ALGONQUIN COLLEGE

CST8227 INTERFACING

Lecture 4

Polling

Polling

- "actively sampling the status of an external device"
- Polling a requires the regular monitoring of a status register.
- Computer wastes time checking status flags bits instead of doing something useful.
- Why is this inefficient?
  - The setting of flags can be sporadic – uncertain when they will be set

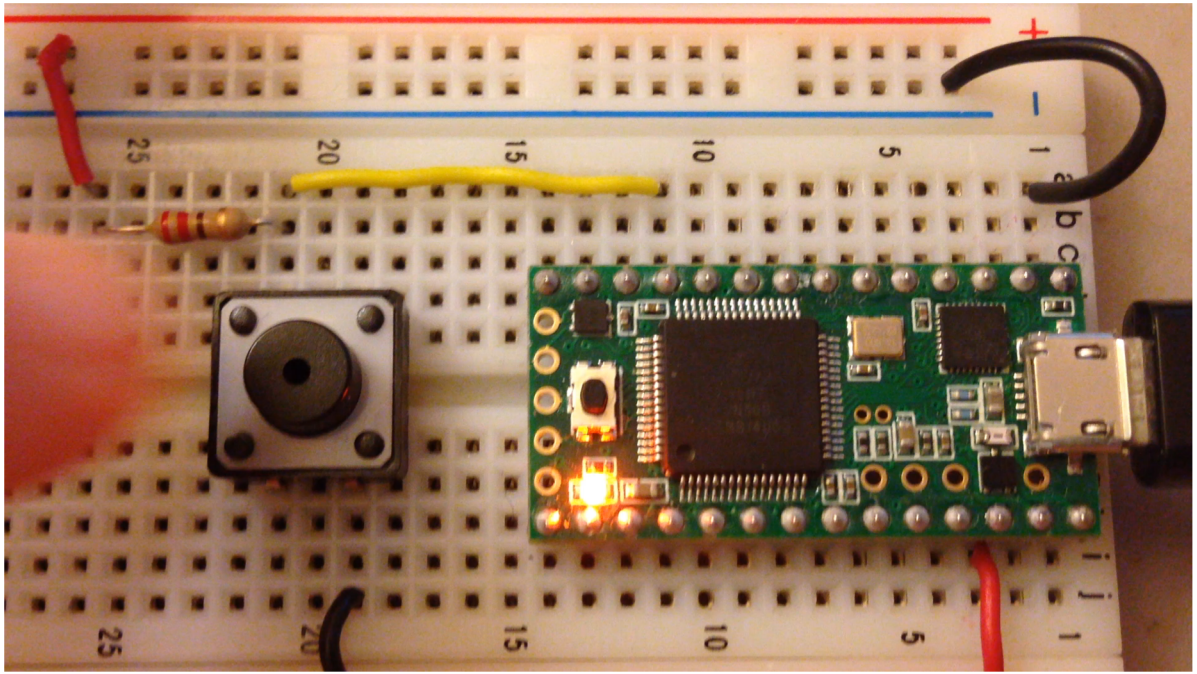
  Ex.
  - A processor can execute an instruction every microsecond ($1x10^{-6}$ s)
  - if a keyboard/processor wants to transfer 10 characters it will take ≈1 character for every 100 000 microseconds
  - But….checking a flag bit requires two instruction cycles

Lab 3 Learning Outcome: Polling

- One of the learning outcomes in Lab #3 is to “Use a polling loop to read a momentary contact switch”.
- Your main loop may have looked like:

```
void loop() {
        byte isButtonPressed = digitalRead(pushButtonPin);

        if (isButtonPressed == HIGH) //or LOW depending on logic
        {
                //invoke functions
        }
} //end loop
```

Lab 3 Demo 3: Polling (No Interrupt)

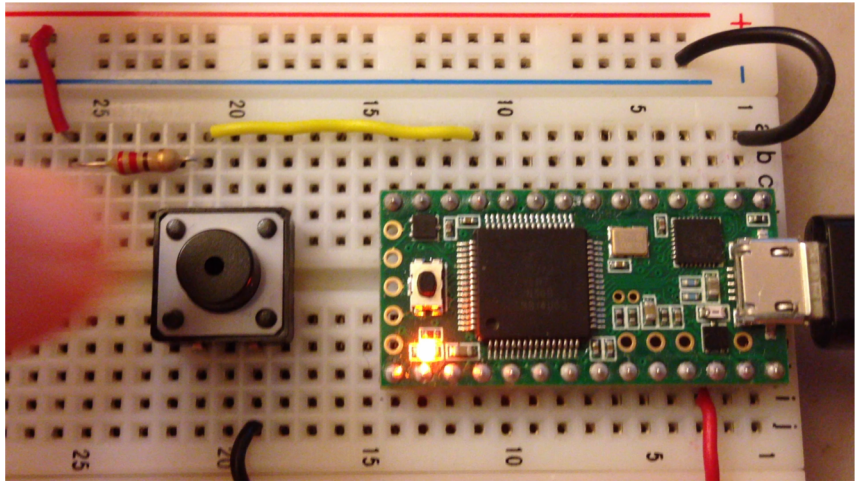


The sketch has a loop with a ~10 second delay.
No other processor function, such as reading the state of the button input, can execute while this delay is executing.

Tip: Research millis()

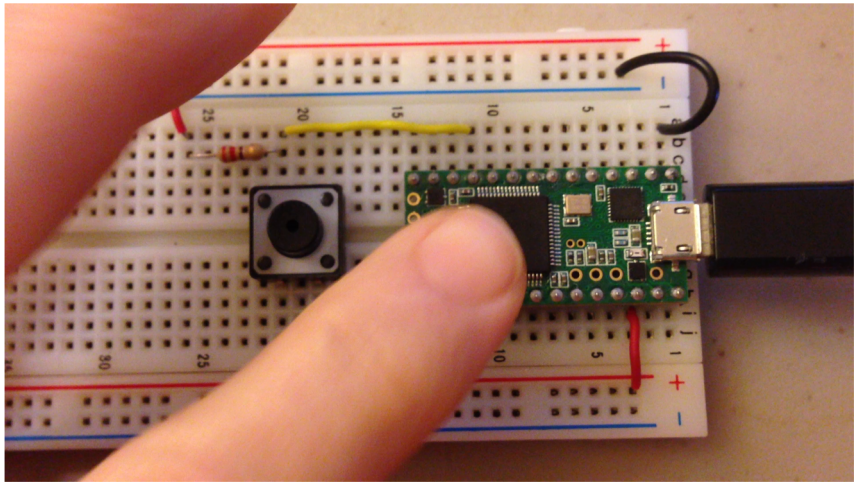
Blink: No Interrupt

```
int pbIn = 9; // digital input from pushbutton on pin 9
int ledOut = 13;       // The onboard LED pin
int state = LOW;       // The input state

void setup()
{
  // Set up the digital Pin 2 to an Input and Pin 4 to an Output
  pinMode(pbIn, INPUT);
  pinMode(ledOut, OUTPUT);
}

void loop()
{
  state = digitalRead(pbIn);      //Read the button
  digitalWrite(ledOut, state);    //write the LED state

  //Simulate a long running process or complex task
  for (int i = 0; i < 1000; i++)
  {
    // do nothing but waste some time
    delay(10);
  }
}
```



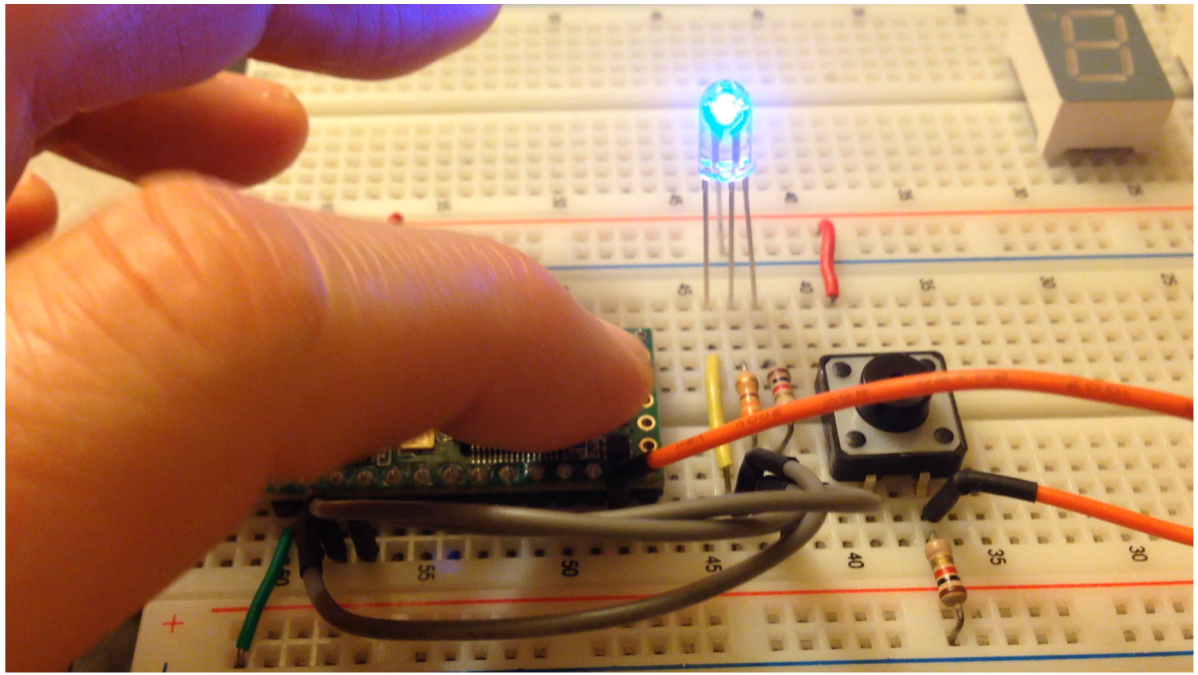
```
int pbIn = 9; // The input pushbutton pin
int ledOut = 13; //The output LED pin
volatile datatype myVariables;  //use modifier volatile for variables in both
                                //the ISR and in main routine
void setup()
{
  // Set up the digital pin 12 to an Interrupt and Pin 13 to an Output
  pinMode(ledOut, OUTPUT);
   pinMode(pbIn, INPUT);

  //Attach the interrupt to the input pin and monitor for ANY Change
  attachInterrupt(pbIn, myISR, Mode);
}

void loop()
{
  //test the ISR:  Simulate a long running process or complex task
  for (int i = 0; i < 100; i++)
  {
    // do nothing but waste some time
    delay(10);
  }
}

void myISR()
{
    //YOUR CODE HERE
}
```

Demo 4: Asynchronous Interrupt



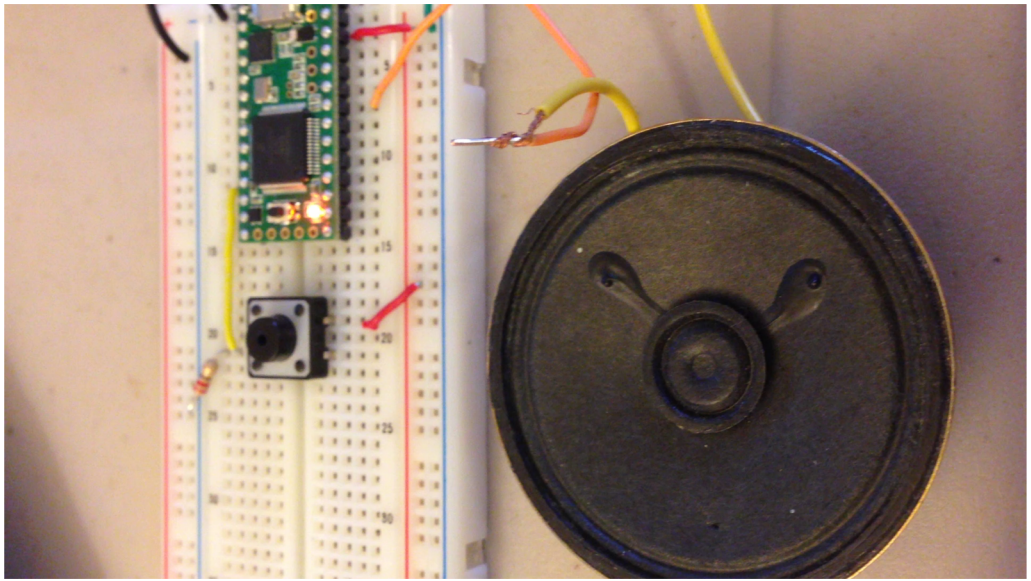
Mode is a placeholder for the keyword – search internet Arduino attachInterrupt
datatype myVariables are placeholders.

Lab 3 Demo 4: Interrupt



Interrupt with tone()

```
int A_440 = 440;
int speaker = 20;
int pushButton = 9;
int led = 13;
volatile int state = LOW;

void setup()
{
  pinMode(pushButton, INPUT);
  pinMode(speaker, OUTPUT);
   pinMode(led, OUTPUT);

attachInterrupt(pushButton, sound, CHANGE);//Specifies pin 0 to be
}

void loop() {

    digitalWrite(led, HIGH);

if(pushButton)
    digitalWrite(led, LOW);

}

void sound()
{
    state = !state;
    tone(20, A_440);
    delay(200);
    noTone(1000);
}
```



Interrupts

- **Interrupts** are signals that indicate the presence of an event that requires immediate attention by the microcontroller
- They are much like a function call:
  1. when invoked, execution of the current function is temporarily suspended
  2. the code representing the interrupt is executed
  3. when the interrupt code is finished, control is then returned back to the function that was originally executing.

- Interrupts are also a key to low-power designs, where the core is in deep-sleep mode and only wakes up if it needs to treat an interrupt.

Interrupt Service Routine (ISR)

- When a processor pin has been configured to be an interrupt, events (such as the pressing of a pushbutton) result in the setting of a flag.
- The processor reacts to the setting of the flag by temporarily deviating from what it is currently doing, services the interrupt event, in a process known as an Interrupt Service Routine (ISR).
- Process then resumes regular program control.
- The use of interrupts usually requires the setting of an “Input Enable” bit – this enables a programmer to decide if interrupts will be used or not.

Interrupt Controlled Systems

- Some embedded systems are predominantly interrupt controlled.
- This means that tasks performed by the system are triggered by different kinds of events.
- An interrupt could be generated for example by a timer in a predefined frequency, or by a serial port controller receiving a byte.
- These kinds of systems are used if event handlers need low latency and are short and simple.
- Usually these kinds of systems run a simple task in a main loop also, but this task is not very sensitive to unexpected delays.
- Sometimes the interrupt handler will add longer tasks to a queue structure. Later, after the interrupt handler has finished, these tasks are executed by the main loop. This method brings the system close to a multitasking kernel with discrete processes.

Interrupts and Teensy 3.2

- All pins on Teensy 3.2 are capable of running an ISR.
- Be very careful when referencing PJRC website and especially the arduino website: a lot of the code is associated with the ***Atmel*** chips and theTeensy 3.2 has an ARM Cortex chip.
- You will have to decide what is relevant to the ARM processor: experiment with the chip.
- The ARM processor on board the Teensy has an interrupt controller called the ***non-vectored interrupt controller*** (NVIC).
- The NVIC controls which interrupts are enabled and disabled.
- Teensy 3.2 supports up to 240 interrupts each with up to 256 levels of priority.

Asynchronous versus Synchronous Interrupts

Asynchronous

- A clock is ***not*** used.
- are generated by other hardware devices at arbitrary times with respect to the CPU clock signals.
- Ex. The pushbutton
- Ex. the arrival of a keystroke from a user sets off an interrupt.

Synchronous

- Clock is used.
- are produced by the CPU control unit while executing instructions
- The control unit issues an interrupt only after terminating the execution of an instruction.
- Intel refers to them as “exceptions”

Characteristics of Interrupts

i. External – Originate from external I/O devices, examples include the pushing of a button, circuit monitoring of power supply, etc… They are considered to be ***asynchronous***

ii. Internal – arise from illegal or erroneous use of instructions or data – ex. Register overflow, attempts to divide by zero, etc… Are ***synchronous***

iii. Software – initiated by executing an instruction – think of it as an imposing subroutine call ex. Request for data transfer

How to Register (implement) an Interrupt

Syntax: attachInterrupt(***digitalPinToInterrupt(pin)***, ***ISR***, ***mode***);

digitalPinToInterrupt(pin): pin number on Teensy

ISR: the ISR to call when the interrupt occurs; this function must take no parameters and return nothing. This function is referred to as an *interrupt service routine.*

mode: 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

Disabling or Enabling Interrupts

interrupts()

- Re-enables interrupts (after they've been disabled by noInterrupts()).
- Interrupts allow certain important tasks to happen in the background and are ***enabled by default***.
- Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.

Parameters

None

Returns

None

noInterrupts()

- Disables interrupts (you can re-enable them with interrupts()).
- Some functions will not work while interrupts are disabled, and incoming communication may be ignored.

Parameters

None.

Returns

None.

Interrupts and Teensy

- Functions available are:

External Interrupts

- attachInterrupt()
- detachInterrupt()

Interrupts (used to enable or disable interrupts)

- interrupts()
- noInterrupts()

Keyword “volatile”

- Used as a modifier when declaring variables,
  i.e. volatile int x;
- Indicates that the value of the variable may change in various places in a program, such as in concurrent threads of execution.
- “Specifically, it directs the compiler to load the variable from RAM and not from a storage register” - Arduino website reference: http://arduino.cc/en/Reference/Volatile
- Used with interrupts.

millis()

millis()

- Returns the number of milliseconds since the Arduino board began running the current program.
- This number will overflow (go back to zero), after approximately 50 days.

Parameters: None.

Returns: Number of milliseconds since the program started (*unsigned long*)

Example:

```
//declare outside loop()
long previousMillis = 0;

Inside loop:

unsigned long currentMillis = millis();
timeElapsed = currentMillis – previousMillis;
previousMillis = currentMillis;
```

Resource: https://learn.adafruit.com/multi-tasking-the-arduino-part-1/using-millis-for-timing

millis() example

```
void loop(){

  Serial.print("Time: ");
  time = millis(); //prints time since program started
  Serial.println(time);
 }
```

millis() versus delay()

- **millis():** is a better alternative to the delay function.
- Using the millis() function allows you to get work done during delay periods.
- delay() used instead of millis() will create issues for synchronous interrupts.

- https://www.arduino.cc/reference/en/language/functions/time/delay/

More on how the CPU reacts to ISRs

- Program counter (which holds the address of the current instruction) return address gets stored in memory stack
- Control branches to ISR
- The way that the CPU chooses the branch address of the ISR varies.
- There are two primary ways:
  1. **Non-vectored Interrupt**
     - Branch address is assigned to a fixed location in memory
  2. **Vectored Interrupt**
     - The source that does the “interrupting” supplies the branch information to the CPU- this way the source can access the interrupt directly - faster

When to use Interrupts or Polling?

Use Interrupt

If the event is:

- Asynchronous (i.e. you don't know when to expect it)
- Urgent
- Infrequent

Use Polling

If the event is:

- Synchronous (i.e. you know when to expect it within a small window)
- Not Urgent (i.e. a slow polling interval has no ill effects)
- Frequent (i.e. majority of your polling cycles create a 'hit')

Avoid Polling

- Polling typically wastes a lot of CPU cycles. The CPU may be in a loop to wait for the next polling.
- Polling drives up power consumption:
  - This may well be an issue for battery-powered embedded applications.
- Exceptions:
  - Polling is only for a short time
  - you can afford to sleep for a reasonable time in your polling loop.

Notable Features of an Interrupt Controller

- Peripheral devices are connected to a programmable interrupt controller
  - The controller can be an external device itself - separate chip that is part of the mother board.
  - The controller can be embedded into the chip design – faster on account of smaller paths that control signals must travel.

- A controller should end a thread in the least amount of clock cycles.
- At the minimum it would be:
  - 1 clock cycle for the original running thread to send a request for interrupt
  - 1 clock cycle to notify the correct interrupt thread that it is required to execute.

•Refer to Chapter 6 of the *Cortex M4 Processor Technical Manual* in the “Data Sheets > Microcontroller > ARM” folder in Brightspace

Question:

Q. If an interrupt is executing, and another higher priority interrupt gets invoked, what happens?

- Does the original, lower priority interrupt continue to execute until completion, followed by the higher priority interrupt?
- Does the lower priority interrupt suspend execution, and the higher priority interrupt now executes?

Some Interrupt links:

- http://www.gammon.com.au/interrupts
- http://www.engblaze.com/we-interrupt-this-program-to-bring-you-a-tutorial-on-arduino-interrupts/
- http://www.youtube.com/watch?v=CRJUdf5TTQQ
- http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=89843&start=all&postdays=0&postorder=asc
- http://www.avr-tutorials.com/interrupts/about-avr-8-bit-microcontrollers-interrupts
- http://www.jeremyblum.com/2011/03/07/arduino-tutorial-10-interrupts-and-hardware-debouncing/
- http://newbiehack.com/IntroductiontoInterrupts.aspx
- http://www.best-microcontroller-projects.com/hardware-interrupt.html
- http://www.dave-auld.net/?option=com_content&view=article&id=107:arduino-interrupts&catid=53:arduino-input-output-basics&Itemid=107