Interrupts

WiFi: meetingroom

Password: conference

code

Important Announcements

Required Learning Objectives

By the end of this session, you should be able to:

- describe what an interrupt is

- understand why they’re important

- write a good interrupt service routine

- use interrupts to control program flow

Introduction

Knock knock…

What are they?

What is an Interrupt in the context of micro-controllers?

An interrupt is made up of two things:

1. A condition
2. An interrupt service routine

An interrupt **condition** would be something like:

“On the falling edge of digital pin 2”

An interrupt service routine (ISR), might look something like:

```
void isr(){
	triggered = true;
}
```

Why though?

What would you use an interrupt for?

An interrupt can be used to briefly stop what the CPU is doing and divert it elsewhere, when a particular condition is met.

Think about the Arduino code you’ve written so far.

Think about the times when you’ve had the CPU “polling” i.e. checking whether something is true or not, almost constantly.

You can see that having the CPU spend most of its time “polling” its digital I/O pins is wasteful and sub-optimal, provided you have access to interrupts.

Ok, how?

Firstly, you should consider the condition you wish to “trigger” and interrupt service routine on.

When we looked at the “latching switch”, we were explicitly wanting to take some action if, and only if, the button was being pushed *right now*.

So we used two variables to track the button state over time, and in the moment when the new state was LOW, and the last state was HIGH, we tooks some action.

Rather than keeping track of those values in our main loop, it would have been more efficient to set up an interrupt that would be triggered by the “falling edge” of the IO pin the button was controlling.

Aside - What makes a good ISR?

Consider that new interrupt triggers may be ignored, or held off, if they occur while your Interrupt Service Routine is executing.

For that reason, best practice for writing ISRs is to keep them short.

If you have an interrupt attached to something that can happen very frequently and your ISR is very long and takes a long time to complete, at best you might miss some interrupts, but at worst the whole system could hang if new interrupts are held and then serviced when the previous one is finished.

So the key take-away is:

Keep your ISRs short and fast!

code

Example 1

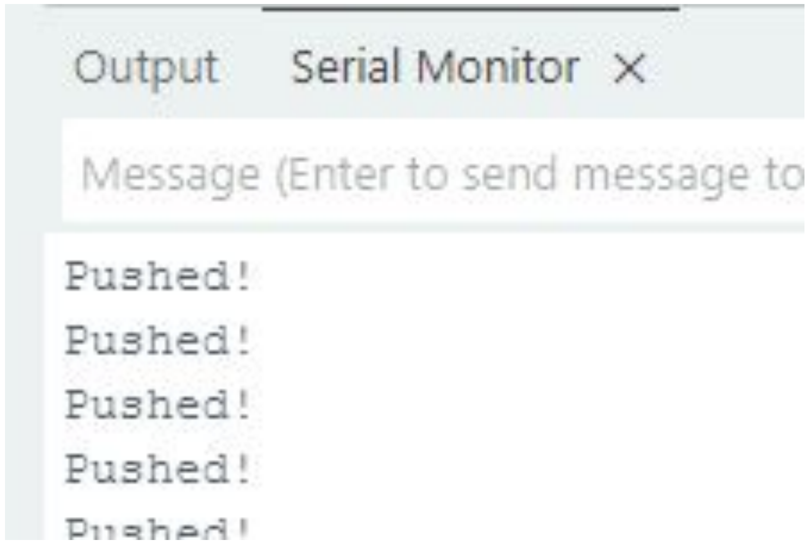
```
#define BUTTON_PIN 2
bool btn = false;
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  pinMode(BUTTON_PIN, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), pushed, FALLING);
}
void loop() {
  // put your main code here, to run repeatedly:
  if(btn)
  {
    btn = false;
    Serial.println("Pushed!");
  }
}
void pushed() {
  btn = true;
}
```

Example 1

Well, that technically *works*, but it's a bit funny.

It looks like I'm getting multiple triggers for what I'd call only one button push...

Why is that?



```
Pushed!
Pushed!
Pushed!
Pushed!
Pushed!
```

Bouncing

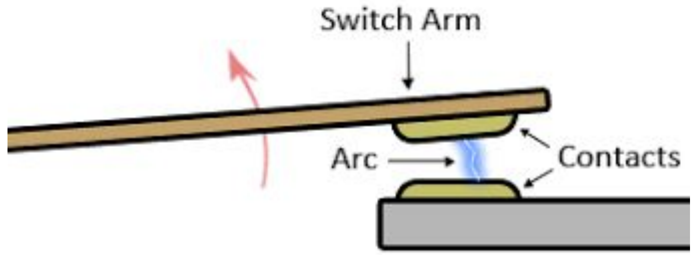
To explain why you might get lots of interrupt triggers when pushing the button, you need to look at the physical nature of a switch.

When you close a switch, you naturally have to bring one contact closer to another until they meet.

Unfortunately, physics is such that you'll never get a clean contact win one go, there'll always be some "bouncing" either from the contacts being very close but not close enough, or making connection briefly, then disconnecting.

So when you're dealing with physical switches, the signal will "bounce".

It'll go high and low a few times, then settle.



Bouncing

What can we do about that?

The most obvious thing to do is something we were already doing in the old “non-interrupt” code, which is to delay the CPU from taking any further action until the signal has settled.

That would *work* but it’s not ideal.

Remember we started using interrupts in the first place to avoid having the CPU sitting idle.

That said, let’s consider it for a moment…

Delay()

What’s wrong with using delay()?

Nothing really, but it’s not ideal.

When you execute this code:

```
loop(){
	Serial.println(“Starting”);
	delay(500);
	Serial.println(“Continuing”);
}
```

during that 500ms delay, the CPU is sitting idle. It can’t do anything else.

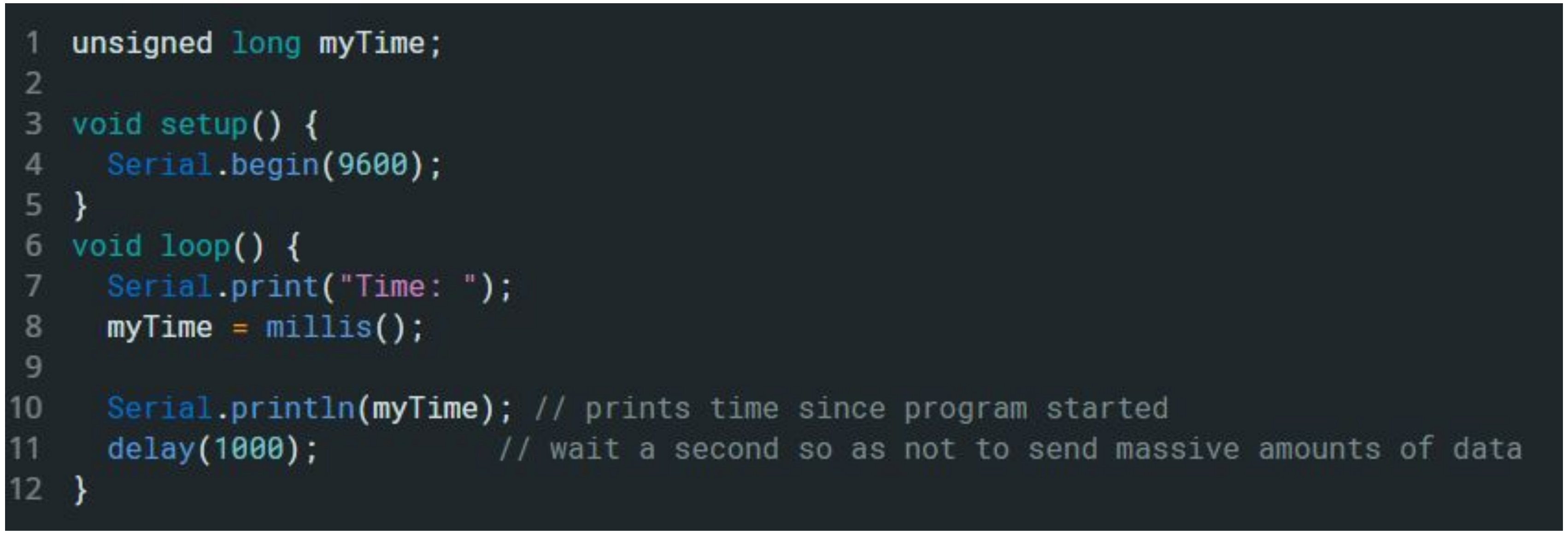
That’s the real issue with delay().

Time Keeping

So what can we do instead?

Consider the Arduino function millis()

millis() returns an “unsigned long” integer representing how many milliseconds have passed since the chip first powered on.



```
unsigned long myTime;

void setup() {
  Serial.begin(9600);
}
void loop() {
  Serial.print("Time: ");
  myTime = millis();

  Serial.println(myTime); // prints time since program started
  delay(1000);          // wait a second so as not to send massive amounts of data
}
```

Time Keeping

That’s handy indeed!

We can keep track of how long it has been since some other point in time without locking up the CPU!

Let’s figure out how to use this to *ignore* some interrupts for a certain period of time…

First we’re going to want to specify how long to ignore for.

Let’s start with 200ms.

That might be too long, or it might be too short, but it’ll get us started.

code

Time Keeping

```
#define BUTTON_PIN 2
#define IGNORE 200 //how many milliseconds to ignore interrupts for

bool btn = false;
unsigned long lastTrigger = 0;
```

See that we've defined IGNORE as 200, and also created an "unsigned long" to hold the last trigger time.

Time Keeping

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  pinMode(BUTTON_PIN, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), pushed, FALLING);
}
```

Setup remains the same.

Time Keeping

```
void loop() {
  if(btn)
  {
    btn = false;
    if((unsigned long)(millis() - lastTrigger) > IGNORE)
    {
      Serial.println("Pushed!");
      lastTrigger = millis();
    }
  }
}
```

With this modified loop, we’re only printing “Pushed!” for triggers more than IGNORE milliseconds apart.

Time Keeping

```
if((unsigned long)(millis() - lastTrigger) > IGNORE)
```

What’s happening here?

This says:

“If the value of millis() minus the value of lastTrigger, **cast to an unsigned long**, is greater than IGNORE, **then**”

Note that we have to explicitly cast the answer to “millis() - lastTrigger” to an unsigned long for reasons to do with the concept of “rollover”, which we might get into if there’s interest/time.

code

New and Improved!

```
#define BUTTON_PIN 2
#define IGNORE 200 //how many milliseconds to ignore interrupts for
bool btn = false;
unsigned long lastTrigger = 0;
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  pinMode(BUTTON_PIN, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), pushed, FALLING);
}
void loop() {
  if(btn)
  {
    btn = false;
    if((unsigned long)(millis() - lastTrigger) > IGNORE)
    {
      Serial.println("Pushed!");
      lastTrigger = millis();
    }
  }
}
void pushed() {
  btn = true;
}
```

What have we achieved?

Well, we've now got a button that we can respond to as fast as possible, without locking up the CPU, and without double triggering due to "switch bounce".

That's pretty good I think.

code

Should you bother?

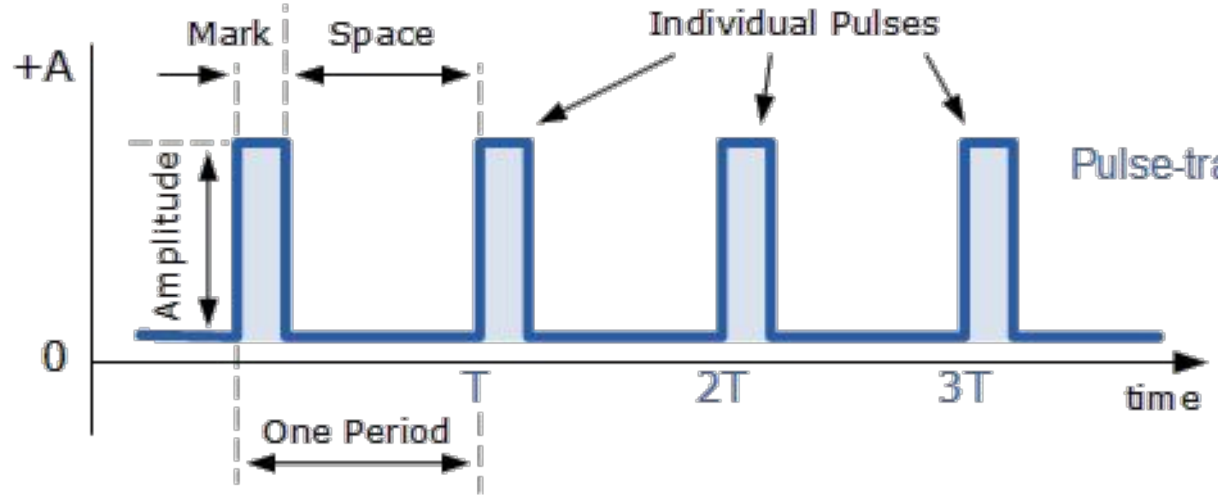
In the case of a single button, controlling something relatively inconsequential, no, probably not (unless you just want to practice using interrupts).

But consider something like a flow meter.
A flow meter measures the rate of flow of a fluid or gas through a pipe.
The output of a flow meter is generally a square wave pulse train where the frequency is proportional to the flow rate.

code

Flow Meter - Output

This is a pulse train.



Inside a flow meter is a “rotor”, a fin that spins round as fluid or gas passes through the pipe. Each time round, either a magnetic or optical sensor causes a “pulse”.

Flow Meter - Output

Clearly, you don’t want the CPU of whatever machine is monitoring the flow to have to sit there watching the flow meter signal line looking for pulses.

Flow meter “pulse counting” is a perfect use of interrupts. There’s normally no bouncing to deal with, and all you want to do inside the ISR is increment a “flow ticks” value, that you can later analyse to determine flow over time.

Other Good Cases for Interrupts

Can you think of any other good use cases for interrupts?

Other Good Cases for Interrupts

Precise Timing - A microcontroller’s timers will generally trigger interrupts when they “time out”

Waking a Microcontroller from Sleep - Some microcontrollers have power saving “deep sleep” modes that they can go into. Having the ability to attach a hardware interrupt to wake them back up again is very useful!

Communications - Often you’ll be able to configure an interrupt to trigger when a new byte is ready to receive.

Summary

In this session you have learnt:

- what an interrupt is with respect to a microcontroller

- why they're important and helpful

- why ISRs should be short and fast

- how to implement an interrupt in your Arduino code

- what some ideal use cases for interrupts are

code

Thank You!