Physics 120/220

Microcontrollers

Prof. Anyes Taffard
What is a microcontroller (µC / MCU)?

A microcontroller is a small computer usually implemented on a single IC that contains a central processing unit (CPU), some memory, and peripheral devices such as counter/timers, analog-to-digital converters, serial communication hardware, etc…

Microcontrollers are dedicated computers and designed for embedded applications. They are often low power devices.

- Emphasis is on size (small) and cost reduction
- The user interface is tailored to a specific application (e.g., buttons on a TV remote control)

In contrast, a microprocessor (µP) is multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in memory, and provides results as output.

- Emphasis is on flexibility and performance
- Generic user-interface such as keyboard, mouse, etc.
- Used in a PC, PDA, cell phone, etc.
Where are microcontrollers used?

Everywhere!

- Car
- Phone
- Electric toothbrush
- Microwave oven
- Copier
- Television
- PC keyboard
- Appliances
- etc...

There are probably about $O(100)$ MCU in a house.
**What is a microcontroller (cont.)?**

Analog or digital inputs/outputs

The crystal creates the “beat”, and the clock is the internal signal at the crystal oscillation frequency that synchronize all operations of the MCU.
Microcontroller Architectures

Microcontroller architecture refers to the internal hardware organization of a microcontroller.

Each hardware architecture has its own set of software instructions called assembly language that allows programming of the microcontroller.

Some of the popular microcontroller architectures

- Intel 8051
- Zilog Z80
- Atmel AVR:
  - We’ll focus on this as a representative example, since it is the core of the Arduino open-source development platform.
  - One of the first MCU to use on-chip flash memory for program storage, as oppose to one time programmable EEPROM

Is 8-bit still relevant

“n-bit”: the “n” refers to the data bus width of the CPU, and is the maximum width of data it can handle at a time:

- 8-bit MCU, 32-bit MCU
- PCs with 64-bit microprocessors are now common

Over 55% of all processors sold per year are 8-bit processors, which comes to over 3 billion of them per year!

8-bit microcontrollers are sufficient and cost-effective for many embedded applications.

More and more advanced features and peripherals are added to 8-bit processors by various vendors.

8-bit MCUs are well-suited for low-power applications that use batteries.
Atmel AVR family: ATmega 328P microcontroller

https://en.wikipedia.org/wiki/Atmel_AVR

**ATMega 328P:**

- CMOS 8-bit processor core
  
  - Optimize for low-level compilers (eg C)
  
  - Typically no OS is used

- 32 kBytes flash program memory

- Program and data memory
  
  - 1 kBytes EEPROM for saving data
  
  - 2 kBytes SRAM: hold variables

- Data converters (A/D C)

- Analog/Digital IOs
  
  - Switchable internal pull-up resistors

- 16MHz clock

Note: flash & EEPROM memory are non-volatile (information persists after power off)
Arduino Uno

Arduino development board come in several versions.

• We are using the Uno: http://arduino.cc/en/Main/ArduinoBoardUno

Based on Atmel AVR microcontroller and designed for rapid development and prototyping. Hardware contains everything you need. Uno provides:

• Convenient ways to power the microcontroller: AC adapter, USB or battery
• A system clock oscillator
• A reset switch
• An LED
• Standard connectors for peripherals and easy bread-boarding

Most capabilities are internal to the microcontroller.

Arduino schematic is relatively simple:


• Simple high-level C/C++ based programming language
  • Low level code is written for you: DigitalWrite(13,HIGH)
• Very easy to use
• Example code and projects
• Large online forums for support: http://arduino.cc/
Arduino Uno capabilities

Intel 286

Arduino

cost ~$20

~$8
Arduino Uno

- **Reset button**
- **USB Jack**
- **Rx + Tx LEDs**
- **Power LED**
- **Pin-13 LED**
- **14 Digital I/O Pins**
- **6 analog out with Pulse Width Modulated PWM (~)**
- **Power/GND Pins**
- **6 analog I/O pins**
- **Crystal oscillator**
- **Voltage regulator**
- **ATMEGA328 Microcontroller**
- **In Circuit Serial Programming (ICSP)**
- **ATMEGA328 Microcontroller**
- **Power Jack**
Handling the Arduino

As with any ICs, the board can be damaged with static electricity. Wash your hand after handling electronics, because of the lead/heavy metal.

If you use the Arduino with the breadboard/scope, don’t forget to use common GND.

How NOT to do it!

Proper Handling – by the edges!!!
# Arduino Uno Features

<table>
<thead>
<tr>
<th>Microcontroller</th>
<th>ATmega328</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Voltage</td>
<td>5V</td>
</tr>
<tr>
<td>Input Voltage (recommended)</td>
<td>7-12V</td>
</tr>
<tr>
<td>Input Voltage (limits)</td>
<td>6-20V</td>
</tr>
<tr>
<td>Digital I/O Pins</td>
<td>14 (of which 6 provide PWM output)</td>
</tr>
<tr>
<td>Analog Input Pins</td>
<td>6</td>
</tr>
<tr>
<td>DC Current per I/O Pin</td>
<td>40 mA</td>
</tr>
<tr>
<td>DC Current for 3.3V Pin</td>
<td>50 mA</td>
</tr>
<tr>
<td>Flash Memory</td>
<td>32 KB (ATmega328) of which 2 KB used by bootloader</td>
</tr>
<tr>
<td>SRAM</td>
<td>2 KB (ATmega328)</td>
</tr>
<tr>
<td>EEPROM</td>
<td>1 KB (ATmega328)</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>16 MHz</td>
</tr>
</tbody>
</table>
Running the Arduino IDE

You’ll install the Arduino software on your lab PC. Connect board via USB cable.

Select board

Select port
Failed compilation.

Note: the program won’t be loaded to board.
Programming the Arduino

Program == “sketch”
Written in a C language subset
Must have:

• Global variables declaration
• setup()
• loop()

setup():

• configures pin modes and registers.
• Setup “talk” to computer: Serial.begin(9600)
• run once are startup (or after reset)

loop():

• run the main body of the program forever

→ You can also write your own functions like in a C code

```c
const int ledPin = 13; // LED on digital pin 13

// the setup function runs once when you press reset or power the board
void setup()
{
    // initialize digital pin 13 as an output.
    pinMode(ledPin, OUTPUT);
}

// the loop function runs over and over again forever
void loop()
{
    digitalWrite(ledPin, HIGH); // turn the LED on (HIGH is the voltage level)
    delay(1000); // wait for a second
    digitalWrite(ledPin, LOW); // turn the LED off by making the voltage LOW
    delay(1000); // wait for a second
}
```
Using setup()

Digital pins can either be inputs or outputs

Input:

- The world outside the MCU determines the voltage applied to the pin.
  - Pin can be configured as INPUT_PULLUP
    - default state is HIGH
  - Input is the default pin configuration
  - High-impedance state: little current needed to move the pin from one state to another.
    Downside: pickup noise, capacitive coupling to nearby pin.

Output:

- Your program determines what the voltage on a pin is.
- Low-impedance state: can provide a substantial amount of current to other circuits.

```c
const int ledPin = 13; // LED on digital pin 13

// the setup function runs once when you press reset
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(ledPin, OUTPUT);
}
```

- `pinMode()`
  - Sets whether a pin is an input or output
- `ledPin`: global constant set to value 13.

Where can you find out about the commands?

Where can you find examples & libraries?
Using setup()

```c
// the loop function runs over and over again forever
void loop() {
  digitalWrite(ledPin, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000); // wait for a second
  digitalWrite(ledPin, LOW); // turn the LED off by making the voltage LOW
  delay(1000); // wait for a second
}
```

- **digitalWrite()**
  - Causes the voltage on the indicated pin to go HIGH or LOW
  - Note: must first configure the pin to be an output
    - To make pin go to 5V (HIGH):
      - `digitalWrite(pin, HIGH);`
    - To make pin go to 0V (LOW):
      - `digitalWrite(pin, LOW);`

- **delay()**
  - Cause the program to wait for a specific time
  - Unit: milliseconds
Designing a program

It is crucial that whatever program you write it is readable to others, well documented and organized.

- If you program lack the above, you won’t be able to tell what your program does 2 weeks later. You’ll have difficulty spotting errors.

**Few important guidelines:**

- Before writing your code, outline it in pseudo code
  - Structure the code: define independent pieces of code that execute particular task.
    - What each sub pieces of code is doing (input/outputs)
    - Logical flow of the code (avoid jumping back and forth)
- Prioritize clarity over minimizing lines of code
  - Clear program are easier to debug and it will save you time in the long run.
- Write optimized code
- Use ANSI (official standard for C programming)
- Test & Debug your code
  - Find test cases
  - If possible, devise different algorithms to solve the problem. Check that you get the same answers (doesn’t prevent making the same mistake twice tho!)
Good practices

• Comment your code
  ✓ Few lines describing what a piece of code does
  ✓ Few line describing a step within a piece of code (ie flow)

• Declare all the variables (avoid cryptic names)
  ✓ Helps to avoid surprises at the execution time.
  ✓ Make the code more readable
  ✓ Avoid v1 when you can use speed_of_light

• Declare the variables that are constant instead of hard coding values
  • Using preprocessor: #define ledPin 13
  • Using const: const int ledPin=13;
Finite state machine (FSM)

In many applications, the “circuit” has memory, and the output will depend on the present input(s) but also on the history: sequential circuit.

Finite state machines are the foundation of nearly all digital computation.

The state machine diagram is a behavior diagram which shows discrete behavior of a system through finite state transitions.

**Goal:** A circuit which sends a single high pulse when the button is pressed.

*The button could be held down for many clock cycles, but the circuit must not send out another pulse until the button is released or pressed again.*

3 states the circuit might be in:

- Waiting for button to be pressed
- Waiting for the button to be released
- Pressed! Sending out pulse

**Basic rules:**

- All possible triggers should be considered at all states
- A setup trigger defines the startup state
Finite state machine (FSM) cont.

- Connect these states with arrows that indicate how the button value would move the system from state to state.
- These transitions will occur each clock cycle.
- Define also the starting state.

Each state includes what to do for both button conditions.

If the button release happens in the clock period immediately following the press event.

You can only stay in this state one cycle since all transitions lead away.
Pulse Width Modulation (PWM) is a technique to get analog results with digital means. PWM generates a square wave with a pulse width that varies between 0% and 100% of the pulse period. The % of the on-time is proportional to the signal voltage.

- For pins 3, 9, 10, 11 the modulation frequency is ~488Hz
- For pins 5 and 6 it is about 977Hz.

The PWM output can be easily converted to an analog voltage level by using an RC low pass filter.

Note: once the output waveform is started by the `analogWrite()` function, it will keep running in parallel of your code. A PWM signal generated by the MCU hardware continues to run totally independently until you change the duty cycle or use `analogWrite(0)` to turn it off.
Analog input pins

Arduino Uno contains an onboard 6 channel analog-to-digital (A/D) converter.

The converter has 10 bit resolution

An analog voltage on an analog pin (eg A0) will be converted to a 10-bit integer from 0 to 1023 using a successive ADC:

- uses a comparator to successively narrow a range that contains the input voltage
- $5V/1024 = 4.9mV$

Analog inputs are possible on pins A0 → A5.

```cpp
pinMode(A0, INPUT);
int value = analogRead(A0);
```
Serial port: UART

Used for communication between the Arduino board and a computer. Arduino boards have at least one serial port (UART: Universal asynchronous receiver/transmitter).

It communicates on the digital pin-0 (RX) and pin-1 (TX) as well as with the computer via the USB.

Build-in serial monitor (open window via: Menu → Tools → Serial Monitor)

To start the serial communication via USB, use: `Serial.begin(9600)`

- 9600 is the baud rate: 9600 bits/s

To print:

- `Serial.print("Hello world \n");`
- `Serial.println(value);`
Few things to keep in mind when programming the Arduino

- SRAM=2kB (max 2048 bytes)
  - Not that much space to store variables.
  - int take 2 bytes, while byte uses only one and ranges from 0-255.
    - Use the smallest data type.
    - If you don’t need to modify some data during your sketch, store them in the flash memory: PROGMEM

- Size of int is 2 bytes: value ranges [-32,767 – 32,768].
  - Don’t go over bound with calculations.
- float has 4 bytes, but has 6-7 decimal digits of precision.
  - Floating point numbers are not exact: watch out for floating-point arithmetic
  - Floating point math is slower: use integer math if possible.

- Commands/loop execution time:
  - Week 8 lab: analogRead execution time
  - Use serial monitor to debug