !");
  }
  delay(intervaltime);
}
// function to read sensor value from ultrasonic sensor
float Get_US() {
  // Trigger
  digitalWrite(trigPin_1, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin_1, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin_1, LOW);
```

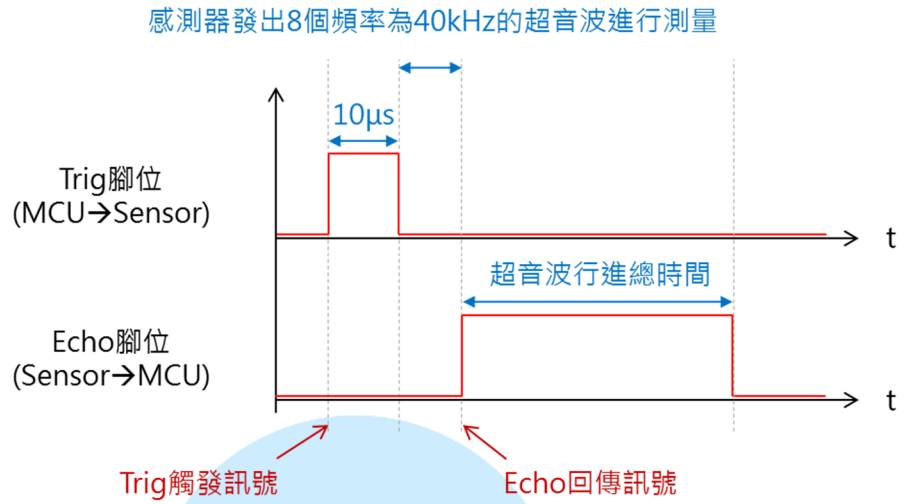
```
    // Read
    long duration = pulseIn(echoPin_1, HIGH, timeout);// timeout in us
    Serial.print("Dur= ");
    Serial.print(duration);

    // convert sensor value to distance
    float distance = (float)(duration) / 2 * Sound_speed;
    return distance;
}
```

The exercise in the section has similar function as the automobile's backup warning sensor. When detected obstacle is getting closer, the associated LED's flashing frequency increases. Before going through and reviewing this exercises' code, let's take a look at the how an ultrasonic sensor function, as shown in the following figures:



When the controller send trigger signal to the ultrasonic sensor, the sensor emit ultrasound and begin to calculate the time delay for the sensor to detect the reflected wave and send the detected value to the controller.

For the HC-SR04 sensor, there are 2 signal pins, Trig and Echo. The "Trig" pin is used to detect trigger signal from the controller to begin the sensor detection process. The "Echo" pin is used to send detected sensor value to the controller after the detection process is completed (there are other ultrasonic sensors in the market built with both Trig and Echo functions into the same pin). Both the "Trig" and "Echo" pins are in normal LOW condition. When the controller send triggering signal, it generates a HIGH voltage condition via the "Trig" pin for 10μs and then go back to the voltage LOW condition, to signal the sensor to begin the detection process. When the sensor detected the reflecting wave, the "Echo" pin send a voltage HIGH signal, where the time period for this HIGH signal is equivalent to "the time it takes for the transmitted ultrasound to be reflected back and detected by the sensor", in μs. This is where the pulseIn() function is useful. The ultrasonic sensor's detection process is shown in the following figure:



The code for this exercise begin with declaring and initializing variables such as measurement interval, timeout value for pulseIn() function, LED flashing parameter, speed of sound in cm/second and etc. In the setup() routine, signal pins connected to the sensor are initialized and set to proper operating mode. As you can see, there are very little code within the main program loop(). It's common for a sensor application to involve large number of sensors where the same code is replicated many times to support each of these sensors. To avoid repeating the same code over and over again, it's best to encapsulate the code into a function and call the function when needed to minimize the need to repeat the code in multiple location and simplify the effort needed to maintain the code. The ultrasound sensor handling code in this exercise is encapsulated in the Get_US() function. When the Get_US() function is called, it handles the necessary process needed to get sensors value and return the value to the calling function within the main program loop(), where the value is used to control LED and output to the Serial Monitor.

When the Get_US() function executes, to prevent unknowingly changing the EduCake's I/O pin associate with the Trig signal pin by other function, the "digitalWrite(trigPin_1, LOW)" function is called to set the pin to the designed settings. The "delayMicroseconds(2)" is called to provide sufficient time for the signal pin to reach a stable state. Then, the "digitalWrite(trigPin_1, HIGH)" function is called to trigger the sensor to emit sound wave, which will remain HIGH for 10μs and then go LOW to begin the detection process. After that, the "pulseIn(echoPin_1, HIGH, timeout)" function is called to capture the sensor value. Since the detected sensor value represent the time it take for the sound wave to travel from the sensor to the reflecting object and back, dividing the value by 2 yield the actual time for the wave to travel from the reflecting object back to the sensor.

The formula to calculate the distance is “distance = travel-time * speed”, where the speed of sound is calculated as “speed of sound = 331 + (0.6 * temp in Celsius)”. For this exercise, the calculation is based on a temperature value of 20 and use 343 m/s as the speed of sound (you can adjust the calculation based on realtime temperature value). Since the speed of sound is in m/s and the detection value from the ultrasound sensor is in µs. To convert the detected distance to cm, we need to convert speed of sound to cm/ µs using the following formula:

- Speed of sound (cm/ µs) = 343 (m/s) * 100 (cm/m) / 1000000

In the following line of code:

- Float distance = float(duration)/2 * Sound_Speed

To calculate the distance (floating point value) from the reflecting object, the duration (sound wave traveling time from sensor to reflecting object and back to sensor) is converted to floating point value, divide by 2 and multiple by the speed of sound to yield the distance.

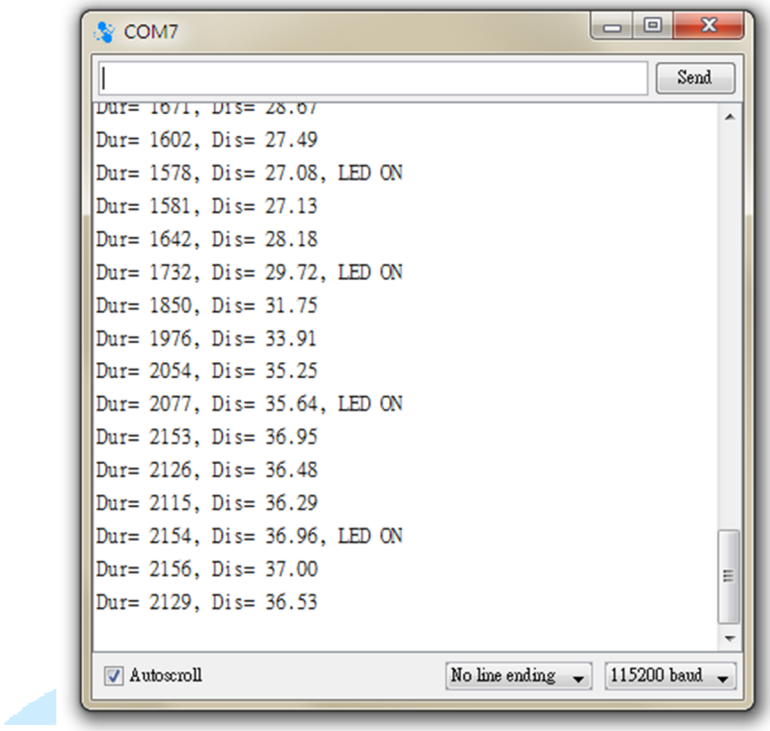
In the main program loop(), after acquiring detected value from the sensor and calculate the detected distance, the result is used to control the LED’s flashing rate and output to the Serial Monitor using the Serial.Print() function.

While the “delay(intervaltime)” function within the main program loop() is called with a constant (intervaltime), the varying time needed to process and capture sensor data create a fluctuating time to loop through the code. The flashing LED uses a counter approach to control the flashing rate.

Several global variables, LED_ON_count, LED_OFF_count and LED_ON related to LED function are declared in the beginning of the program. As the code in the main program loop() execute, the LED_ON_count value is incremented by one each time the code go through the loop, where the upper limit is set to 1 (which you can modify to yield different result). When the LED_ON_count reach the upper limit, the LED_ON variable is set to false, the LED_ON_count is set to zero and the LED is turned off.

When the LED_ON is false, as the LED_OFF_count is increment by 1, the upper limit is now set by the distance, using the value from “int(distance/10)”. Since the time delay for the main program loop() is set to 100 ms, when the detected distance is 200 cm, the LED will remain off for about 2 seconds. You can modify the code to use different value to see different result.

As the code is running, output from the Serial Monitor and LED flashing rate can be used to view the detected distance, as shown in the figure below:



From this exercise, you learn about how ultrasound sensor function. Think about a different approach, using interrupt or the digitalRead() function to get sensor data from the Echo signal pin. Instead of the flashing LED, think about replacing the flashing LED circuit with a Buzzer circuit to create a simple obstacle avoidance system with audible warning, such as backup obstacle warning system for automobile.