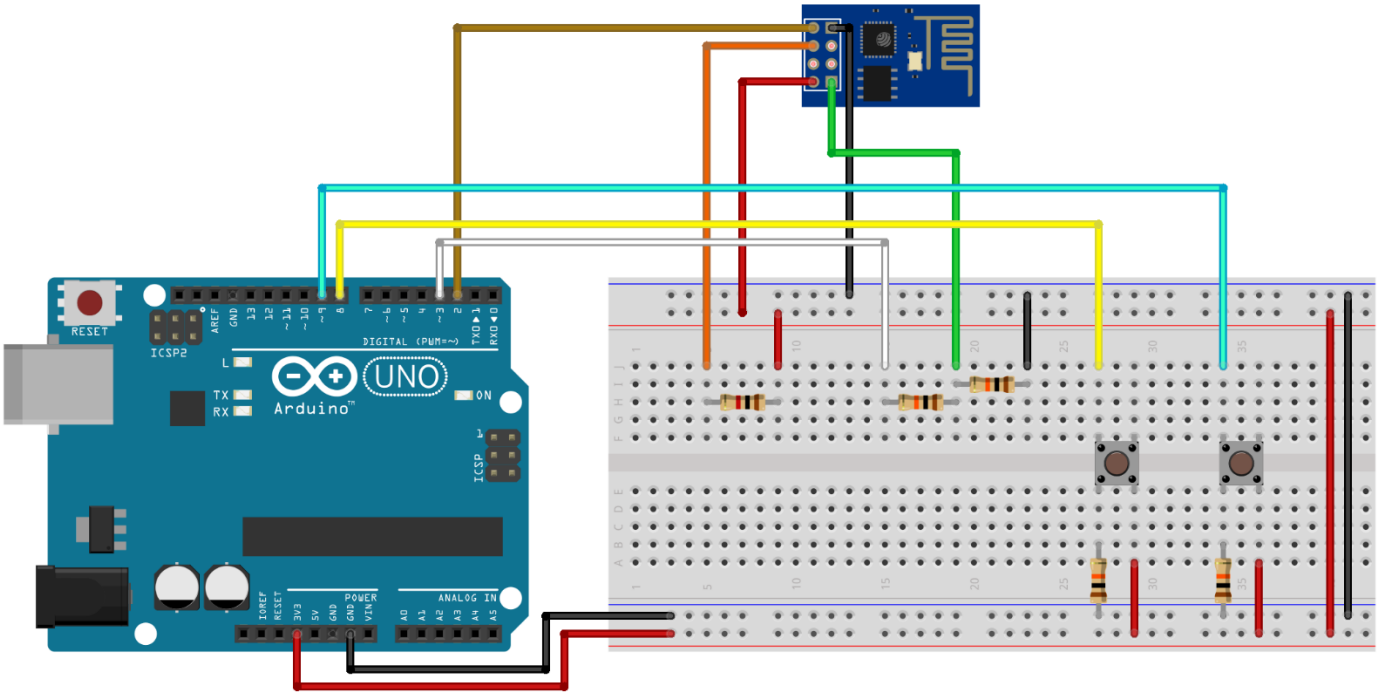
HBO ICT
THEMATIC SEMESTER INTELLINGENT DEVICES

Reader for introduction to devices
TICT-VDEIN-17



Micro-controllers

2017-2018

Author(s)	**Wouter van Ooijen**
Date	**2017-09-13**
Version	**0.2 work-in-progress**

1. Contents

2. Introduction

This reader provides information for dealing with micro-controllers in the context of small intelligent devices. It is definitely not intended to educate the reader to the level of an embedded software designer, but it aims to give a common language for communication with such a specialist.

This reader is intended for a one-week course: 3 hours instruction, and 3 hours lab work.

The full code for the mentioned Arduino projects is available from https://www.github.com/wovo/vdein-examples.

3. Micro-controllers

3.1. Overview

A micro-controller is a chip (integrated circuit) that combines the essential elements of a computer (CPU, RAM memory, ROM memory, and I/O) on a single chip. The range micro-controllers extends from very small and inexpensive ones, with for instance 32 bytes of RAM and 1kb of ROM running at 1 MIPS (Million Instructions Per Second), up to expensive and powerful chips with 100kb's of RAM running at up to 1000 MIPS.

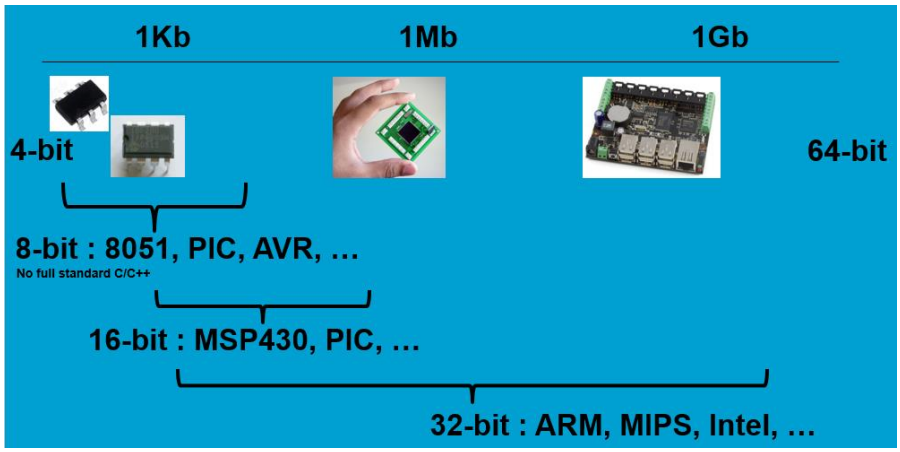


Figure 1: range of embedded system memory sizes

The small chips can be found in very simple devices like a radio controlled toy car, a melody birthday card, or a flashlight that has a blink function. The powerful ones are used to control industrial processes, airplanes, or the motor of a modern car. The 'seriousness' of micro-controller application and embedded systems in general varies widely, from toys and gift-cards to airplanes and nuclear power stations.

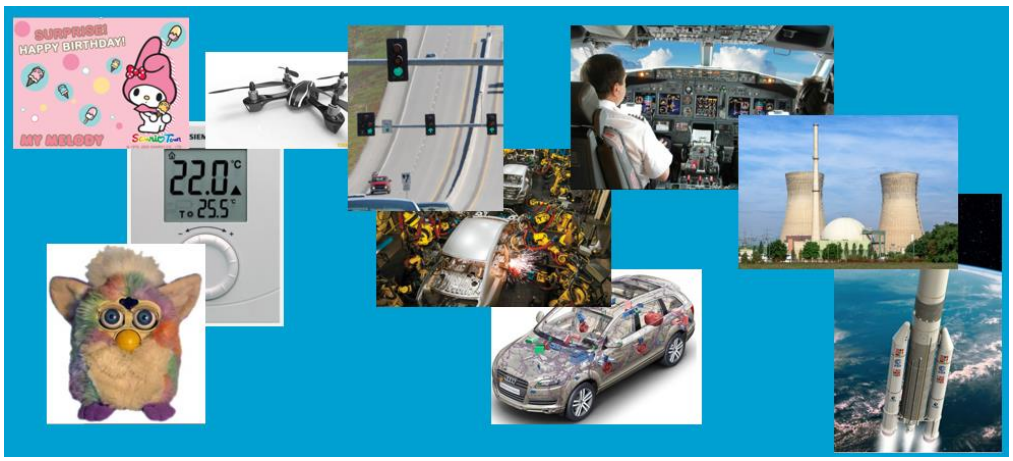


Figure 2: range of embedded system 'seriousness'

Besides cost a major driver for using a small micro-controller instead of a bigger one (or even a computer system built from separate components) is power. Modern intelligent devices, especially sensors, are often powered by a battery or another low-power source. The price (or weight, volume, etc.) of the power source is often the major design and cost driver of the product. A small micro-controller can typically be put in a (very) low power mode, waking up (and consuming significant power)

only let's say for 10 seconds each hour. This is the way remote sensors can work for a year or longer on a battery.

3.2. Memory

A computer system can have three types of memory:

- Volatile (RAM)
- Non-volatile read-only (ROM) or mostly-read (some types of FLASH)
- Non-volatile read-write (hard disks, solid-state-disks, EEPROM, some types of FLASH)

A computer system needs RAM and some form of non-volatile memory. In modern micro-controllers the non-volatile memory is generally FLASH. A flash memory section is degraded each time it is erased (which must be done before it is re-written). After a number of erase actions the flash is worn out. How many cycles it can endure depends on the cell size: smaller cells wear out quicker. A flash memory card is designed with a more memory than it is advertised for, and a controller that detects defect sections and replaces them with spare ones. Flash in micro-controllers generally doesn't use that technique, hence it can endure only a limited number of erase (and write) cycles. This makes it unsuitable for storing data that changes frequently. For this purpose some micro-controllers have a small amount of EEPROM memory, which behaves much like flash but has larger memory cells (hence the lower amount of memory), that can endure much more erase/write cycles. When no-chip EEPROM is available an external EEPROM chip can be used.

The optimal chip manufacturing processes for RAM, flash and logic (the CPU) are different, hence a chip that contains a mix of these parts must make a compromise. With the current techniques a big micro-controller can have up to ~ 100 kb of RAM and ~ 1 Mb of (flash) ROM. This is enough for a LOT of code, but not for a modern Operating System (Windows, Linux, etc.). Hence systems that do run such an OS (PC, Mobile, and single-board computers like a Raspberry Pi) use separate chips for CPU, RAM and non-volatile memory.

3.3. Embedded

The term 'embedded' is often used, but it doesn't tell much about the hardware involved. It refers to the way a (computer) system is used, not necessarily to its properties. All types of computers, including PCs, laptops, and single-board computers can be used as embedded systems. Three broad classes of embedded hardware can be identified:

- PC/Raspberry Pi level hardware, running a general-purpose operating system like Windows or Linux;
- Single-board computers, often running a real-time Operating System (RTOS);
- Micro-controllers, running one application program, without any separate Operating System (bare metal).

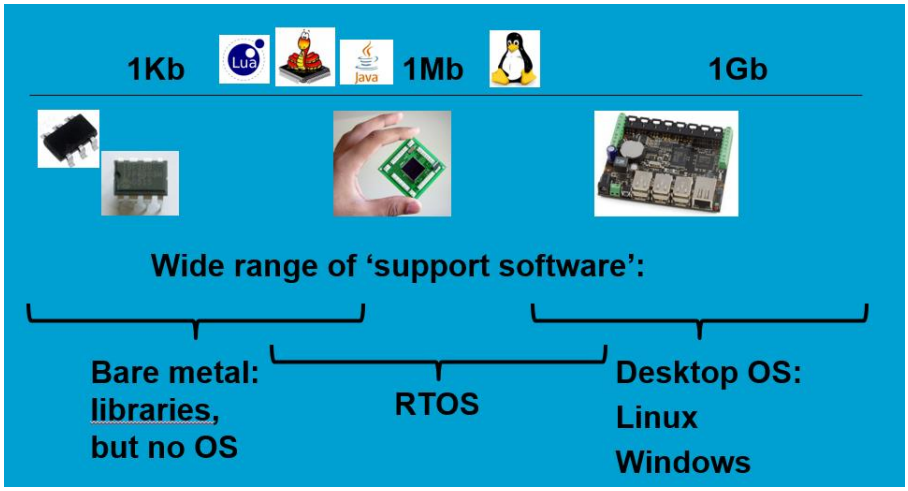


Figure 3: range of embedded system 'support software'

4. Application development

4.1. Interpreted languages

When an application is run by an interpreter the system stores both the application itself, and the interpreter (which can be a large application in itself!). When the application runs the interpreter reads the instructions one-by-one and executes them. This is beyond the resources of most small micro-controllers, but it is feasible for mid-range chips with typically 10Kb or more of RAM and up to 100kB of ROM.

An interpreted application will run slower and require a more expensive micro-controller than a compiled application, but this can be more than outweighed by the ease at which applications (especially smaller ones) can be developed. Beyond the interpreter, which is often pre-programmed into the micro-controller by the supplier, you need only a connection to your PC and a terminal application running on the PC.

Examples are the ESP8266 chip with a Lua interpreter, and various ports of micro-Python. These chips often provide a built-in library for easy use of the special features of the chip, for instance wireless communication (Wi-Fi, Bluetooth, Lora WAN).

4.2. Compiled languages

When speed or product-cost are important, an application is developed and compiled on the host system (laptop, PC) and only the compiled 'image' is downloaded to the micro-controller where it is stored in its ROM. This is called cross-development: the development environment runs on one system, but generates an application for another system. A wide range of programming languages is used for embedded systems, but for (smaller) micro-controllers C, C++ dominate, and sometimes even assembly is used.

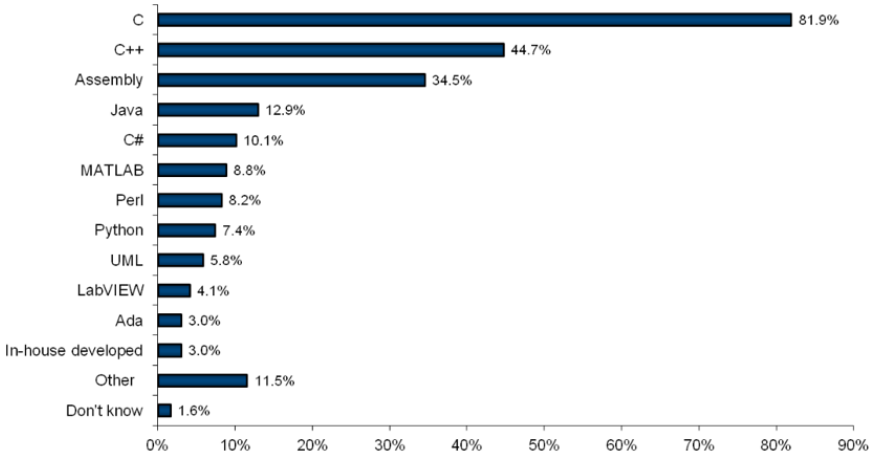


Figure 4: programming languages used for embedded systems

4.3. Downloading

When an application image has been built on the host system, it must somehow be transferred to the micro-controller to be stored in its ROM. The micro-controller manufacturer often provides a very primitive and awkward protocol for doing so, requiring some gadget between the PC and the micro-controller to implement this protocol. This makes economic sense: putting more hardware on the chip to implement an easier protocol would make the chip more expensive, which raises the unit price of the final product. A gadget that is required for programming raises the development cost, which is much less important for high-volume products (which is where the profit lies for a micro-controller manufacturer). Clones of programming gadgets for the more popular micro-controllers are often cheaply available, but for the more exotic products they can still be very expensive.

Most modern micro-controllers can write to their own flash memory. This makes it possible to create a micro-controller application that resides in a corner of the flash memory, can take control on startup, receives an application image over some communication line, and stores it in flash (without overwriting itself). This is called a bootloader.

Some manufacturers omit the classic interface for downloading, but put a bootloader in a (non-erasable!) part of the ROM/flash memory. Bootloaders for remote systems can be very complex and sophisticated. Examples are remote sensors and satellites.

One key characteristic of any bootloader is how the bootloader decides between waiting for a new image, or running the existing image. When you have physical access to the chip you can use a pin to instruct the bootloader what to do, but for a remote system (that is only reachable by a communication channel, for instance radio) this is generally not an option. In such cases the bootloader is often not activated at each startup, but only when the application chooses to do so. An unfortunate consequence is that an erroneous application can render the system useless, so one must be very careful with downloading a new application.

5. Arduino hardware

5.1. The Arduino Uno

The Arduino Uno is a single-board computer with an ATmega328P micro-controller, the components required for this chip to run, and an USB-to-serial chip for communication with a PC (using a virtual serial port on the PC). The ATmega chip can be programmed directly using a programmer like the USBasp, but the normal way to download an application to the chip is to use the USB-serial interface and the Arduino-specific bootloader that is pre-loaded in the chip. The ATmega328P is a small and cheap micro-controller, with 2 Kb RAM and 32 Kb FLASH ROM.

By itself, an Arduino Uno is almost useless: it has one on-board LED[1] and its serial connection to the PC, so you can use it to blink the LED or to say “Hello world”. To do something more the Uno has four so-called header strips that provide access to the power and GPIO pins of the micro-controller. You can use these strips to make connections to external hardware using for instance jumper wires and a (solderless) breadboard.

Figure 5: the Arduino Uno

For commonly used hardware there are the so-called shields, which can be plugged directly into the connectors. This is very convenient when (only) one shield is used. Some shields provide header strips similar to the Arduino itself, for plugging in another shield. This way you can create a stack of shields. This might seem a good idea, but in most cases a shield uses specific GPIO pins, and even two shields have a good chance of both using at least one common GPIO pin, which in most cases doesn’t work.[2]

[1] It has 3 more LEDs, but they indicate power, transmitted data, and received data: they are not under the direct control of the micro-controller.

[2] Exceptions are pins that are used for I2C and SPI, which are protocols that are designed to share pins with a number of peripheral chips.

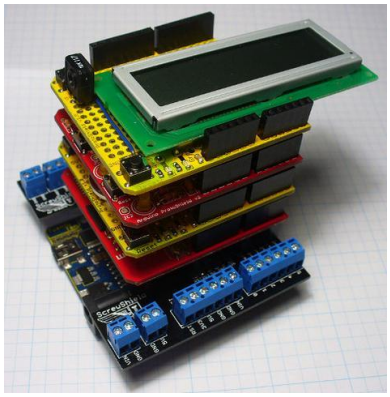


Figure 6: an Arduino Uno with a (ridiculous) stack of shields

5.2. Other Arduino's and alternatives

Figure 7: The Arduino Uno (again)

The original Arduino, the Arduino Uno, was first designed around 2005, and it uses a micro-controller that was popular, cheap, and moderately powerful at that time. Since that time the price more powerful micro-controllers has dropped significantly, and alternatives to the original Arduino have been developed.

Figure 8: The Arduino Mega 2550

The Arduino Mega 2550 uses a micro-controller from the same family as the Uno, but the chip has more RAM, FLASH, and GPIO pins. The board is larger, mainly to accommodate the extra pin headers. The Mega headers are a superset of the Uno headers, hence Uno shields can be used on a Mega.

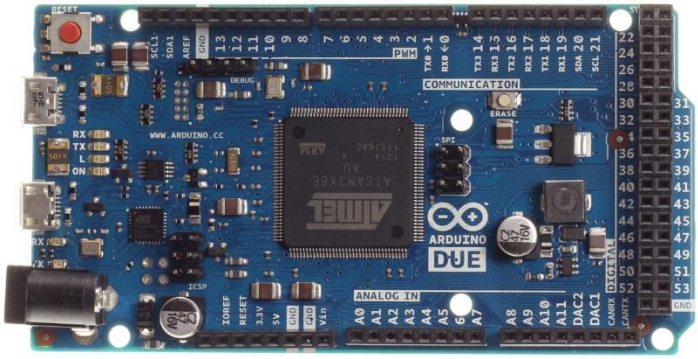


Figure 9: the Arduino Due

The Arduino Due uses a much more powerful Cortex micro-controller with much more RAM and FLASH than the Mega. The board uses the same headers as the Mega, but as the Cortex chip runs on 3.3V (instead of the 5V of most other Arduino's) not all shields are directly compatible with the Due. On the other hand, many modern peripheral chips (sensor etc.) run on 3.3V, so they are actually easier to use with a Due than with a 5V Arduino.

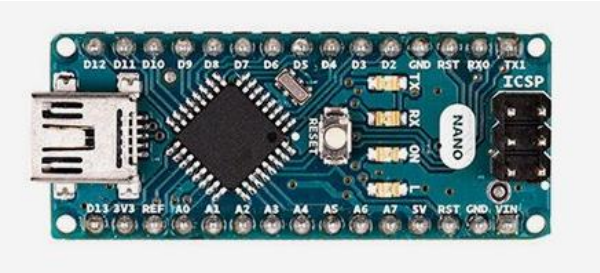


Figure 10: the Arduino Nano

The Arduino Nano is a development in the other direction: it uses the same micro-controller as the Uno, but reduces the board to the minimum and omits some components (power regulator) to reduce the price.

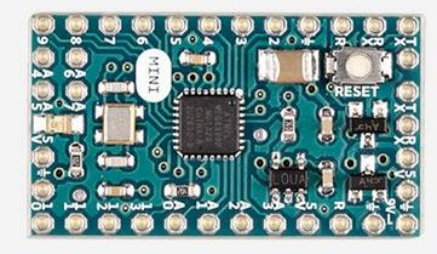


Figure 11: the Arduino Micro (pro)

The Arduino Micro is an even more reduced version of the Uno: compared to the Nano it omits the USB interface, hence it must be programmed using an external USB-to-serial converter.

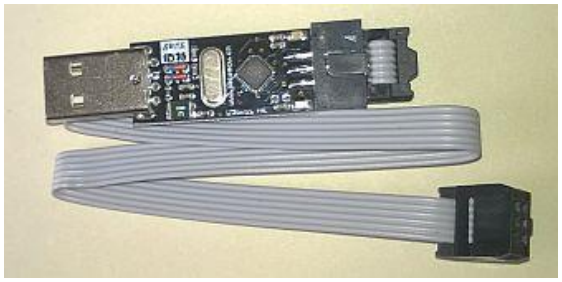


Figure 12: USBasp programmer for AVR chips

Beware: The Nano and Micro are sometimes sold without the Arduino bootloader pre-programmed. In that case you must use a programmer gadget to download the bootloader first, or forgo the bootloader and load your application directly. Fortunately, a programming gadget for these chips (USBasp) is quite cheap.

Figure 13: ESP8266 modules

The popular ESP8266 chip is a combination of a micro-controller[3] and a Wi-Fi transceiver. It is designed to be combined with an external FLASH chip that stores its application (hence it is not a pure micro-controller, but who cares). The manufacturer provides a standard application, which makes it easy for a connected micro-controller (for instance an Arduino) to use Wi-Fi. But there is also a Software Development Kit (SDK) provided by the manufacturer for the ESP8266 that makes it possible to create an application that runs on the ESP8266 without any extra micro-controller. The Arduino IDE also supports the ESP8266. Interpreter applications are available for the ESP8266 that run Lua or (a reduced version of) Python.

Figure 14: an integrated wireless development module

The PyCom is an example of a module that combines a micro-controller running and interpreter (in this case Python) with various wireless interfaces (in this case Bluetooth, Lora WAN, and Wi-Fi). Such a module makes prototyping much cheaper, but it is more expensive than separate components and the programmer has less control over the code that runs on the chip. Hence choosing between an easy-to-use

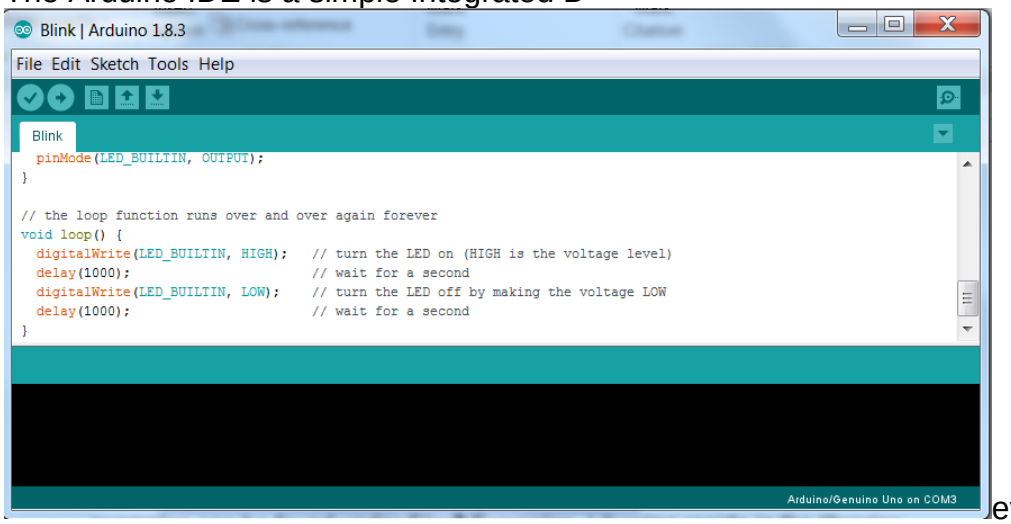
[3] The micro-controller part of an ESP8266 is comparable in CPU power to a Cortex-M, as used the Arduino Due, but it uses a totally different instruction set architecture.

but more expensive to buy module, and a specially designed and built module (higher development cost for both hardware and software, but lower per-module production cost) can be an interesting technical/management descision.

6. Arduino programming

6.1. The Arduino IDE

The Arduino IDE is a simple Integrated D



evelopment Environment for the Arduino's (and some other targets, including the ESP8266). Applications for the Arduino are called sketches. An Arduino IDE window has one sketch open, but you can have multiple IDE windows open to work on more than one sketch.

Figure 15: the Arduino IDE

The IDE has a number of built-in libraries and examples, check https://www.arduino.cc/en/Reference/Libraries for documentation. The examples can be found under File➔Examples. Libraries reside in the libraries directory (on my PC C:\Program Files (x86)\Arduino\libraries). This is the place where you put third-party libraries. The IDE uses the C++ compiler, but most sketches and libraries don't use much OO style coding.

6.2. Example sketches

The Arduino uses a simple application framework. The user supplies two functions: setup() and loop(), both without any parameters. The framework calls setup() once at the start of the application, and then calls loop() repeatedly.

```
void setup() {
  // put your setup code here, to run once

}

void loop() {
  // put your main code here, to run repeatedly

}
```

Arduino sketch template	template

6.2.1. Blink

The first thing to do on any embedded system is to blink a LED. The blink1 example shows how this can be done. The setup() configures the LED_BUILTIN pin (the pin

number of the on-board LED) as output. The loop() makes that pin high, waits half a second, makes the pin low, waits another half a second, and finishes. The framework calls the loop() repeatedly, which causes the LED to blink.

```
void setup() {
   pinMode( LED_BUILTIN, OUTPUT );   // initialize digital pin LED_BUILTIN as output
}

void loop() {
  digitalWrite( LED_BUILTIN, HIGH ); // turn the LED on (HIGH is the voltage level)
  delay( 500 );                      // wait for half a second
  digitalWrite( LED_BUILTIN, LOW );  // turn the LED off by making the voltage LOW
  delay( 500 );                      // wait for half a second
}
```

Blink the on-board LED	blink1

This simple style of programming works fine for a single task, but for doing more than one thing a different style must be used, in which no delay() calls are allowed in the loop(). This is shown in the blink2 example. A global variable stores the last time (obtained from the micros() library function) the LED was toggled. In the loop, the current time is obtained. If the last toggle was more than 500 ms ago, the LED is toggled and the last toggle moment is updated.

```
void setup() {
   pinMode( LED_BUILTIN, OUTPUT );   // initialize digital pin LED_BUILTIN as output
}

bool level = 0;                      // the current state of the LED, initially low
long int last = micros();            // the startup ‘moment’

void loop() {
  long int now = micros();           // get the current time
  if( ( now - last ) > 500000 ){     // more than 500 ms since the last LED change?
     level = ! level;                    // toggle LED state
     digitalWrite( LED_BUILTIN, level );  // write it to the LED
     last = now;                         // remember this moment in time
  }
}
```

Blink without waiting in the loop()	blink2

The blink3 example shows how the LED blinker can be abstracted into a class. A class is a container that bundles related variables and functions, and can restrict outside access to these variables and functions. The class constructor (a function with the same name as the class) takes the pin number and the interval time, and saves these values in private class variables. The class has an update() function that takes the current time as argument, and does the updating. Note that the interval is expressed as unsigned long, because a plain int on the Arduino Uno is only 16 bits.

```
class blinker {
private:

   bool level;
   long int last;
   long int interval;
   int pin;

public:

   blinker( int p, long int t ){
      level = 0;                        // initial LED level: low
      last = micros();                  // initial last-update moment

      pin = p;                          // the number of the LED pin
      interval = t;                     // the microseconds between pin toggles

      pinMode( pin, OUTPUT );           // make the pin an output
   }

   void update( long int now ){
     if( ( now - last ) > interval ){   // last update is more than 'interval' ago?
        level = ! level;                   // toggle pin level
        digitalWrite( pin, level );        // write it to the pin
        last = now;                        // remember when we did this
     }
  }

};
```

A separate FSM blinker class | blink3

Now the application itself is reduced to the creation of the (global) blinker object with the appropriate parameters, and the calling of the led.update() function in the loop(). What was previously done in the setup() is now done in the constructor call. The delay must be expressed using an UL suffix, because otherwise the value would overflow the 16 bits of an int.

```
blinker led( LED_BUILTIN, 500 * 1000UL );  // create a blinker object

void setup(){}                             // no (other) setup actions needed

void loop() {
   long int now = micros();                // get the current time
   led.update( now );                      // the blinker object must update itself
}
```

Using the blinker class | blink3

This style of programming is often call a Finite State Machine (FSM).

6.2.2. Switch

To connect components to an Arduino a breadboard is used. We use either a separate solderless breadboard, or one on a breadboard-shield. On such a breadboard the columns of holes are connected.

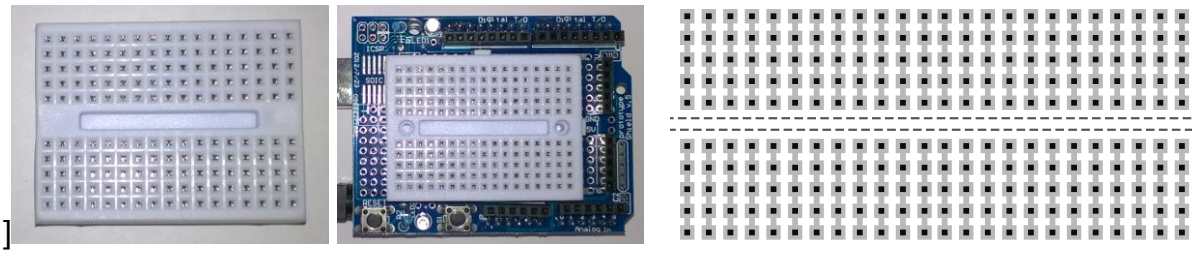


Figure 16: a separate (mini) breadboard, breadboard shield, and the connections inside the breadboard

An Arduino pin can be used as input, for instance to read the status (pressed or not pressed) of a switch. To do this, the switch is connected to the pin and to ground (or to the (5V) power). So when the switch is pressed, the pin is connected to the 5V, so its voltage level is 5V.

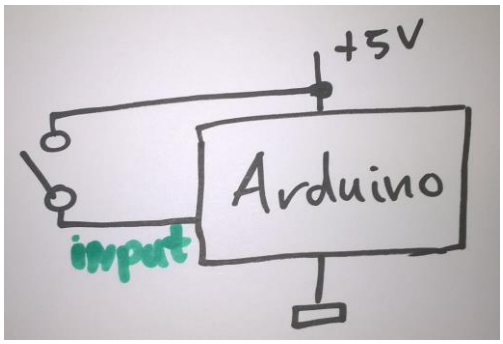


Figure 17: a switch connected to an Arduino (input) pin

So far so good, but when the switch is *not pressed* the pin is not connected to anything. An Arduino input pin is very sensitive, in this situation it can pick up stray signals, which means that its level is not predictable: it can be low or high, or it can vary rapidly. To get a defined level on the pin when the switch is open, a resistor is required to 'pull' the pin level in the opposite direction when the switch is not closed. (There is no specific symbol for an Arduino, I use a box with the relevant connections shown.)

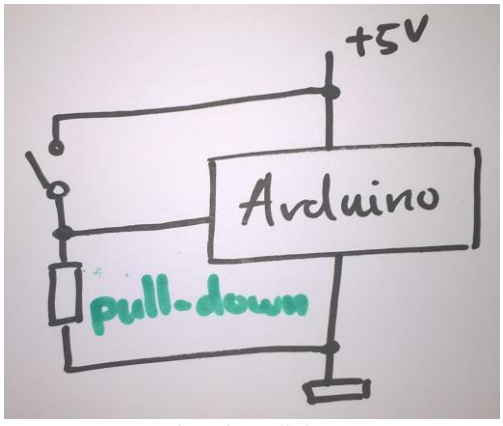


Figure 18: a switch and a pull-down resistor connected to an Arduino (input) pin

For convenience, the Arduino chip has built-in pull-up resistors of ~ 100kΩ that can be enabled in the sketch, hence we can omit that resistor. (But the other side of the switch must now be connected to the ground, not to the power.)

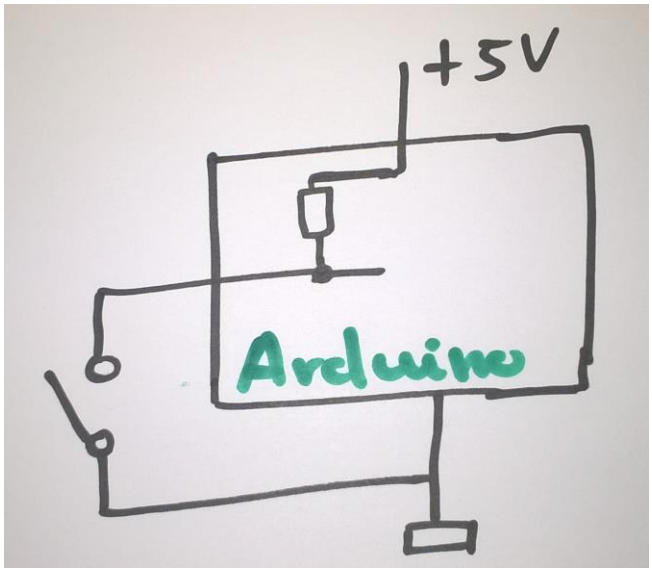


Figure 19: a switch (to ground) and the internal pull-up resistor in an Arduino

The LED needs a series resistor to limit its current.

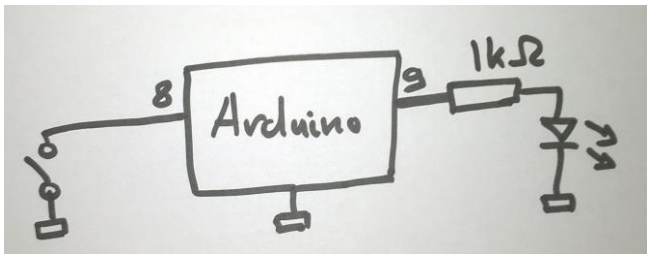


Figure 20: switch-and-LED circuit diagram

The circuit is built using jumper wires. The convention is to use black (or as a second choice blue) for the ground wires, and red (or as a second choice orange) for power supply which (of which there are none in this simple circuit). Don't use these colors for other purposes if you can avoid it.

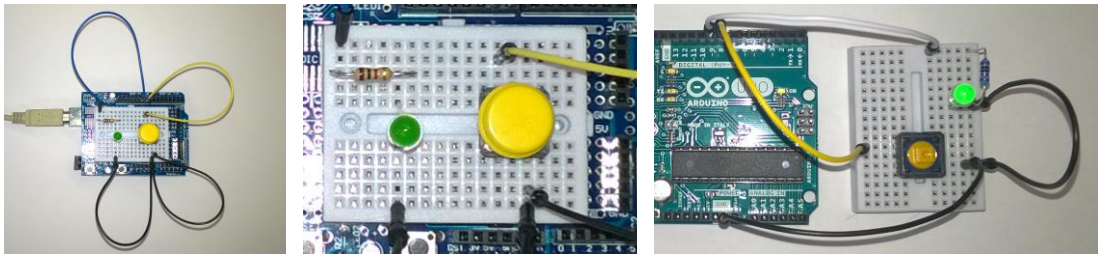


Figure 21: switch-and-LED setup

In the sketch, the first two lines define the pins that will be used. In the setup, the pull-up resistor is activated by setting the pin mode to INPUT_PULLUP. The switch input is active low (high when the switch is not pressed, low when the pin is pressed).

The LED is (also) connected between the (output) pin and the ground, but in this case this means that the pin is active high. To make the LED light up when the switch is pressed, the value read from the pin is inverted by a '!' before it is written to the output pin.

```
const int sw = 8;                                  // the pin for the switch (d8)
const int led = 9;                                 // the pin for the LED (d9)

void setup(){
   pinMode( sw, INPUT_PULLUP );                    // switch pin: input with pull-up
   pinMode( led, OUTPUT );                         // led pin: output
}

void loop() {
   digitalWrite( led, ! digitalRead( sw ));        // read switch, invert, write to LED
}
```

Switch-controls-LED sketch	switch1

6.2.3. Analog input

An Arduino Uno can read an analog value using one of its analog input pins (A0 ... A5). The analogRead function returns a value in the range 0 .. 1023 (10 bits) for voltages in the range 0V .. 5V. The following sketch prints the analog value on the serial line to the PC. You can see the result on the PC using the serial terminal tool. The Serial.begin() call is needed to initialize the serial line to the correct baudrate.

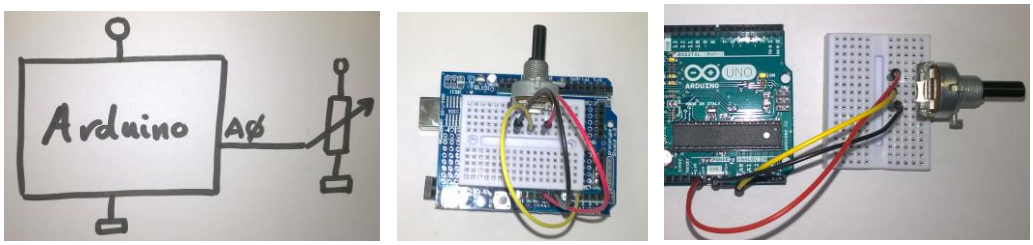


Figure 22: analog input

```
const int potentiometer = A0;                 // pin for the voltage input: a0

void setup() {
  Serial.begin( 9600 );                       // initialize communication to the PC
}

void loop() {
  int p = analogRead( potentiometer );       // read the voltage input
  Serial.print( "\nsensor = " );              // print the value to the PC
  Serial.print( p );
  delay( 200 );                               // wait a bit
}
```

Read a potentiometer and log the value	potentiometer

Start the serial monitor using Tools ➔ Serial Monitor ansd you can see the values. When you turn the potentiometer fully left the value will be (close to) 0, for fully right it will be (close) to 1023.

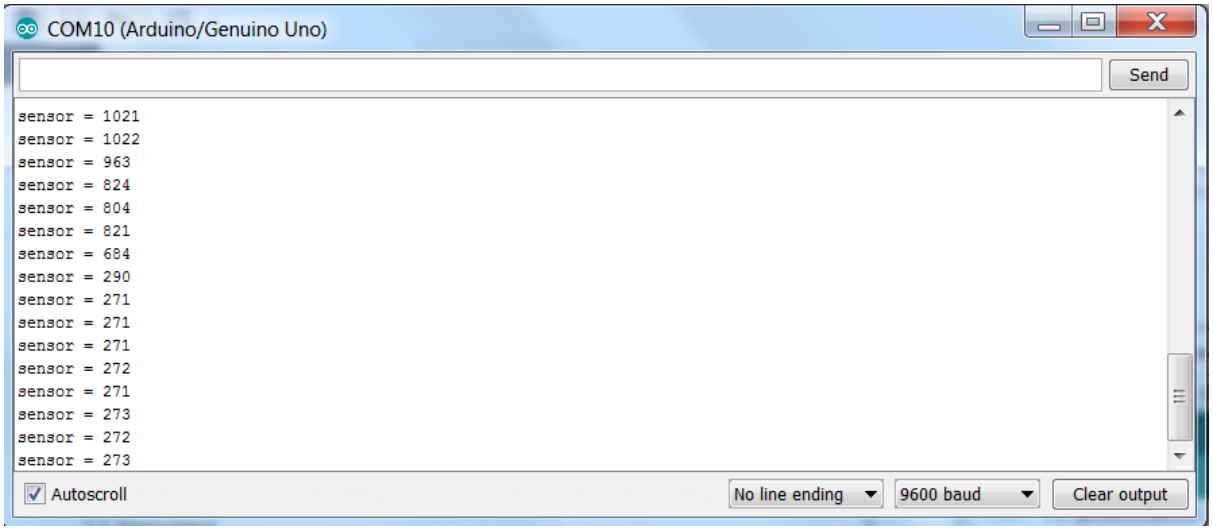


Figure 23: Serial Monitor output

6.2.4. PWM

The brightness of a LED can be controlled using PWM. The Arduino Uno library has some PWM functionality, but we can also do PWM in software. To demonstrate this, we dim a LED using the input from a potentiometer.

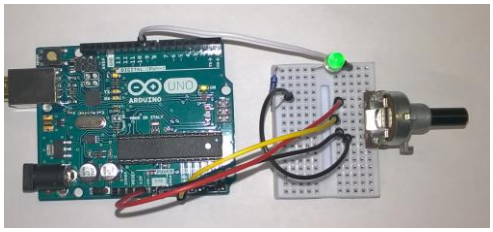


Figure 24: potentiomtere and LED setup

The dimmer class is initialized with the pin on which the PWM signal must be created. This pin is configured as output. The pwm also stores the percentage of the time that the LED must be on. This value is initially zero and can be changed by the set() function.

The update() function checks whether enough time has elapsed since the last time it did something. If so, it increments the count variable modulo 100 (it counts from 0 to 100 and then starts over again). The LED is set on if (and only if) the count is smaller than the percentage.

```
class pwm {
private:
   int pin, percentage, count;
   unsigned long last = 0;

public:

   pwm( int p ){
      pin = p;
      percentage = 0;
      count = 0;
      pinMode( led, OUTPUT );
   }

   void set( int p ){
      percentage = p;
   }

   void update( unsigned long now ){
      if( now - last > 10UL ){
         count = ( count + 1 ) % 100;
         digitalWrite( led, count < percentage );
         last = now;
      }
   }
};
```

PWM LED dimmer class	pwm

The rest of the application defines the pins involved and creates the pwm object. The loop calls the pwm update, and then reads the potmeter and passes the A/D value, converted to the range 0 .. 100, to the pwm object.

```
const int led = 9;
const int potmeter = A0;

pwm led_dimmer( led );

void setup(){}

void loop(){
   unsigned long now = micros();
   led_dimmer.update( now );

   int val = analogRead( potmeter );
   int p =  map( val, 0, 1023, 0, 100 );
   led_dimmer.set( p );
}
```

PWM LED dimmer sketch	pwm

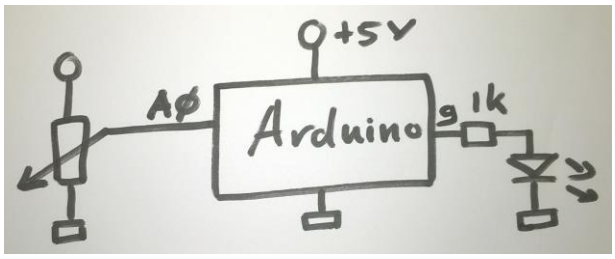
Map is an Arduino library function that converts a value in a range linearly to a value in another range. The documentation of this function (and a lot of others) can be found at https://www.arduino.cc/en/Reference/Map.

map(value, fromLow, fromHigh, toLow, toHigh)

Description

Re-maps a number from one range to another. That is, a **value** of **fromLow** would get mapped to **toLow**, a value of **fromHigh** to **toHigh**, values in-between to values in-between, etc.

Figure 25: map function documentation



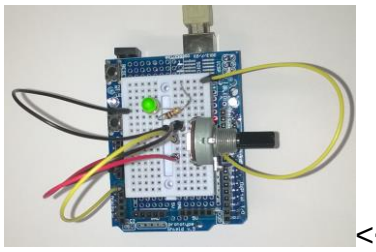

<<>>

Figure 26: LED PWM control

7. Assignments

7.1. Introduction

These assignments are done in couples (two students). Each couple must make all assignments, but helping other couples is allowed. For the last assignment the couple must program the two Arduino's with different sketches, one as sensor/sender and the other one as receiver/actuator.

7.2. Items you will need

You must have:

- PC or laptop with a free USB port
- The wires and breadboard from the previous week

You must loan from the TI-lab:

- Arduino Uno, USB cable
- IR receiver module
- IR transmitter LED, 1k resistor
- potentiometer 1kΩ
- MPU-6050 acceleration sensor
- Small servo motor
- NEC-compatible IR remote control (for testing the IR receiver)

Due to the limited number of sets available, you aren't allowed to take these sets home. Use them in the TI lab, or in the adjacent room.

7.3. Preparation: blink the on-board LED, and say "Hello"

1. Install the Arduino IDE from https://www.arduino.cc/en/main/software.
2. Download the example sketches from http://www.github.com/wovo/vdein-examples.
3. Connect the Arduino to your PC.
4. Double-click on the blink1 example (the blink1.ino file in the blink1 directory) to start the Arduino IDE
5. Under Tools➔Board select "Arduino/Genuino Uno"
6. Under Tools➔Port select the com port created by Windows for the Arduino
7. Compile and download the blink1 (Sketch➔Upload, or ctrl-U)
8. Check that the LED on your Arduino Uno blinks.
9. Open the hello sketch (File➔Open hello.uno in the hello directory)
10. Compile and upload the sketch
11. Open the serial monitor (Tools➔Serial Monitor)
12. Check that you see lines "Hello world" printed in the monitor window.

7.4. Rewrite to FSM and combine

Get the switch1 example, buikld the circuit, and verify that it works. Rewrite the sketch in the FSM-object style. When this works, combine it with the blinker FSM to a sketch that combines both functions (blinking and switch-controls-LED).

7.5. Connect an IR receiver

Install the IR library from https://github.com/z3t0/Arduino-IRremote conform the instructions on that page. Connect the IR receiver. There are two types of IR

receivers, check which one you have and connect it accordingly. Vs is the 5V connection. Check the sketch to find the Arduino pin that you must connect the IR receivers's OUT pin to.

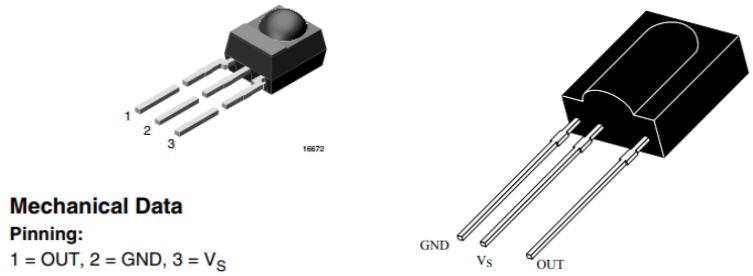


Figure 27: pinout of two types of IR receivers

Use the IR remote and the ir-recieve example to verify that your setup works. The sketch translates the received IR code for the keys 0..9 (which seem to be totally random numbers) to the values 0..9. The function use() receives that value.

7.6. Connect a servo motor

Connect

- A servo motor to ground (brown), 5V (red) and digital pin 9 (yellow)
- A potentiometer to ground, 5V, and (its middle pin) to analog pin 0.

Put one of the plastic arms on the shaft of the servo to make the shaft movements visible. Run the example sketch File➔Examples➔Servo➔Knob to verify that your setup works. The servo motor can cause problems while downloading a sketch because it draws large and varying amounts of current. If you get a downloading error disconnect the servo (just the 5V wire) while downloading and re-connect it afterwards.

7.7. Servo control using IR remote

Connect the servo and the IR receiver. Combine the servo and IR sketches. The combined sketch must set the servo to one of 10 positions as commanded by the IR remote. Keep the logging to the PC in place.

7.8. MPU6050

The MPU6050 is a gravitation/acceleration sensor: it measures both the (stationary) pull of gravity (or another force) in 3 directions (X, Y and Z axis), and the momentaneous change in those forces (acceleration). For this assignment we will be using the acceleration (gravity) in one direction.

Figure 28: an MPU6050 acceleration sensor module

Install the MPU6050 library from http://diyhacking.com/projects/MPU6050.zip and the I2C library from http://diyhacking.com/projects/I2Cdev.zip Connect the MPU-6050 module.

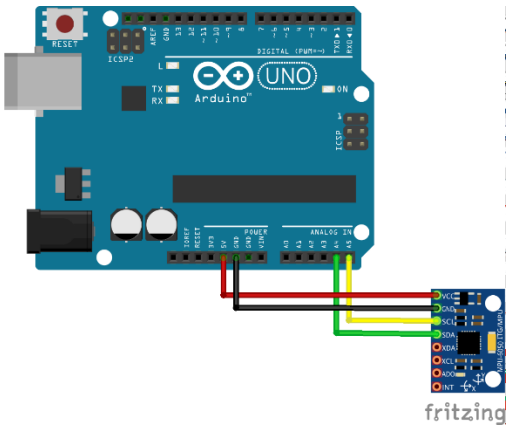


Figure 29: connecting an MPU6050 gravitation/acceleration sensor, using only VCC, GND, DCL and SDA

Run the mpu6050 ➔ mpu-6050-raw example to verify that the setup works. (You will have to change the baudrate to 9600.) In the Serial Monitor you can see the output: six columns of numbers. The first three columns are acceleration in X, Y and Z direction. Choose one axis to turn the sensor. Note which of the 3 values varies (most), and what the range is.

7.9. Servo control using MPU6050

Combine the MPU-6050 and servo sketches. The turning of the MPU must control the servo. Use the range you found in the previous assignment for the translation of the MPU6050 output to the 0 .. 180 input for the servo.

7.10. MPU6050 remote control of the servo via IR

For this assignment two Arduino's are needed, the second one must run the IR receiver and servo motor sketch from a previous assignment. You'll have to ask another couple to run this on their Arduino, or ask the teacher to run it so you can demonstrate your sketch and setup.

Your Arduino must be programmed to read the MPU-6050, and send its orientation as an IR command. The second Arduino receives this signal and orients the servo motor's shaft accordingly. The map() function will again be handy to convert the MPU6050 reading to a value (0..9) that can be transmitted. An example project (github) shows how IR transmission is done.