FREE eBook

LEARNING
arduino

Free unaffiliated eBook created from
Stack Overflow contributors.

#arduino

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: arduino

It is an unofficial and free arduino ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official arduino.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with arduino

Remarks

What is Arduino?

Arduino is an open-source electronics platform based on easy-to-use hardware and software.

Why Use Arduino?

- Inexpensive. You can also buy clones that are even cheaper.
- Easy to use and get started with
- Huge community
- Completely Open Source

Versions

Version	Release Date
1.0.0	2016-05-08

Examples

Bare Minimum

Here's the 'bare minimum' Arduino sketch. This can be loaded into the Arduino IDE by choosing `File > Examples > 01. Basics > Bare Minimum`.

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

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

Code in the `setup()` function will be run once when the program starts. This is useful to set up I/O pins, initialize variables, etc. Code in the `loop()` function will be run repeatedly until the Arduino is switched off or a new program is uploaded. Effectively, the code above looks like this inside the Arduino runtime library:

```
setup();
while(1) {
```

```
  loop();
}
```

Unlike programs running on your computer, Arduino code can never quit. This is because the microcontroller only has one program loaded into it. If this program quit there would be nothing to tell the microcontroller what to do.

Blink

Here's a short example that demonstrates the `setup()` and `loop()` functions. This can be loaded into the Arduino IDE by choosing `File > Examples > 01. Basics > Blink`. (*Note:* Most Arduino boards have an LED already connected to pin 13, but you may need to add an external LED to see the effects of this sketch.)

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

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

The above snippet:

1. Defines the `setup()` function. The `setup()` function gets called first on execution in every Arduino program.

    1. Sets pin 13 as an output.

        Without this, it might be set to an input, which would make the LED not work; however once it is set as an output it will stay that way so this only needs to be done once when the program starts.

2. Defines the `loop()` function. The `loop()` function is called repeatedly for as long as the program is running.

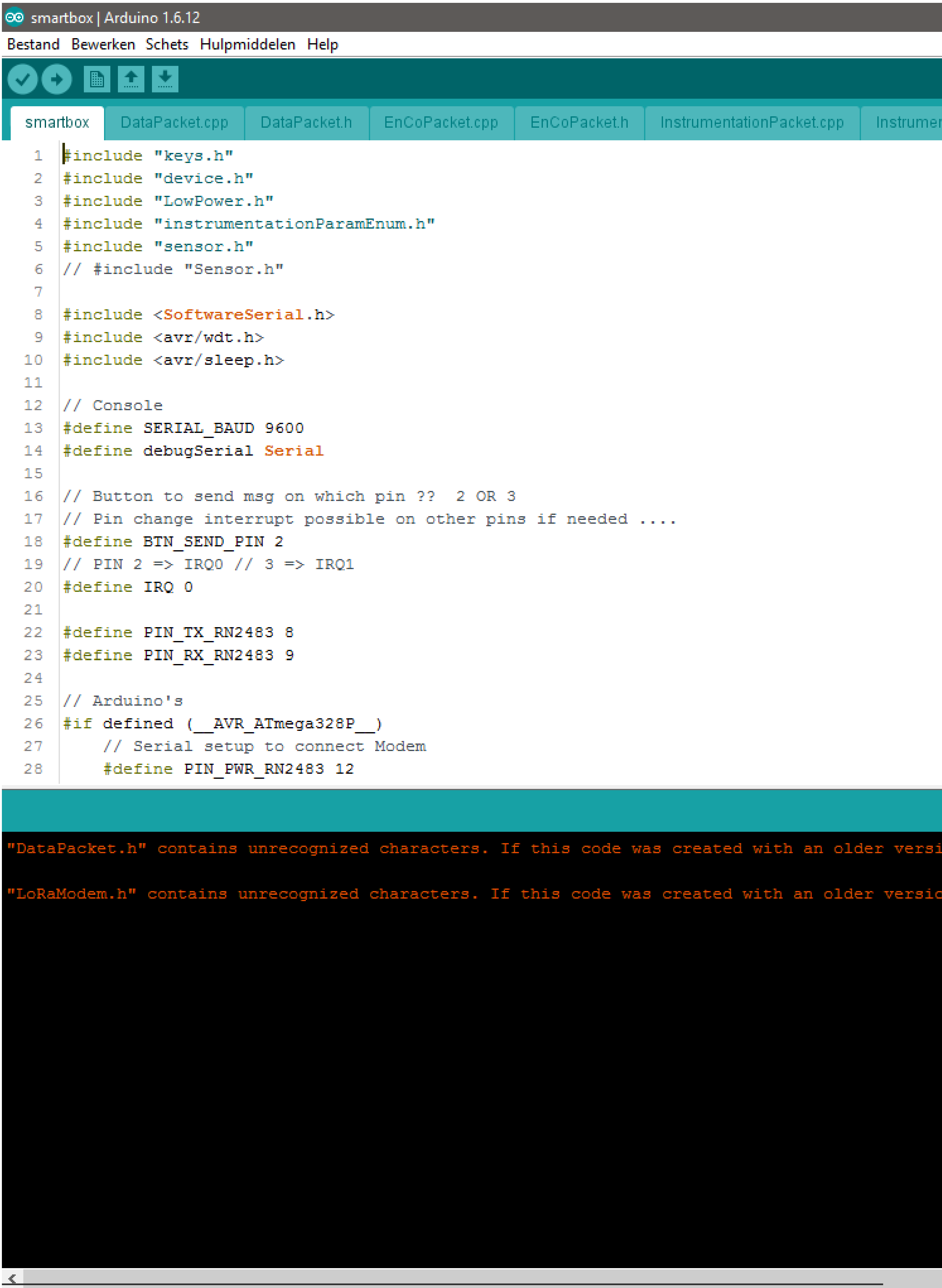
    1. `digitalWrite(13, HIGH);` turns the LED on.
    2. `delay(1000);` waits one second (1000 milliseconds).
    3. `digitalWrite(13, LOW);` turns the LED off.
    4. `delay(1000);` waits one second (1000 milliseconds).

Because `loop()` is run repeatedly for as long as the program is running, the LED will flash on and off with a period of 2 seconds (1 second on, 1 second off). This example is based off of the Arduino Uno and any other board that already has an LED connected to Pin 13. If the board that is being used does not have an on-board LED connected to that pin, one can be attached externally.

More on timing (for example delays and measuring time): Time Management

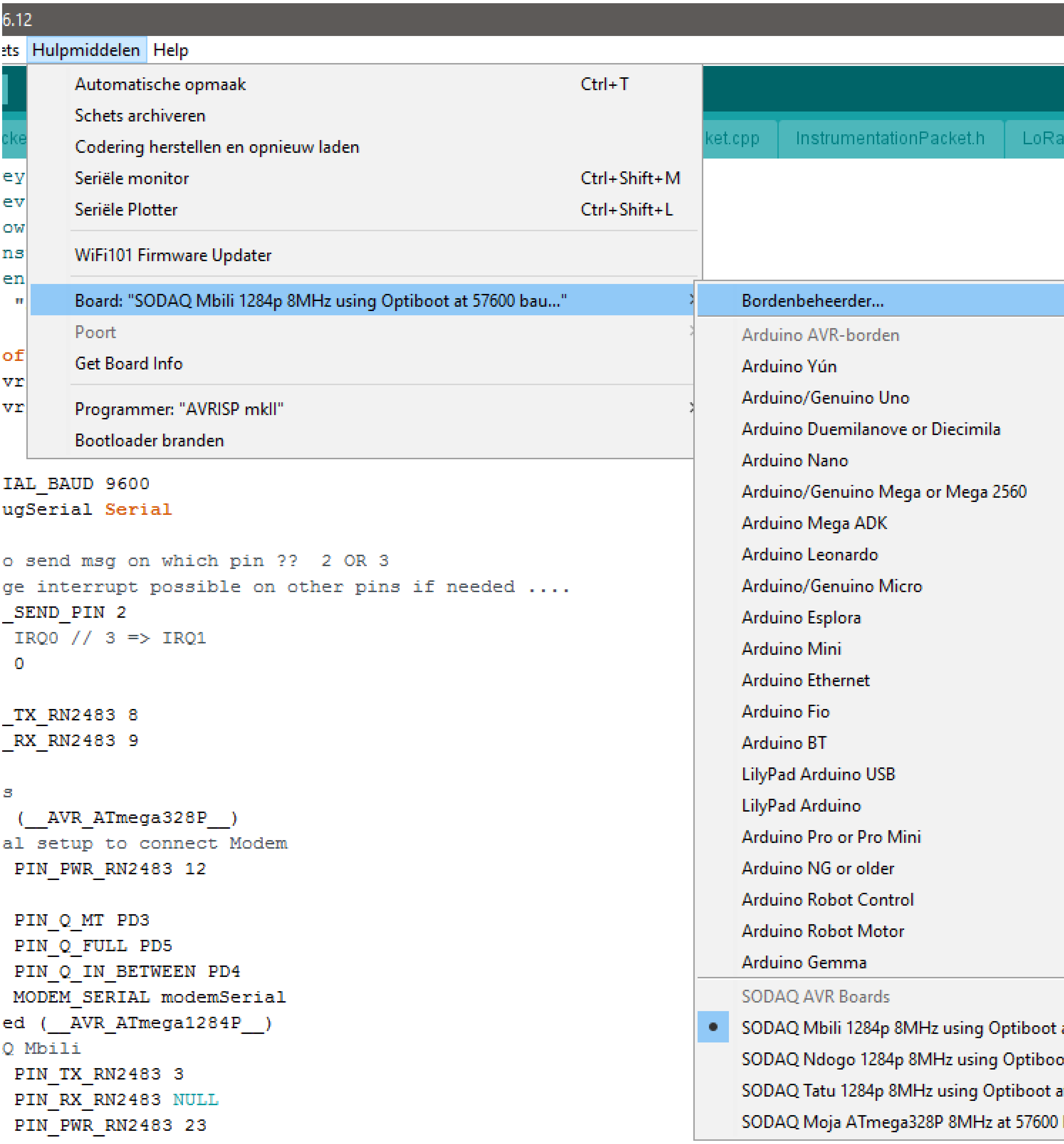
First Time Setup

Software needed: Arduino IDE


smartbox | Arduino 1.6.12
Bestand Bewerken Schets Hulpmiddelen Help
smartbox
DataPacket.cpp
DataPacket.h
EnCoPacket.cpp
EnCoPacket.h
InstrumentationPacket.cpp
#include "keys.h"
#include "device.h"
#include "LowPower.h"
#include "instrumentationParamEnum.h"
#include "sensor.h"
// #include "Sensor.h"
#include <SoftwareSerial.h>
#include <avr/wdt.h>
#include <avr/sleep.h>
// Console
#define SERIAL_BAUD 9600
#define debugSerial Serial
// Button to send msg on which pin ?? 2 OR 3
// Pin change interrupt possible on other pins if needed ....
#define BTN_SEND_PIN 2
// PIN 2 => IRQ0 // 3 => IRQ1
#define IRQ 0
#define PIN_TX_RN2483 8
#define PIN_RX_RN2483 9
// Arduino's
#if defined (__AVR_ATmega328P__)
// Serial setup to connect Modem
#define PIN_PWR_RN2483 12
"DataPacket.h" contains unrecognized characters. If this code was created with an older versi
"LoRaModem.h" contains unrecognized characters. If this code was created with an older versio
1

functions. This is enough to upload to an Arduino board, but it will do nothing at all. The "Blink" example sketch works as a simple test when first using an Arduino board. Go to File → Examples → 01.Basics → Blink. This will open a new window with the Blink sketch.

Select your board. Go to Tools → Board → [name of your Arduino board].

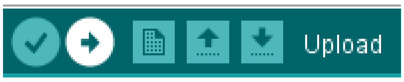


Select the COM port for your board. Most Aurduino-compatible boards will create a fake COM port, which is used for serial communication (debugging) and for programming the board. COM 1 is *usually* already present, and your board will create a new one, e.g. COM 4. Select this from Tools → Port → COM 4 (or other COM number).

Some boards have additional settings in the Tools menu, such as clock speed. These vary from board to board, but usually an acceptable set of defaults is already selected.

Upload

You are now ready to upload Blink. Click the Upload button or select Sketch → Upload. The sketch will compile, then upload to your Arduino board. If everything worked, the on-board LED will start blinking on and off every second.



Serial monitor

In the Arduino IDE ypu hava a serial monitor. To open it use the button *serial monitor* at the right side of the window.

Be sure that the code is uploaded before you open the monitor. The upload and monitor will not run at the same time!

LED - With Button control

You can also use this code to setup an LED with a button switch with a pull up resistor, this could preferably be with the next step after setting up the intial LED controller

```
int buttonState = 0; // variable for reading the pushbutton status

void setup()
{
    // initialize the LED pin as an output:
    pinMode(13, OUTPUT); // You can set it just using its number
    // initialize the pushbutton pin as an input:
    pinMode(2, INPUT);
}

void loop()
{
    // read the state of the pushbutton value:
    buttonState = DigitalRead(2);

    // check if the pushbutton is pressed.
    // If it's not, the buttonState is HIGH : if (buttonState == HIGH)
    {
        // turn LED off:
```

```
        digitalWrite(13, LOW);
    }
    else
    {
        // turn LED off:
        digitalWrite(13, HIGH);
    }
}
```

Read Getting started with arduino online: https://riptutorial.com/arduino/topic/610/getting-started-with-arduino

Chapter 2: Analog Inputs

Syntax

- `analogRead(pin)` //Read from the given pin.

Remarks

```
Serial.println(val)
```

For help with Serial communication, see: Serial Communication

Examples

Print out an Analog Value

```
int val = 0;    // variable used to store the value
                // coming from the sensor

void setup() {
  Serial.begin(9600); //Begin serializer to print out value

  // Note: Analogue pins are
  // automatically set as inputs
}

void loop() {

  val = analogRead(0); // read the value from
                       // the sensor connected to A0.

  Serial.println(val); //Prints the value coming in from the analog sensor

  delay(10); // stop the program for
             // some time
}
```

Get Voltage From Analog Pin

Analog pins can be used to read voltages which is useful for battery monitoring or interfacing with analog devices. By default the AREF pin will be the same as the operating voltage of the arduino, but can be set to other values externally. If the voltage to read is larger than the input voltage, a potential devider will be needed to lower the analog voltage.

```
#define analogPin 14    //A0 (uno)
#define AREFValue 5        //Standard for 5V Arduinos
#define ADCResolution 1023    //Standard for a 10bit ADC

int ADCValue = 0;
```

```
float voltage = 0;

void setup()
{
    Serial.begin(9600);
}

void loop()
{
    readADC();
    Serial.print(voltage); Serial.println("V");
}

void readADC()
{
    ADCValue = analogRead(analogPin);
    float = ( ( (float)ADCValue/ADCRange ) * AREFValue );    //Convert the ADC value to a
float, devide by the ADC resolution and multiply by the AREF voltage
}
```

Read Analog Inputs online: https://riptutorial.com/arduino/topic/2382/analog-inputs

Chapter 3: Arduino IDE

Examples

Installing on Windows

1. Go to https://www.arduino.cc/en/Main/Software
2. Click the "Windows Installer" link
3. Follow the instructions

Portable app on Windows

To use the Arduino IDE on Windows without needing to install it:

1. Go to https://www.arduino.cc/en/Main/Software
2. Click the "Windows ZIP file for non admin install" link
3. Extract the archive to a folder
4. Open the folder, and double click `Arduino.exe`

Installing on Fedora

1. Open a terminal and run: `sudo dnf install arduino`
2. Open the Arduino application, or type `arduino` into the terminal

Installing on Ubuntu

1. Open a terminal and run: `sudo apt-get install arduino`
2. Open the Arduino application, or type `arduino` into the terminal

Installing on macOS

1. Go to https://www.arduino.cc/en/Main/Software
2. Click the `Mac OS X` link.
3. Unzip the `.zip`file.
4. Move the `Arduino` application to `Applications`.

Read Arduino IDE online: https://riptutorial.com/arduino/topic/3790/arduino-ide

Chapter 4: Audio Output

Parameters

Parameter	Details
speaker	Should be an output to an analog speaker

Examples

Basic Note Outputs

```
#define NOTE_C4  262 //From pitches.h file defined in [Arduino Tone Tutorial][1]

int Key = 2;
int KeyVal = 0;

byte speaker = 12;

void setup()
{
  pinMode(Key, INPUT);  //Declare our key (button) as input
  pinMode(speaker, OUTPUT);
}

void loop()
{
  KeyVal = digitalRead(Key);
  if (KeyVal == HIGH) {
    tone(speaker, NOTE_C4); //Sends middle C tone out through analog speaker
  } else {
    noTone(speaker); //Ceases tone emitting from analog speaker
  }

  delay(100);
}
```

[1]: https://www.arduino.cc/en/Tutorial/toneMelody

Read Audio Output online: https://riptutorial.com/arduino/topic/2384/audio-output

Chapter 5: Bluetooth Communication

Parameters

method	details
SoftwareSerial.h	Documentation
SoftwareSerial(rxPin, txPin, inverse_logic)	Constructor. **rxPin**: Data in (receive) pin, defaults to 0. **txPin**: Data out (transmit) pin, defaults to 1. **inverse_logic**: If true, treats LOW as if it were HIGH and HIGH as LOW when determining bit values. defaults to false.
begin(speed)	Sets the baud rate for serial communication. Supported baud rates are 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 31250, 38400, 57600, and 115200.
available()	Check if there is some data over serial
read()	Reads a string from serial
isListening()	Checks to see if requested software serial port is actively listening.
overflow()	Checks if a software serial buffer overflow has occurred. Calling this function clears the overflow flag, meaning that subsequent calls will return false unless another byte of data has been received and discarded in the meantime. The software serial buffer can hold 64 bytes.
peek()	Return a character that was received on the RX pin of the software serial port. Unlike read(), however, subsequent calls to this function will return the same character. Note that only one SoftwareSerial instance can receive incoming data at a time (select which one with the `listen()` function).
print(data)	Prints data to the transmit pin of the software serial port. Works the same as the `Serial.print()` function.
println(data)	Prints data to the transmit pin of the software serial port, followed by a carriage return and line feed. Works the same as the `Serial.println()` function.
listen()	Enables the selected software serial port to listen. Only one software serial port can listen at a time; data that arrives for other ports will be discarded. Any data already received is discarded during the call to `listen()` (unless the given instance is already

method	details
	listening).
write(data)	Prints data to the transmit pin of the software serial port as raw bytes. Works the same as the `Serial.write()` function.

Remarks

Common Mistake : If you keep the rx and tx pins at default values (0 and 1), you cannot upload new code until and unless you remove it, so it's almost always better to change the tx and rx pins in the SoftwareSerial constructor.

Examples

Basic bluetooth hello world

```
#include <SoftwareSerial.h>
// its always better to change the default tx and rx as the may interfere with other process
in future.

// configure tx , rx by defualt they will be 0 and 1 in arduino UNO
SoftwareSerial blue(3,2);
void setup() {
  // preferred baud rate/data transfer rate in general is 38400
     blue.begin(38400);
  // do initialization or put one time executing code here
}

void loop() {

  // put code that you want it to run every time no matter what
    if(blue.available()){
        // put only that code which needsd to run when there is some data
        // This means that the their is some data sent over the bluetooth
        // You can do something with the data

        int n;
        // consider that the data received to be integer, read it by using blue.parseInt();

        n = blue.parseInt();

    }

}
```

Read Bluetooth Communication online: https://riptutorial.com/arduino/topic/2543/bluetooth-communication

Chapter 6: Data Storage

Examples

cardInfo

```
/*
 SD card test

 This example shows how use the utility libraries on which the'
 SD library is based in order to get info about your SD card.
 Very useful for testing a card when you're not sure whether its working or not.

 The circuit:
  * SD card attached to SPI bus as follows:
 ** MOSI - pin 11 on Arduino Uno/Duemilanove/Diecimila
 ** MISO - pin 12 on Arduino Uno/Duemilanove/Diecimila
 ** CLK -  pin 13 on Arduino Uno/Duemilanove/Diecimila
 ** CS -   depends on your SD card shield or module.
           Pin 4 used here for consistency with other Arduino examples

 created  28 Mar 2011
 by Limor Fried
 modified 9 Apr 2012
 by Tom Igoe
 */

// include the SD library:
#include <SPI.h>
#include <SD.h>

// set up variables using the SD utility library functions:
Sd2Card card;
SdVolume volume;
SdFile root;

// change this to match your SD shield or module;
// Arduino Ethernet shield: pin 4
// Adafruit SD shields and modules: pin 10
// Sparkfun SD shield: pin 8
const int chipSelect = 4;

void setup()
{
  // Open serial communications and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for Leonardo only
  }

  Serial.print("\nInitializing SD card...");

  // we'll use the initialization code from the utility libraries
  // since we're just testing if the card is working!
  if (!card.init(SPI_HALF_SPEED, chipSelect)) {
    Serial.println("initialization failed. Things to check:");
    Serial.println("* is a card inserted?");
```

```
    Serial.println("* is your wiring correct?");
    Serial.println("* did you change the chipSelect pin to match your shield or module?");
    return;
  } else {
    Serial.println("Wiring is correct and a card is present.");
  }

  // print the type of card
  Serial.print("\nCard type: ");
  switch (card.type()) {
    case SD_CARD_TYPE_SD1:
      Serial.println("SD1");
      break;
    case SD_CARD_TYPE_SD2:
      Serial.println("SD2");
      break;
    case SD_CARD_TYPE_SDHC:
      Serial.println("SDHC");
      break;
    default:
      Serial.println("Unknown");
  }

  // Now we will try to open the 'volume'/'partition' - it should be FAT16 or FAT32
  if (!volume.init(card)) {
    Serial.println("Could not find FAT16/FAT32 partition.\nMake sure you've formatted the
card");
    return;
  }

  // print the type and size of the first FAT-type volume
  uint32_t volumesize;
  Serial.print("\nVolume type is FAT");
  Serial.println(volume.fatType(), DEC);
  Serial.println();

  volumesize = volume.blocksPerCluster();    // clusters are collections of blocks
  volumesize *= volume.clusterCount();       // we'll have a lot of clusters
  volumesize *= 512;                            // SD card blocks are always 512 bytes
  Serial.print("Volume size (bytes): ");
  Serial.println(volumesize);
  Serial.print("Volume size (Kbytes): ");
  volumesize /= 1024;
  Serial.println(volumesize);
  Serial.print("Volume size (Mbytes): ");
  volumesize /= 1024;
  Serial.println(volumesize);


  Serial.println("\nFiles found on the card (name, date and size in bytes): ");
  root.openRoot(volume);

  // list all files in the card with date and size
  root.ls(LS_R | LS_DATE | LS_SIZE);
}

void loop(void) {

}
```

SD card datalogger

```
/*
  SD card datalogger

 This example shows how to log data from three analog sensors
 to an SD card using the SD library.

 The circuit:
 * analog sensors on analog ins 0, 1, and 2
 * SD card attached to SPI bus as follows:
 ** MOSI - pin 11
 ** MISO - pin 12
 ** CLK - pin 13
 ** CS - pin 4

 created  24 Nov 2010
 modified 9 Apr 2012
 by Tom Igoe

 This example code is in the public domain.

 */

#include <SPI.h>
#include <SD.h>

const int chipSelect = 4;

void setup()
{
  // Open serial communications and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for Leonardo only
  }

  Serial.print("Initializing SD card...");

  // see if the card is present and can be initialized:
  if (!SD.begin(chipSelect)) {
    Serial.println("Card failed, or not present");
    // don't do anything more:
    return;
  }
  Serial.println("card initialized.");
}

void loop()
{
  // make a string for assembling the data to log:
  String dataString = "";

  // read three sensors and append to the string:
  for (int analogPin = 0; analogPin < 3; analogPin++) {
    int sensor = analogRead(analogPin);
    dataString += String(sensor);
    if (analogPin < 2) {
      dataString += ",";
    }
```

```
  }

  // open the file. note that only one file can be open at a time,
  // so you have to close this one before opening another.
  File dataFile = SD.open("datalog.txt", FILE_WRITE);

  // if the file is available, write to it:
  if (dataFile) {
    dataFile.println(dataString);
    dataFile.close();
    // print to the serial port too:
    Serial.println(dataString);
  }
  // if the file isn't open, pop up an error:
  else {
    Serial.println("error opening datalog.txt");
  }
}
```

SD card file dump

```
/*
  SD card file dump

 This example shows how to read a file from the SD card using the
 SD library and send it over the serial port.

 The circuit:
 * SD card attached to SPI bus as follows:
 ** MOSI - pin 11
 ** MISO - pin 12
 ** CLK - pin 13
 ** CS - pin 4

 created  22 December 2010
 by Limor Fried
 modified 9 Apr 2012
 by Tom Igoe

 This example code is in the public domain.

 */

#include <SPI.h>
#include <SD.h>

const int chipSelect = 4;

void setup()
{
  // Open serial communications and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for Leonardo only
  }


  Serial.print("Initializing SD card...");
```

```
  // see if the card is present and can be initialized:
  if (!SD.begin(chipSelect)) {
    Serial.println("Card failed, or not present");
    // don't do anything more:
    return;
  }
  Serial.println("card initialized.");

  // open the file. note that only one file can be open at a time,
  // so you have to close this one before opening another.
  File dataFile = SD.open("datalog.txt");

  // if the file is available, write to it:
  if (dataFile) {
    while (dataFile.available()) {
      Serial.write(dataFile.read());
    }
    dataFile.close();
  }
  // if the file isn't open, pop up an error:
  else {
    Serial.println("error opening datalog.txt");
  }
}

void loop()
{

}
```

SD card basic file example

```
/*
  SD card basic file example

 This example shows how to create and destroy an SD card file
 The circuit:
 * SD card attached to SPI bus as follows:
 ** MOSI - pin 11
 ** MISO - pin 12
 ** CLK - pin 13
 ** CS - pin 4

 created   Nov 2010
 by David A. Mellis
 modified 9 Apr 2012
 by Tom Igoe

 This example code is in the public domain.

 */
#include <SPI.h>
#include <SD.h>

File myFile;

void setup()
{
  // Open serial communications and wait for port to open:
```

```
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for Leonardo only
  }

  Serial.print("Initializing SD card...");

  if (!SD.begin(4)) {
    Serial.println("initialization failed!");
    return;
  }
  Serial.println("initialization done.");

  if (SD.exists("example.txt")) {
    Serial.println("example.txt exists.");
  }
  else {
    Serial.println("example.txt doesn't exist.");
  }

  // open a new file and immediately close it:
  Serial.println("Creating example.txt...");
  myFile = SD.open("example.txt", FILE_WRITE);
  myFile.close();

  // Check to see if the file exists:
  if (SD.exists("example.txt")) {
    Serial.println("example.txt exists.");
  }
  else {
    Serial.println("example.txt doesn't exist.");
  }

  // delete the file:
  Serial.println("Removing example.txt...");
  SD.remove("example.txt");

  if (SD.exists("example.txt")) {
    Serial.println("example.txt exists.");
  }
  else {
    Serial.println("example.txt doesn't exist.");
  }
}

void loop()
{
  // nothing happens after setup finishes.
}
```

Listfiles

```
/*
 Listfiles

 This example shows how print out the files in a
 directory on a SD card

 The circuit:
```

```
 * SD card attached to SPI bus as follows:
 ** MOSI - pin 11
 ** MISO - pin 12
 ** CLK - pin 13
 ** CS - pin 4

 created   Nov 2010
 by David A. Mellis
 modified 9 Apr 2012
 by Tom Igoe
 modified 2 Feb 2014
 by Scott Fitzgerald

 This example code is in the public domain.
*/

#include <SPI.h>
#include <SD.h>

File root;

void setup()
{
  // Open serial communications and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for Leonardo only
  }

  Serial.print("Initializing SD card...");

  if (!SD.begin(4)) {
    Serial.println("initialization failed!");
    return;
  }
  Serial.println("initialization done.");

  root = SD.open("/");

  printDirectory(root, 0);

  Serial.println("done!");
}

void loop()
{
  // nothing happens after setup finishes.
}

void printDirectory(File dir, int numTabs) {
   while(true) {

     File entry =  dir.openNextFile();
     if (! entry) {
       // no more files
       break;
     }
     for (uint8_t i=0; i<numTabs; i++) {
       Serial.print('\t');
     }
     Serial.print(entry.name());
```

```
      if (entry.isDirectory()) {
        Serial.println("/");
        printDirectory(entry, numTabs+1);
      } else {
        // files have sizes, directories do not
        Serial.print("\t\t");
        Serial.println(entry.size(), DEC);
      }
      entry.close();
    }
}
```

SD card read/write

```
/*
  SD card read/write

 This example shows how to read and write data to and from an SD card file
 The circuit:
 * SD card attached to SPI bus as follows:
 ** MOSI - pin 11
 ** MISO - pin 12
 ** CLK - pin 13
 ** CS - pin 4

 created   Nov 2010
 by David A. Mellis
 modified 9 Apr 2012
 by Tom Igoe

 This example code is in the public domain.

 */

#include <SPI.h>
#include <SD.h>

File myFile;

void setup()
{
  // Open serial communications and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for Leonardo only
  }


  Serial.print("Initializing SD card...");

  if (!SD.begin(4)) {
    Serial.println("initialization failed!");
    return;
  }
  Serial.println("initialization done.");

  // open the file. note that only one file can be open at a time,
  // so you have to close this one before opening another.
  myFile = SD.open("test.txt", FILE_WRITE);
```

```
  // if the file opened okay, write to it:
  if (myFile) {
    Serial.print("Writing to test.txt...");
    myFile.println("testing 1, 2, 3.");
    // close the file:
    myFile.close();
    Serial.println("done.");
  } else {
    // if the file didn't open, print an error:
    Serial.println("error opening test.txt");
  }

  // re-open the file for reading:
  myFile = SD.open("test.txt");
  if (myFile) {
    Serial.println("test.txt:");

    // read from the file until there's nothing else in it:
    while (myFile.available()) {
      Serial.write(myFile.read());
    }
    // close the file:
    myFile.close();
  } else {
    // if the file didn't open, print an error:
    Serial.println("error opening test.txt");
  }
}

void loop()
{
  // nothing happens after setup
}
```

Read Data Storage online: https://riptutorial.com/arduino/topic/6584/data-storage

Chapter 7: Digital Inputs

Syntax

- `pinMode(pin, pinMode)` // Sets the pin to the mode defined.
- `digitalRead(pin);` // Reads the value from a specified digital pin,

Parameters

Paramter	Details
pinmode	Should be set to `INPUT` or `INPUT_PULLUP`

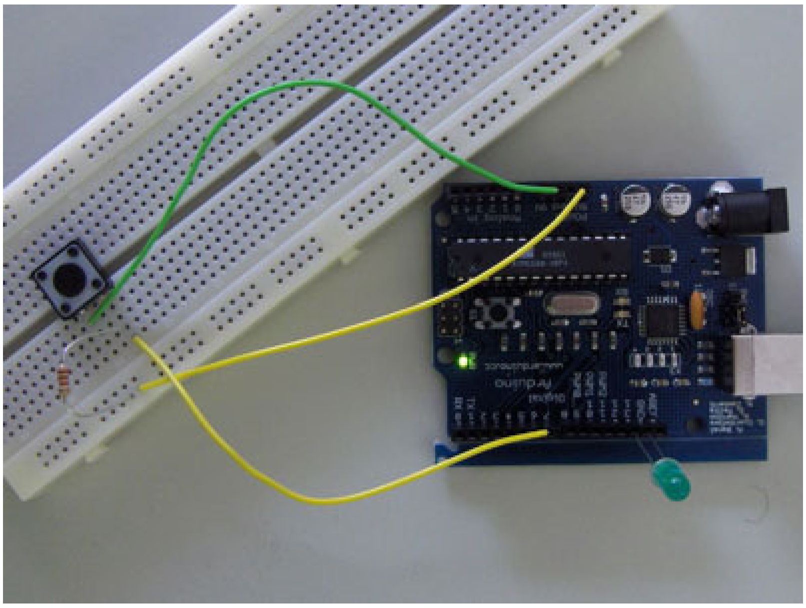
Remarks

If the input pin is not pulled LOW or HIGH, the value will float. That is, it won't be clearly a 1 or a 0, but somewhere in between. For digital input, a pullup or pulldown resistor is a necessity.

Examples

Pushbutton reading

This is an basic example on how to wire up and make an LED turn on/off when the pushbutton is pressed.



```
/* Basic Digital Read
 * ------------------
 *
 * turns on and off a light emitting diode(LED) connected to digital
```

```
 * pin 13, when pressing a pushbutton attached to pin 7. It illustrates the
 * concept of Active-Low, which consists in connecting buttons using a
 * 1K to 10K pull-up resistor.
 *
 * Created 1 December 2005
 * copyleft 2005 DojoDave <http://www.0j0.org>
 * http://arduino.berlios.de
 *
 */

int ledPin = 13; // choose the pin for the LED
int inPin = 7;   // choose the input pin (for a pushbutton)
int val = 0;     // variable for reading the pin status

void setup() {
  pinMode(ledPin, OUTPUT);  // declare LED as output
  pinMode(inPin, INPUT);    // declare pushbutton as input
}

void loop(){
  val = digitalRead(inPin);  // read input value
  if (val == HIGH) {         // check if the input is HIGH (button released)
    digitalWrite(ledPin, LOW);  // turn LED OFF
  } else {
    digitalWrite(ledPin, HIGH);  // turn LED ON
  }
}
```

Example taken from Arduino.cc.

Read Digital Inputs online: https://riptutorial.com/arduino/topic/1662/digital-inputs

Chapter 8: Digital Output

Syntax

- `digitalWrite(pin, value)`

Examples

Write to pin

```
int ledPin = 13;                  // LED connected to digital pin 13

void setup()
{
  pinMode(ledPin, OUTPUT);       // sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH);    // sets the LED on
  delay(1000);                   // waits for a second
  digitalWrite(ledPin, LOW);     // sets the LED off
  delay(1000);                   // waits for a second
}
```

Example at Arduino.cc.

Read Digital Output online: https://riptutorial.com/arduino/topic/2477/digital-output

Chapter 9: Functions

Remarks

Other than in ordinary C / C++ , the Arduino IDE allows to call a function before it is defined.

In .cpp files, you have to define the function, or at least declare the function prototype before you can use it.
In an .ino file, the Arduino IDE creates such a prototype behind the scenes.

Arduino - function declaration - official

Examples

Create simple function

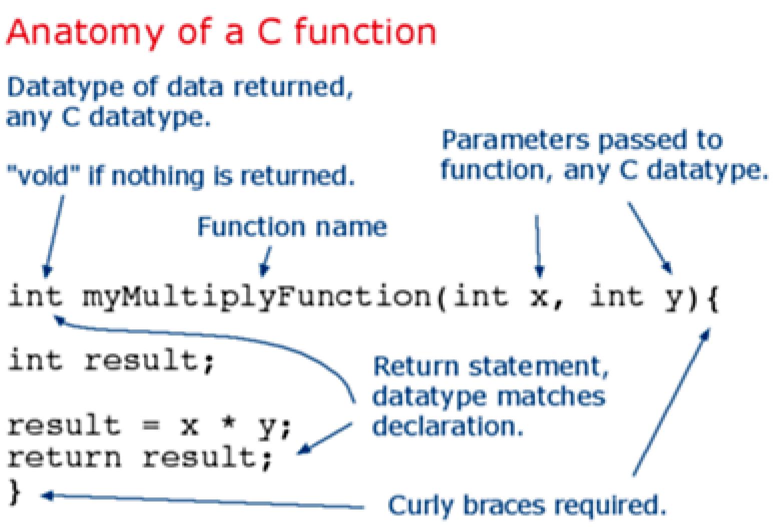
```
int squareNum (int a) {
    return a*a;
}
```

`int` : return type

`squareNum` : function name

`int a` : parameter type and name

`return a*a` : return a value (same type as the return type defined at the beginning)



Call a function

If you have a function declared you can call it anywhere else in the code. Here is an example of calling a function:

```
void setup(){
  Serial.begin(9600);
}

void loop() {
  int i = 2;

  int k = squareNum(i); // k now contains 4
  Serial.println(k);
  delay(500);
}

int squareNum(int a) {
    return a*a;
}
```

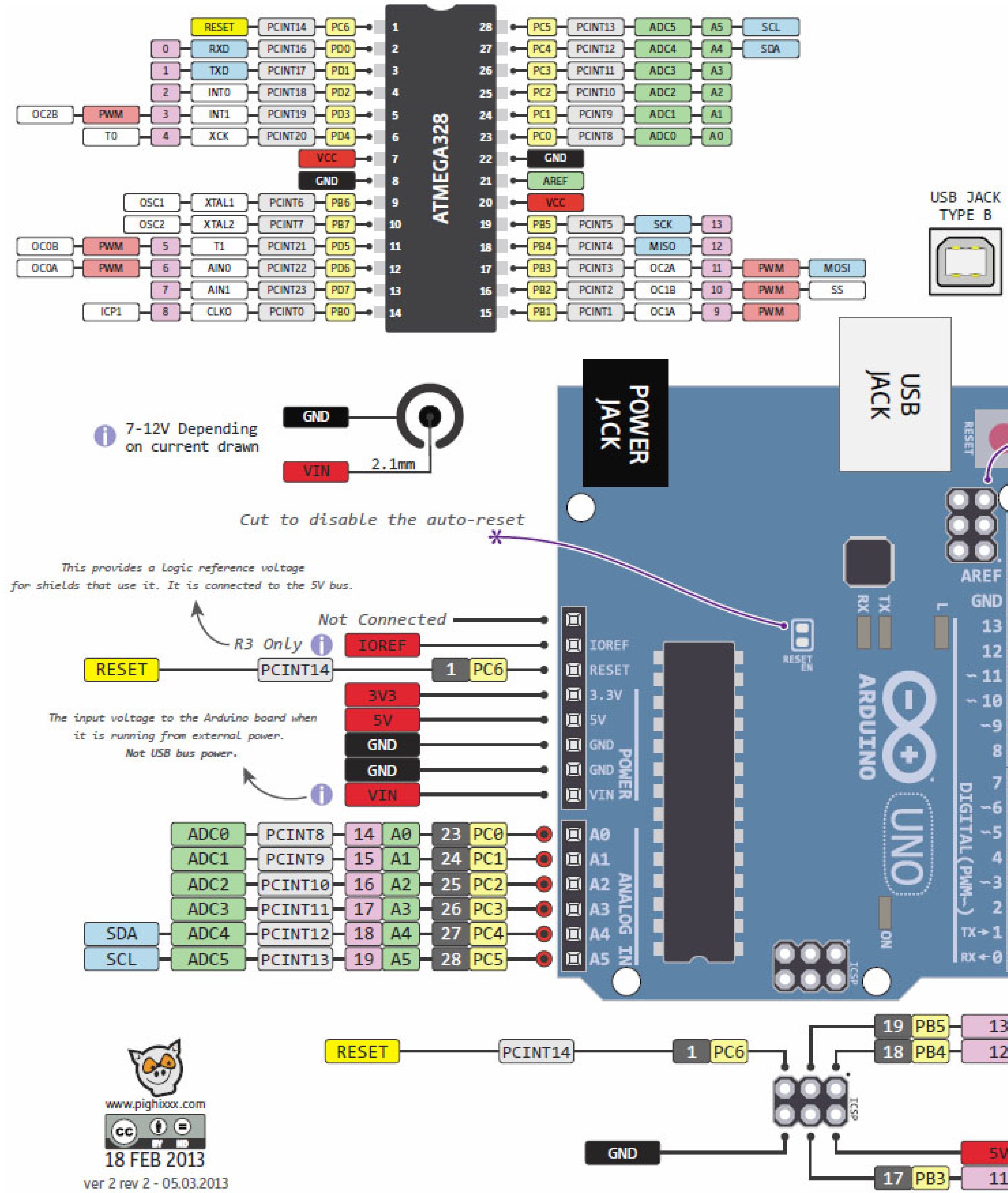
Read Functions online: https://riptutorial.com/arduino/topic/2380/functions

Chapter 10: Hardware pins

Examples

Arduino Uno R3

Microcontrollers use pins to interact with the rest of the circuit. These pins will usually be one of input / output pins, vin or ground. I/O pins can be simple digital I/O pins, or they can have some special carachteristics like being able to vary the voltage of their output using pulse width modulation. Here's a schematic of the Arduino R3 Uno and its pins.



([source](https://riptutorial.com/))

PWM Pins

PWM allows you to control the voltage of the output by switching the output between high and low very very quickly. The percentage of time the pin is high is called its 'duty cycle'.

PWM Pins: 3, 5, 6, 9, 10, 11

Analog Inputs

Just like a PWM pin can put out a range of voltages, analog pins on the Arduino Uno R3 can sense a range of oinput voltages. You might use this to read the position of a potentiometer or another input with a smoothly variable input. Please note that analog pins can't do analogWrite output - for this you need to use PWM pins.

Analog ADC Pins: A0, A1, A2, A3, A4, A5

Serial, SPI and I2C

The serial pins on the Arduino Uno R3 are also used by (for instance) the USB to Serial chip when it communicates with a computer via the on board USB port. Serial: Tx on 0, Rx on 1

SPI and I2C are communication protocols the Arduino can use to talk to shields, sensors, outputs etc...:

SPI Pins: MOSI on 11, MISO on 12, SCLK on 13, SS on 10

I2C Pins: SCL on A5, SDA on A4

On-board LED

The Arduino Uno R3 has an LED with its own resistor attached to pin 13. This means that even if you don't attach any LEDs to your board, if you set pin 13 to an output and set it high, you should see an LED on the board come on. Use the 'Blink' example sketch to locate your onboard LED.

From the Arduino Digital Pins Page

> NOTE: Digital pin 13 is harder to use as a digital input than the other digital pins because it has an LED and resistor attached to it that's soldered to the board on most boards. If you enable its internal 20k pull-up resistor, it will hang at around 1.7V instead of the expected 5V because the onboard LED and series resistor pull the voltage level down, meaning it always returns LOW. If you must use pin 13 as a digital input, set its pinMode() to INPUT and use an external pull down resistor.

On-board LED pin: 13

Read Hardware pins online: https://riptutorial.com/arduino/topic/4386/hardware-pins

Chapter 11: How Python integrates with Arduino Uno

Syntax

- `Serial.begin(baudrate) // Set baud rate (bits per second) for serial data transmission`
- `Serial.println(value) // Print data to serial port followed by Carriage Return \r and Newline character \n`
- `serial.Serial((port=None, baudrate=9600, bytesize=EIGHTBITS, parity=PARITY_NONE, stopbits=STOPBITS_ONE, timeout=None, xonxoff=False, rtscts=False, write_timeout=None, dsrdtr=False, inter_byte_timeout=None) // Initialize serial port with all parameters`
- `serial.readline() // Read serial data which contains Carriage Return \r and Newline character \n`

Parameters

Parameter	Details
serial	Python package contains classes and methods to access serial port
time	Python package includes time-related functions

Remarks

I use an Arduino Uno with Arduino IDE 1.6.9 and Python 2.7.12 running in Windows 10.

Examples

First serial communication between Arduino and Python

In this very first example, a basic serial write operation is started from an Arduino device.

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.println("Hello World!");
  delay(100);
}
```

In `setup()`, function `Serial.begin(9600)` sets up the baud rate for serial data communication. In this example, a baud rate of 9600 is used. Other values can be read here: Arduino Serial.begin() function

In `loop()`, the first message we would like to send is "Hello World!". This message is transmitted by using `Serial.println("Hello World!")` as it will send this string to serial port in ASCII format. At the end of the message, there are Carriage Return `(CR, \r)` and Newline character `(\n)`. Also, a delay of 100 milliseconds is used each time program prints to serial port.

Next, upload this Arduino sketch via COM port (remember this COM port number as it will be used in Python program).

The Python program reading serial data sent from Arduino device is shown below:

```
import serial
import time

ser = serial.Serial('COM8', 9600)
while (1):
    print ser.readline()
    time.sleep(0.1)
```

First, pyserial package should be imported. For more information about installing pyserial in Windows environment, please check this instruction: Installing Python and pyserial. Then, we initialize the serial port with COM port number and baud rate. The baud rate needs to be the same as used in Arduino sketch.

Received message will be printed in while loop using `readline()` function. A delay of 100 milliseconds is also used here as same as in Arduino sketch. Please notice that pyserial `readline()` function requires a timeout when opening a serial port (pyserial documentation: PySerial ReadLine).

Read How Python integrates with Arduino Uno online: https://riptutorial.com/arduino/topic/6722/how-python-integrates-with-arduino-uno

Chapter 12: How to store variables in EEPROM and use them for permanent storage

Syntax

- EEPROM.write(address, value); //(Store variables in EEPROM in a particular address)
- EEPROM.read(address); //(Retrieve values from EEPROM and read data stored in EEPROM)

Parameters

Parameters of EEPROM.write	Detail
address	The address where value is to be stored in EEPROM
value	Main variable to store in EEPROM. Note that this is a `uint_8` (single byte)—you must split multiple-byte data types into single bytes yourself. Or you can use `EEPROM.put` to store floats or other data types.
Parameters of EEPROM.Read	Detail
address	The address from which the variable is to be read

Remarks

The allowable addresses vary by hardware.

- ATMega328 (Uno, Pro Mini, etc.): 0–1023
- ATMega168: 0-511
- ATMega1280: 0-4095
- ATMega2560: 0-4095

source

Examples

Store a variable in EEPROM and then retrieve it and print to screen

First, add a reference to `<EEPROM.h>` at the start of your sketch:

```
#include <EEPROM.h>
```

Then your other code:

```
// Stores value in a particular address in EEPROM. There are almost 512 addresses present.

    // Store value 24 to Address 0 in EEPROM
    int addr = 0;
    int val = 24;
    EEPROM.write(addr, val);     // Writes 24 to address 0

    // ---------
    // Retrieves value from a particular address in EEPROM
    // Retrieve value from address 0 in EEPROM
    int retrievedVal = EEPROM.read(0);    // Retrieves value stored in 0 address in
                                          // EEPROM

    // *[NOTE: put Serial.begin(9600); at void setup()]*
    Serial.println(retrievedVal);        // Prints value stored in EEPROM Address 0 to
                                         // Serial (screen)
```

Read How to store variables in EEPROM and use them for permanent storage online: https://riptutorial.com/arduino/topic/5987/how-to-store-variables-in-eeprom-and-use-them-for-permanent-storage

Chapter 13: I2C Communication

Introduction

I2C is a communication protocol that can make two or more Arduino boards talk to each other. The protocol uses two pins - SDA (data line) and SCL (clock line). Those pins are different from one Arduino board type to another, so check the board specification. The I2C protocol set one Arduino board as the master, and all the others as a slave. Each slave has a different address that the programmer set hard-coded. Remark: Make sure all boards connected to the same VCC source

Examples

Multiple slaves

The following example shows how the master can receive data from multiple slaves. In this example the slave sends two short numbers. The first one is for temperature, and the second one is for moisture. Please notice that the temperature is a float (24.3). In order to use only two bytes and not four (float is four bytes), I multiple the temperature in 10, and save it as a short. So here is the master code:

```
#include <Wire.h>

#define BUFFER_SIZE  4
#define MAX_NUMBER_OF_SLAVES 24
#define FIRST_SLAVE_ADDRESS 1
#define READ_CYCLE_DELAY 1000

byte buffer[BUFFER_SIZE];

void setup()
{
  Serial.begin(9600);
  Serial.println("MASTER READER");
  Serial.println("*************");

  Wire.begin();        // Activate I2C link
}

void loop()
{
  for (int slaveAddress = FIRST_SLAVE_ADDRESS;
       slaveAddress <= MAX_NUMBER_OF_SLAVES;
       slaveAddress++)
  {
    Wire.requestFrom(slaveAddress, BUFFER_SIZE);    // request data from the slave
    if(Wire.available() == BUFFER_SIZE)
      {  // if the available data size is same as I'm expecting
        // Reads the buffer the slave sent
        for (int i = 0; i < BUFFER_SIZE; i++)
        {
```

```
          buffer[i] = Wire.read();  // gets the data
        }

        // Parse the buffer
        // In order to convert the incoming bytes info short, I use union
        union short_tag {
          byte b[2];
          short val;
        } short_cast;

        // Parse the temperature
        short_cast.b[0] = buffer[0];
        short_cast.b[1] = buffer[1];
        float temperature = ((float)(short_cast.val)) / 10;

        // Parse the moisture
        short_cast.b[0] = buffer[2];
        short_cast.b[1] = buffer[3];
        short moisture = short_cast.val;

        // Prints the income data
        Serial.print("Slave address ");
        Serial.print(slaveAddress);
        Serial.print(": Temprature = ");
        Serial.print(temprature);
        Serial.print("; Moisture = ");
        Serial.println(moisture);
      }
    }
    Serial.println("*************************");

    delay(READ_CYCLE_DELAY);
  }
}
```

And now the slave code:

```
#include <Wire.h>
#include <OneWire.h>
#include <DallasTemperature.h>

//=====================
// This is the hard-coded address. Change it from one device to another
#define SLAVE_ADDRESS 1
//=====================

// I2C Variables
#define BUFFER_SIZE 2
#define READ_CYCLE_DELAY 1000
short data[BUFFER_SIZE];

// Temprature Variables
OneWire oneWire(8);
DallasTemperature temperatureSensors(&oneWire);
float m_temperature;

// Moisture Variables
short m_moisture;

// General Variables
```

```
int m_timestamp;

void setup()
{
  Serial.begin(9600);
  Serial.println("SLAVE SENDER");
  Serial.print("Node address: ");
  Serial.println(SLAVE_ADDRESS);
  Serial.print("Buffer size: ");
  Serial.println(BUFFER_SIZE * sizeof(short));
  Serial.println("***********************");

  m_timestamp = millis();
  Wire.begin(NODE_ADDRESS);  // Activate I2C network
  Wire.onRequest(requestEvent); // Set the request event handler
  temperatureSensors.begin();
}

void loop()
{
  if(millis() - m_timestamp < READ_CYCLE_DELAY) return;

  // Reads the temperature
  temperatureSensors.requestTemperatures();
  m_temperature = temperatureSensors.getTempCByIndex(0);

  // Reads the moisture
  m_moisture = analogRead(A0);
}

void requestEvent()
{
  data[0] = m_temperature * 10; // In order to use short, I multiple by 10
  data[1] = m_moisture;
  Wire.write((byte*)data, BUFFER_SIZE * sizeof(short));
}
```

Read I2C Communication online: https://riptutorial.com/arduino/topic/9092/i2c-communication

Chapter 14: Interrupts

Syntax

- digitalPinToInterrupt(pin); // converts a pin id to an interrupt id, for use with `attachInterrupt()` and `detachInterrupt()`.
- attachInterrupt(digitalPinToInterrupt(pin), ISR, mode); // recommended
- attachInterrupt(interrupt, ISR, mode); // not recommended
- detachInterrupt(digitalPinToInterrupt(pin));
- detachInterrupt(interrupt);
- noInterrupts(); // disables interrupts
- interrupts(); // re-enable interrupts after `noInterrupts()` has been called.

Parameters

Parameter	Notes
interrupt	Id of the interrupt. Not to be mistaken for pin number.
ISR	Interrupt Service Routine. This is the method which will be executed when the interrupt occurs.
mode	What should cause the interrupt to trigger. One of LOW, CHANGE, RISING, or FALLING. Due boards also allow HIGH.

Remarks

Interrupt Service Routines (ISRs) should be as short as possible, since they pause main program execution and can thus screw up time-dependent code. Generally this means in the ISR you set a flag and exit, and in the main program loop you check the flag and do whatever that flag is supposed to do.

You cannot use `delay()` or `millis()` in an ISR because those methods themselves rely on interrupts.

Examples

Interrupt on Button Press

This example uses a push button (tact switch) attached to digital pin 2 and GND, using an internal pull-up resistor so pin 2 is HIGH when the button is not pressed.

```
const int LED_PIN = 13;
const int INTERRUPT_PIN = 2;
volatile bool ledState = LOW;

void setup() {
    pinMode(LED_PIN, OUTPUT);
    pinMode(INTERRUPT_PIN, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), myISR, FALLING); // trigger when
button pressed, but not when released.
}

void loop() {
    digitalWrite(LED_PIN, ledState);
}

void myISR() {
    ledState = !ledState;
    // note: LOW == false == 0, HIGH == true == 1, so inverting the boolean is the same as
switching between LOW and HIGH.
}
```

One gotcha with this simple example is that push buttons tend to bounce, meaning that when pressing or releasing, the circuit opens and closes more than once before it settles into the final closed or open state. This example doesn't take that into account. As a result, sometimes pressing the button will toggle the LED multiple times, instead of the expected once.

Read Interrupts online: https://riptutorial.com/arduino/topic/2913/interrupts

Chapter 15: Libraries

Introduction

Here you will find documentation on:

-Installing libraries into the Arduino IDE

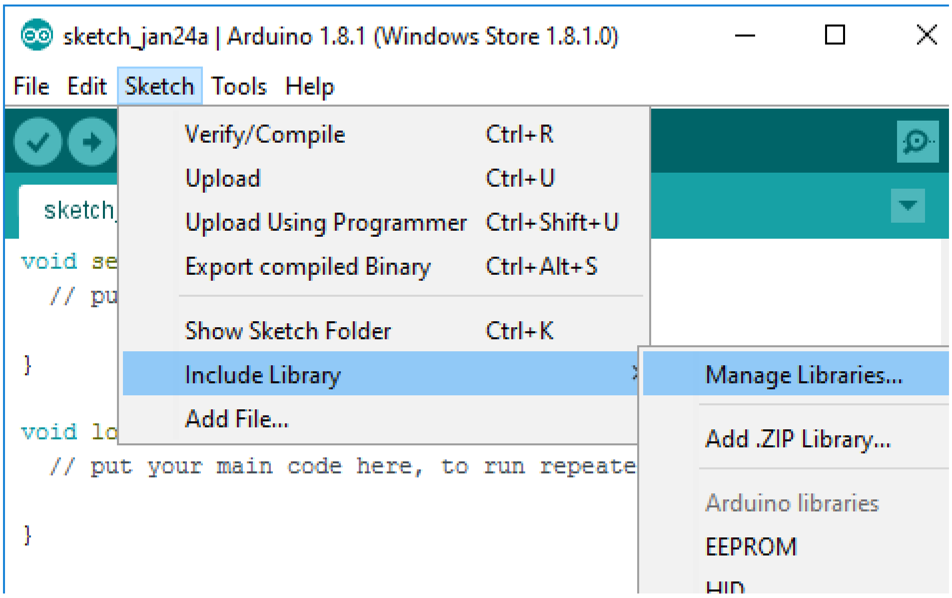
-Including libraries into a Sketch

Examples

Installing libraries with the Library Manager

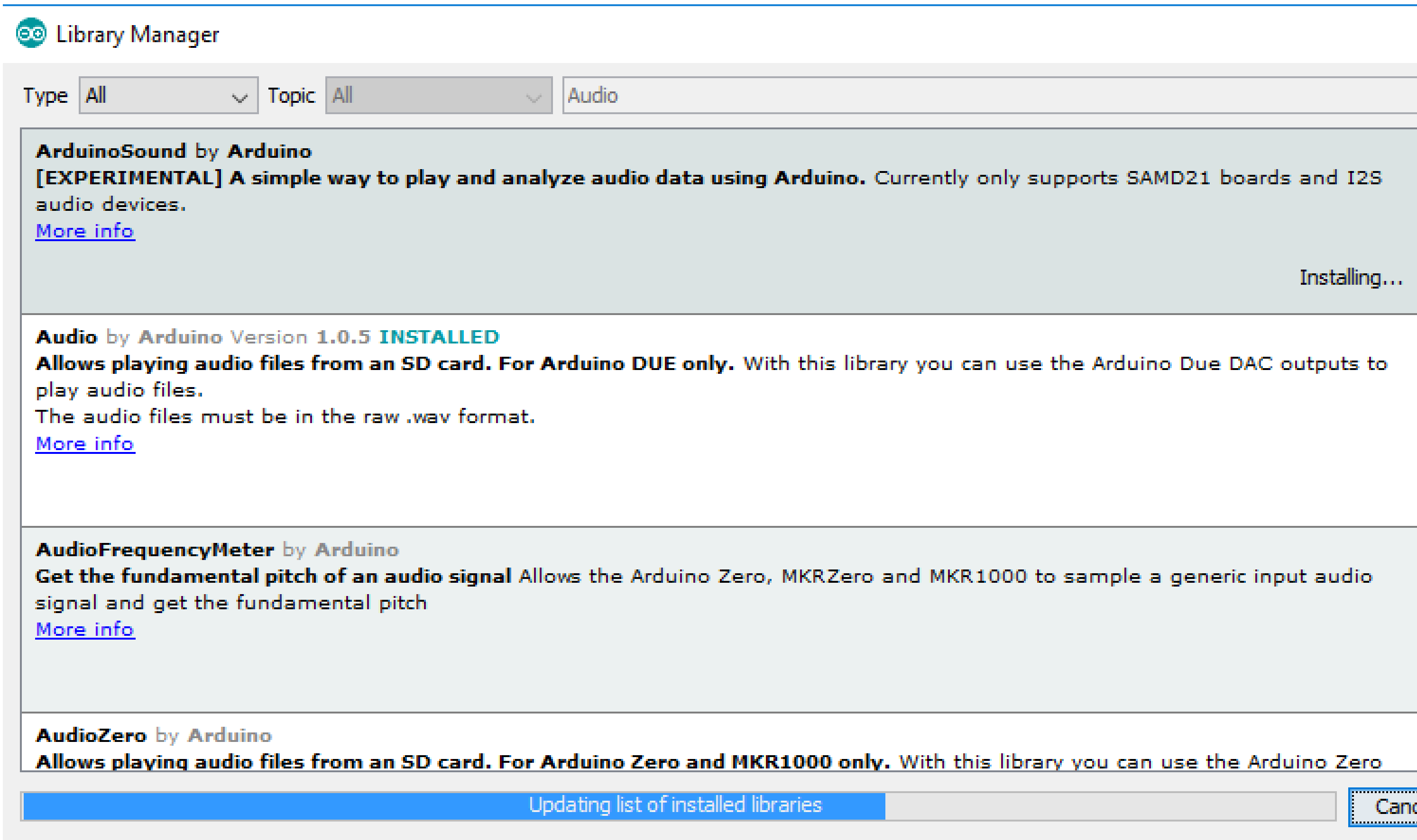
To install a new library into the Arduino IDE:

- **Open Sketch Menu > Include Library > Manage Libraries.**



Once you have opened the Library Manager you can use the menu in the top to filter the results.

- **Click on the library you want, select a version in the drop down menu, and click install.**

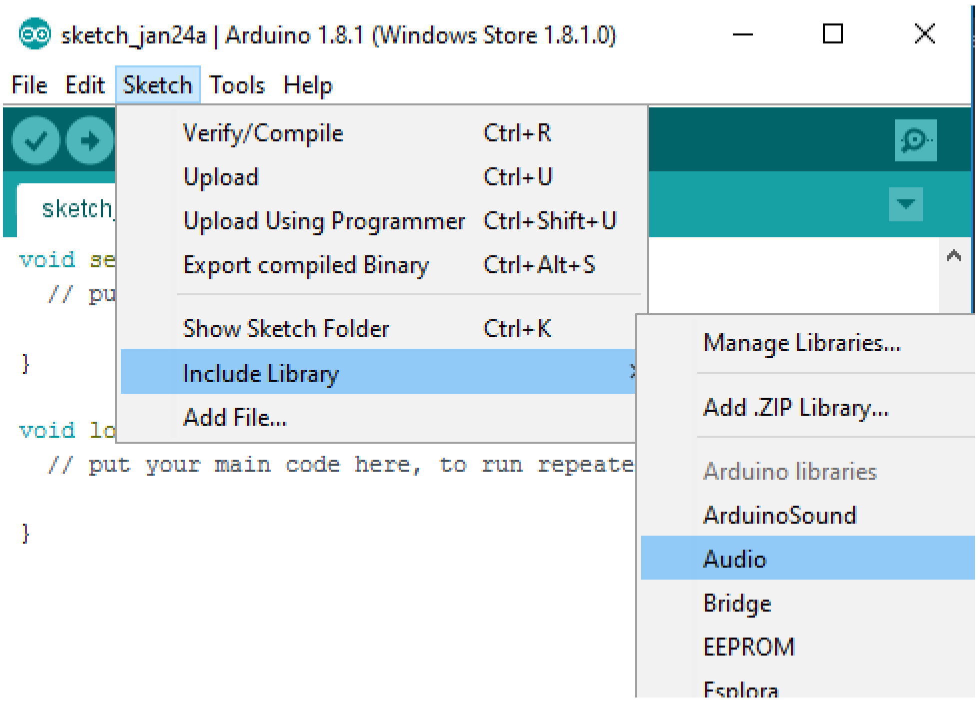


Now your library is installed. In order to use it, you need to include it in your sketch.

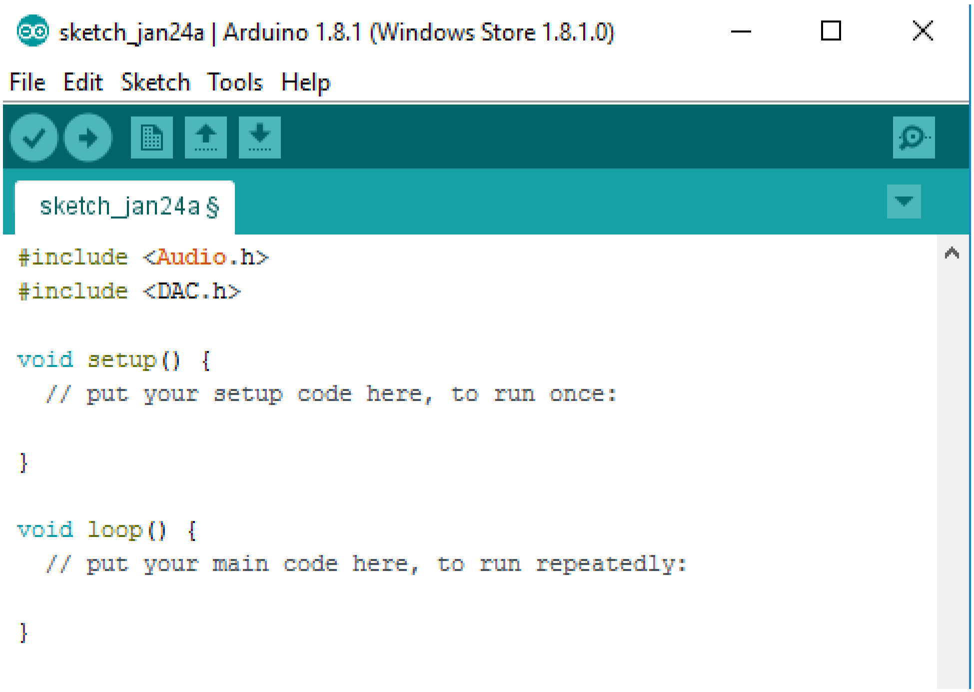
Including libraries in your sketch.

Once you have installed a library, you need to include it in your sketch in order to use it.

- **Open the Sketch Menu > Include Library and click the Library you want to include.**



- **Now, the IDE has generated the required inclusion tags into your code.**



Now the Library is included in your sketch, and you can use it in your code.

Read Libraries online: https://riptutorial.com/arduino/topic/8896/libraries

Chapter 16: Liquid Crystal Library

Introduction

Arduino's `Liquid Crystal Library` is a library for controlling LCD displays compatible the Hitachi HD44780 driver, characterised by their 16 pin interface. The 16 pins might be connected via an I2C interface. These displays contain a matrix of 5x7 pixel blocks used to display characters or small monochromatic images. The displays are usually named according to how many rows and columns they have, e.g. 16x2 or 1602 for 16 columns and 2 rows, and 20x4 or 2004 for 20 columns and 4 rows.

Syntax

- #include <LiquidCrystal.h> // Includes the library
- LiquidCrystal(rs, enable, d4, d5, d6, d7) //
- LiquidCrystal(rs, rw, enable, d4, d5, d6, d7)
- LiquidCrystal(rs, enable, d0, d1, d2, d3, d4, d5, d6, d7)
- LiquidCrystal(rs, rw, enable, d0, d1, d2, d3, d4, d5, d6, d7)

Parameters

LiquidCrystal Parameter	Details
rs	the number of the Arduino pin that is connected to the RS pin on the LCD
rw	the number of the Arduino pin that is connected to the RW pin on the LCD (optional)
enable	the number of the Arduino pin that is connected to the enable pin on the LCD
d0 - d7	the numbers of the Arduino pins that are connected to the corresponding data pins on the LCD. d0, d1, d2, and d3 are optional; if omitted, the LCD will be controlled using only the four data lines (d4, d5, d6, d7).

Examples

Basic Usage

```
/*
  Wiring:
   LCD pin 1 (VSS) -> Arduino Ground
   LCD pin 2 (VDD) -> Arduino 5V
```

```
   LCD pin 3 (VO)  -> Arduino Ground
   LCD pin 4 (RS)  -> Arduino digital pin 12
   LCD pin 5 (RW)  -> Arduino Ground
   LCD pin 6 (E)   -> Arduino digital pin 11
   LCD pin 11 (D4) -> Arduino digital pin 5
   LCD pin 12 (D5) -> Arduino digital pin 4
   LCD pin 13 (D6) -> Arduino digital pin 3
   LCD pin 14 (D7) -> Arduino digital pin 2
*/

#include <LiquidCrystal.h> // include the library

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  // start writing on the first row and first column.
  lcd.setCursor(0, 0);
  // Print a message to the LCD.
  lcd.print("hello, world!");
}

void loop() {
  // No need to do anything to keep the text on the display
}
```

Read Liquid Crystal Library online: https://riptutorial.com/arduino/topic/9395/liquid-crystal-library

Chapter 17: Loops

Syntax

- for (declaration, condition, iteration) { }
- while (condition) { }
- do { } while (condition)

Remarks

`General Remark` If you intend to create a loop to wait for something to happen, you're probably on the wrong track here. Rather remember that all code after setup() is run from a method called loop(). So if you need to wait for something, it's easiest to not do anything (or only other independent stuff) and come back to check for the waiting condition next time.

`do { } while(condition)` will not evaluate the condition statement until after the first iteration. This is important to keep in mind if the condition statement has side effects.

Examples

While

A `while` loop will evaluate its condition, and if `true`, it will execute the code inside and start over. That is, as long as its condition evaluates to `true`, the `while` loop will execute over and over.

This loop will execute 100 times, each time adding 1 to the variable `num`:

```
int num = 0;
while (num < 100) {
    // do something
    num++;
}
```

The above loop is equivalent to a `for` loop:

```
for (int i = 0; i < 100; i++) {
    // do something
}
```

This loop will execute forever:

```
while (true) {
    // do something
}
```

The above loop is equivalent to a `for` loop:

```
for (;;) {
    // do something
}
```

For

`for` loops are simplified syntax for a very common loop pattern, which could be accomplished in more lines with a `while` loop.

The following is a common example of a `for` loop, which will execute 100 times and then stop.

```
for (int i = 0; i < 100; i++) {
    // do something
}
```

This is equivalent to a `while` loop:

```
int num = 0;
while (num < 100) {
    // do something
    num++;
}
```

You can create an endless loop by omitting the condition.

```
for (;;) {
    // do something
}
```

This is equivalent to a `while` loop:

```
while (true) {
    // do something
}
```

Do ... While

A `do while` loop is the same as a `while` loop, except that it is guaranteed to execute at least one time.

The following loop will execute 100 times.

```
int i = 0;
do {
    i++;
} while (i < 100);
```

A similar loop, but with a different condition, will execute 1 time.

```
int i = 0;
do {
    i++;
} while (i < 0);
```

If the above loop were merely a `while` loop, it would execute 0 times, because the condition would evaluate to `false` before the first iteration. But since it is a `do while` loop, it executes once, then checks its condition before executing again.

Flow Control

There are some ways to break or change a loop's flow.

`break;` will exit the current loop, and will not execute any more lines within that loop.

`continue;` will not execute any more code within the current iteration of the loop, but will remain in the loop.

The following loop will execute 101 times (i = 0, 1, ..., 100) instead of 1000, due to the `break` statement:

```
for (int i = 0; i < 1000; i++) {
    // execute this repeatedly with i = 0, 1, 2, ...
    if (i >= 100) {
        break;
    }
}
```

The following loop will result in `j`'s value being 50 instead of 100, because of the `continue` statement:

```
int j=0;
for (int i = 0; i < 100; i++) {
    if (i % 2 == 0) { // if `i` is even
        continue;
    }
    j++;
}
// j has the value 50 now.
```

Read Loops online: https://riptutorial.com/arduino/topic/2802/loops

Chapter 18: MIDI Communication

Introduction

The intent of this topic to demonstrate some basic MIDI programs that show how to operate with the protocol and progressively add useful features that more complex applications require.

Examples

MIDI THRU Example

The MIDI Thru is simple and easy to test. When working properly you will be able to install your Arduino project between two MIDI devices, MIDI IN to MIDI OUT and you will be able to verify that the two device operate together. If you have the ability to measure latency, you will see an increase due to the serial buffer capture and re-transmit instructions.

```
// This is a simple MIDI THRU.  Everything in, goes right out.
// This has been validate on an Arduino UNO and a Olimex MIDI Shield

boolean byteReady;
unsigned char midiByte;

void setup() {
    // put your setup code here, to run once:
    //  Set MIDI baud rate:
    Serial.begin(31250);
    byteReady = false;
    midiByte = 0;
}

// The Loop that always gets called...
void loop() {
   if (byteReady) {
        byteReady = false;
        Serial.write(midiByte);
    }
}

// The little function that gets called each time loop is called.
// This is automated somwhere in the Arduino code.
void serialEvent() {
  if (Serial.available()) {
    // get the new byte:
    midiByte = (unsigned char)Serial.read();
    byteReady = true;
  }
}
```

MIDI Thru with Queue

```
// This is a more complex MIDI THRU.  This version uses a queue.  Queues are important because
```

```
some
// MIDI messages can be interrupted for real time events.  If you are generating your own
messages,
// you may need to stop your message to let a "real time" message through and then resume your
message.


#define QUEUE_DEPTH 128

// Queue Logic for storing messages
int headQ = 0;
int tailQ = 0;
unsigned char tx_queue[QUEUE_DEPTH];

void setup() {
    // put your setup code here, to run once:
    //  Set MIDI baud rate:
    Serial.begin(31250);
}

// getQDepth checks for roll over.  Folks have told me this
// is not required.  Feel free to experiment.
int getQDepth() {
int depth = 0;
    if (headQ < tailQ) {
        depth = QUEUE_DEPTH - (tailQ - headQ);
    } else {
        depth = headQ - tailQ;
    }
    return depth;
}

void addQueue (unsigned char myByte) {
    int depth = 0;
    depth = getQDepth();

    if (depth < (QUEUE_DEPTH-2)) {
        tx_queue[headQ] = myByte;
        headQ++;
        headQ = headQ % QUEUE_DEPTH; // Always keep the headQ limited between 0 and 127
    }
}

unsigned char deQueue() {
    unsigned char myByte;
    myByte = tx_queue[tailQ];
    tailQ++;
    tailQ = tailQ % QUEUE_DEPTH;  // Keep this tailQ contained within a limit
    // Now that we dequeed the byte, it must be sent.
    return myByte;
}

void loop() {
   if (getQDepth>0) {
        Serial.write(deQueue());
    }
}

// The little function that gets called each time loop is called.
// This is automated somwhere in the Arduino code.
void serialEvent() {
```

```
  if (Serial.available()) {
    // get the new byte:
    addQueue((unsigned char)Serial.read());;
  }
}
```

MIDI Clock Generation

```
// This is a MiDI clk generator.  This takes a #defined BPM and
// makes the appropriate clk rate.  The queue is used to let other messages
// through, but allows a clock to go immediately to reduce clock jitter

#define QUEUE_DEPTH 128
#define BPM 121
#define MIDI_SYSRT_CLK 0xF8

// clock tracking and calculation
unsigned long lastClock;
unsigned long captClock;
unsigned long clk_period_us;

// Queue Logic for storing messages
int headQ = 0;
int tailQ = 0;
unsigned char tx_queue[QUEUE_DEPTH];

void setup() {
    //  Set MIDI baud rate:
    Serial.begin(31250);
    clk_period_us = 60000000 / (24 * BPM);
    lastClock = micros();
}

// getQDepth checks for roll over.  Folks have told me this
// is not required.  Feel free to experiment.
int getQDepth() {
int depth = 0;
    if (headQ < tailQ) {
        depth = QUEUE_DEPTH - (tailQ - headQ);
    } else {
        depth = headQ - tailQ;
    }
    return depth;
}

void addQueue (unsigned char myByte) {
    int depth = 0;
    depth = getQDepth();

    if (depth < (QUEUE_DEPTH-2)) {
        tx_queue[headQ] = myByte;
        headQ++;
        headQ = headQ % QUEUE_DEPTH; // Always keep the headQ limited between 0 and 127
    }
}

unsigned char deQueue() {
    unsigned char myByte;
    myByte = tx_queue[tailQ];
```

```
    tailQ++;
    tailQ = tailQ % QUEUE_DEPTH;  // Keep this tailQ contained within a limit
    // Now that we dequeed the byte, it must be sent.
    return myByte;
}

void loop() {
    captClock = micros();

    if (lastClock > captClock) {
        // we have a roll over condition - Again, maybe we don't need to do this.
        if (clk_period_us <= (4294967295 - (lastClock - captClock))) {
            // Add a the ideal clock period for this BPM to the last measurement value
            lastClock = lastClock + clk_period_us;
            // Send a clock, bypasing the transmit queue
            Serial.write(MIDI_SYSRT_CLK);
        }
    } else if (clk_period_us <= captClock-lastClock) {
        // Basically the same two commands above, but not within a roll over check
        lastClock = lastClock + clk_period_us;
        // Send a clock, bypasing the transmit queue
        Serial.write(MIDI_SYSRT_CLK);
    }

    if (getQDepth>0) {
        Serial.write(deQueue());
    }
}

// The little function that gets called each time loop is called.
// This is automated somwhere in the Arduino code.
void serialEvent() {
  if (Serial.available()) {
    // get the new byte:
    addQueue((unsigned char)Serial.read());;
  }
}
```

MIDI Messages Defined

In general, MIDI protocol is broken down into "messages". There are 4 general classes of messages:

- Channel Voice
- Channel Mode
- System Common
- System Real-Time Messages

Messages start with a byte value above 0x80. Any value below 0x7F is considered data. Effectively meaning that 127 is the maximum value that can be encoded into a single MIDI data byte. To encode larger values, two or more MIDI data bytes are required.

It should be pointed out that messages must be sent start to finish without interruption... EXCEPT... System Real-Time messages, which are a single byte, which can be injected in the middle of any message.

Channel Voice Messages

Status D7..D0	Data Bytes	Description
1000nnnn	0kkkkkkk 0vvvvvvv	Note Off event.This message is sent when a note is released (ended). (kkkkkkk) is the key (note) number. (vvvvvvv) is the velocity.
1001nnnn	0kkkkkkk 0vvvvvvv	Note On event. This message is sent when a note is depressed (start). (kkkkkkk) is the key (note) number. (vvvvvvv) is the velocity.
1010nnnn	0kkkkkkk 0vvvvvvv	Polyphonic Key Pressure (Aftertouch). This message is most often sent by pressing down on the key after it "bottoms out". (kkkkkkk) is the key (note) number. (vvvvvvv) is the pressure value.
1011nnnn	0ccccccc 0vvvvvvv	Control Change. This message is sent when a controller value changes. Controllers include devices such as pedals and levers. Controller numbers 120-127 are reserved as "Channel Mode Messages" (below). (ccccccc) is the controller number (0-119). (vvvvvvv) is the controller value (0-127).
1100nnnn	0ppppppp	Program Change. This message sent when the patch number changes. (ppppppp) is the new program number.
1101nnnn	0vvvvvvv	Channel Pressure (After-touch). This message is most often sent by pressing down on the key after it "bottoms out". This message is different from polyphonic after-touch. Use this message to send the single greatest pressure value (of all the current depressed keys). (vvvvvvv) is the pressure value.
1110nnnn	0lllllll 0mmmmmmm	Pitch Bend Change. This message is sent to indicate a change in the pitch bender (wheel or lever, typically). The pitch bender is measured by a fourteen bit value. Center (no pitch change) is 2000H. Sensitivity is a function of the receiver, but may be set using RPN 0. (lllllll) are the least significant 7 bits. (mmmmmmm) are the most significant 7 bits.

Channel Mode Messages

Status D7..D0	Data Bytes	Description
1011nnnn	0ccccccc 0vvvvvvv	Channel Mode Messages. This the same code as the Control Change (above), but implements Mode control and special message by using reserved controller numbers 120-127. The

Status D7..D0	Data Bytes	Description
		commands are:
		All Sound Off. When All Sound Off is received all oscillators will turn off, and their volume envelopes are set to zero as soon as possible. c = 120, v = 0: All Sound Off
		Reset All Controllers. When Reset All Controllers is received, all controller values are reset to their default values. (See specific Recommended Practices for defaults).
		c = 121, v = x: Value must only be zero unless otherwise allowed in a specific Recommended Practice.
		Local Control. When Local Control is Off, all devices on a given channel will respond only to data received over MIDI. Played data, etc. will be ignored. Local Control On restores the functions of the normal controllers.
		c = 122, v = 0: Local Control Off
		c = 122, v = 127: Local Control On
		All Notes Off. When an All Notes Off is received, all oscillators will turn off.
		c = 123, v = 0: All Notes Off (See text for description of actual mode commands.)
		c = 124, v = 0: Omni Mode Off
		c = 125, v = 0: Omni Mode On
		c = 126, v = M: Mono Mode On (Poly Off) where M is the number of channels (Omni Off) or 0 (Omni On)
		c = 127, v = 0: Poly Mode On (Mono Off) (Note: These four messages also cause All Notes Off)

System Common Messages

Status D7..D0	Data Bytes	Description
11110000	0iiiiiii [0iiiiiii 0iiiiiii] 0ddddddd --- --- 0ddddddd	System Exclusive. This message type allows manufacturers to create their own messages (such as bulk dumps, patch parameters, and other non-spec data) and provides a

Status D7..D0	Data Bytes	Description
	11110111	mechanism for creating additional MIDI Specification messages. The Manufacturer's ID code (assigned by MMA or AMEI) is either 1 byte (0iiiiiii) or 3 bytes (0iiiiiii 0iiiiiii 0iiiiiii). Two of the 1 Byte IDs are reserved for extensions called Universal Exclusive Messages, which are not manufacturer-specific. If a device recognizes the ID code as its own (or as a supported Universal message) it will listen to the rest of the message (0ddddddd). Otherwise, the message will be ignored. (Note: Only Real-Time messages may be interleaved with a System Exclusive.)
11110001	0nnndddd	MIDI Time Code Quarter Frame. nnn = Message Type dddd = Values
11110010	0lllllll 0mmmmmmm	Song Position Pointer. This is an internal 14 bit register that holds the number of MIDI beats (1 beat= six MIDI clocks) since the start of the song. l is the LSB, m the MSB.
11110011	0sssssss	Song Select. The Song Select specifies which sequence or song is to be played.
11110100		Undefined. (Reserved)
11110101		Undefined. (Reserved)
11110110		Tune Request. Upon receiving a Tune Request, all analog synthesizers should tune their oscillators.
11110111		End of Exclusive. Used to terminate a System Exclusive dump (see above).

System Real-Time Messages

Status D7..D0	Data Bytes	Description
11111000		Timing Clock. Sent 24 times per quarter note when synchronization is required (see text).
11111001		Undefined. (Reserved)
11111010		Start. Start the current sequence playing. (This message will be followed with Timing Clocks).
11111011		Continue. Continue at the point the sequence was Stopped.

Status D7..D0	Data Bytes	Description
11111100		Stop. Stop the current sequence.
11111101		Undefined. (Reserved)
11111110		Active Sensing. This message is intended to be sent repeatedly to tell the receiver that a connection is alive. Use of this message is optional. When initially received, the receiver will expect to receive another Active Sensing message each 300ms (max), and if it does not then it will assume that the connection has been terminated. At termination, the receiver will turn off all voices and return to normal (non- active sensing) operation.
11111111		Reset. Reset all receivers in the system to power-up status. This should be used sparingly, preferably under manual control. In particular, it should not be sent on power-up.

Read MIDI Communication online: https://riptutorial.com/arduino/topic/9406/midi-communication

Chapter 19: PWM - Pulse Width Modulation

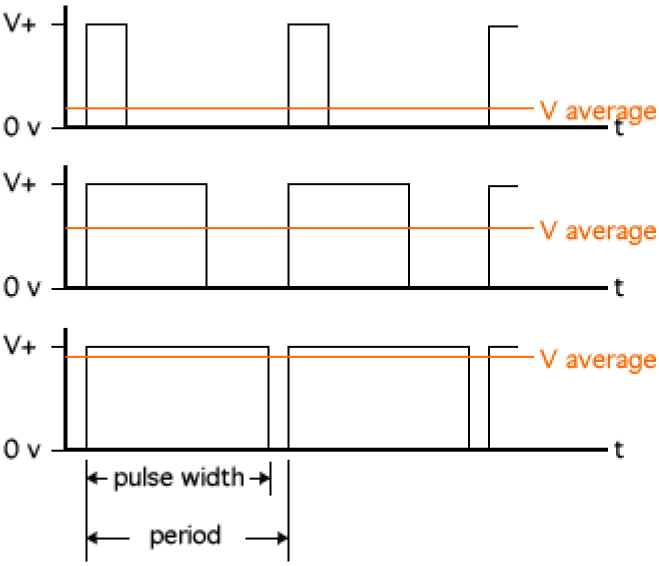
Examples

Control a DC motor through the Serial port using PWM

In this example we aim to accomplish one of the most common tasks: *I have a small DC motor laying around, how do I use my Arduino to control it?* Easy, with PWM and serial communication, using the function `analogWrite()` and the `Serial` library.

The basics

Pulse Width Modulation or PWM for short is a technique for mimicking analog signals using digital output. How does this work? Using a pulse train whose relation D (duty cycle) between time at high level (digital 1, usually 5V) and time at low level (digital 0, 0V) in each period can be modified to produce an average voltage between these two levels:



By using Arduino's `analogWrite(pin,value)` function we can vary the `value` of the duty cycle of `pin`'s output. Note that the `pin` must be put into output mode and the `value` must be between 0 (0V) and 255 (5V). Any value in between will simulate a proportional intermediate analog output.

However, the purpose of analog signals is usually related to the control of mechanical systems that require more voltage and current than the Arduino board alone is capable of. In this example, we will learn how to amplify Arduino's PWM capabilities.

For this a MOSFET diode is used. In essence, this diode acts as a switch. It allows or interrupts the electric flow between its *source* and *drain* terminals. But instead of a mechanical switch, it features a third terminal called *gate*. A very small current (<1mA) will "open" this gate and allow the current to flow. This is very convenient, because we can send Arduino's PWM output to this gate, thereby creating *another* PWM pulse train with the same duty cycle through the MOSFET, which allows voltages and currents that would destroy the Arduino.

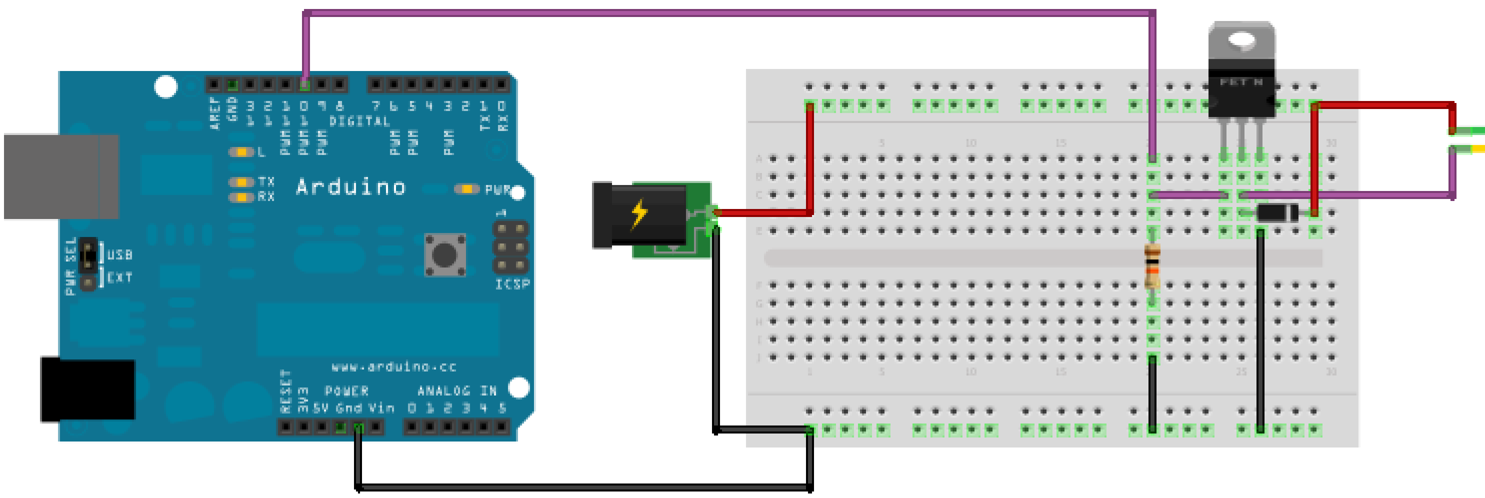
Bill of materials: what do you need to build this example

- MOSFET diode: for instance, the popular BUZ11
- Protection diode for the motor: Schottky SB320
- Resistor: anything 10K ~ 1M Ohm
- Motor: A typical small motor (a typical one can be 12V)
- A power source compatible with the motor you have selected
- A breadboard
- Colorful cables!
- An Arduino, but you already knew that.

The build

Put everything together! Power the rails of the breadboard and place the MOSFET diode in it. Connect the motor between the positive rail and the MOSFET drain. Connect the protection diode in the same way: between the MOSFET drain and the positive rail. Connect the source of the MOSFET to the common ground rail. Finally, connect the PWM pin (we're using pin 10 in this example) to the gate of the MOSFET and also to the common ground through the resistor (we need very low current!).

Here's an example of how this build looks. If you prefer an scheme here's one.



The code

Now we can connect the Arduino to a computer, upload the code and control the motor, by sending values through the serial communication. Recall that these values should be integers between 0 and 255. The actual code of this example is very simple. An explanation is provided in each line.

```
int in = 0;                  // Variable to store the desired value
byte pinOut = 10;            // PWM output pin

void setup() {               // This executes once
  Serial.begin(9600);            // Initialize serial port
  pinMode(pinOut, OUTPUT);       // Prepare output pin
}

void loop() {                // This loops continuously
  if(Serial.available()){        // Check if there's data
    in = Serial.read();          // Read said data into the variable "in"
    analogWrite(pinOut, in);     // Pass the value of "in" to the pin
  }
}
```

And that's it! Now you can use Arduino's PWM capabilities to control applications that require analog signals even when the power requirements exceed the board's limits.

PWM with a TLC5940

The TLC5940 is a handy item to have when you run out of PWM ports on the Arduino. It has 16 channels, each individually controllable with 12 bits of resolution (0-4095). An existing library is available at http://playground.arduino.cc/Learning/TLC5940. It is useful for controlling multiple servos or RGB LEDs. Just keep in mind, the LEDs must be common anode to work. Also, the chips are daisy-chainable, allowing even more PWM ports.

Example:

```
// Include the library
#include <Tlc5940.h>

void setup() {
    // Initialize
    Tlc.init();
    Tlc.clear();
}

unsigned int level = 0;
void loop() {
    // Set all 16 outputs to same value
    for (int i = 0; i < 16; i++) {
        Tlc.set(i, level);
    }
    level = (level + 1) % 4096;
    // Tell the library to send the values to the chip
    Tlc.update();
    delay(10);
}
```

Read PWM - Pulse Width Modulation online: https://riptutorial.com/arduino/topic/1658/pwm---pulse-width-modulation

Chapter 20: Random Numbers

Syntax

- random(max) //Returns a (long) pseudo-random number between 0 (inclusive) and max (exclusive)
- random(min, max) //Returns a (long) pseudo-random number between min (inclusive) and max (exclusive)
- randomSeed(seed) //Initializes de pseudo-random number generator, causing it to start at a specified point in its sequence.

Parameters

Parameter	Details
min	The minimum possible value (inclusive) to be generated by the `random()` function.
max	The maximum possible value (exclusive) to be generated by the `random()` function.
seed	The seed that will be used to shuffle the `random()` function.

Remarks

If `randomSeed()` is called with a fixed value (eg. `randomSeed(5)`), the sequence of random numbers generated by the sketch will repeat each time it is run. In most cases, a random seed is preferred, which can be obtained by reading an unconnected analog pin.

Examples

Generate a random number

The `random()` function can be used to generate pseudo-random numbers:

```
void setup() {
    Serial.begin(9600);
}

void loop() {
    long randomNumber = random(500);  // Generate a random number between 0 and 499
    Serial.println(randomNumber);

    randomNumber = random(100, 1000); // Generate a random number between 100 and 999
```

```
    Serial.println(randomNumber);

    delay(100);
}
```

Setting a seed

If it is important for a sequence of numbers generated by `random()` to differ, it is a good idea to specify a seed with `randomSeed()`:

```
void setup() {
    Serial.begin(9600);

    // If analog pin 0 is left unconnected, analogRead(0) will produce a
    // different random number each time the sketch is run.
    randomSeed(analogRead(0));
}

void loop() {
    long randomNumber = random(500); // Generate a random number between 0 and 499
    Serial.println(randomNumber);

    delay(100);
}
```

Read Random Numbers online: https://riptutorial.com/arduino/topic/2238/random-numbers

Chapter 21: Serial Communication

Syntax

- `Serial.begin(speed)` // Opens the serial port on the given baud rate
- `Serial.begin(speed, config)`
- `Serial[1-3].begin(speed)` // **Arduino Mega only!** When writing 1-3 it means you can choose between the numbers 1 to 3 when choosing the serial port.
- `Serial[1-3].begin(speed, config)` // **Arduino Mega only!** When writing 1-3 it means you can choose between the numbers 1 to 3 when choosing the serial port.
- `Serial.peek()` // Reads the next byte of input without removing it from the buffer
- `Serial.available()` // Gets the number of bytes in the buffer
- `Serial.print(text)` // Writes text to the serial port
- `Serial.println(text)` // Same as `Serial.print()` but with a trailing newline

Parameters

Parameter	Details
Speed	The rate of the serial port (usually 9600)
Text	The text to write to the serial port (any data type)
Data bits	Number of data bits in a packet (from 5 - 8), default is 8
Parity	Parity options for error detection: none (default), even, odd
Stop bits	Number of stop bits in a packet: one (default), two

Remarks

The Arduino Mega has four serial ports which there can be choosed from. They are accesed in the following way

```
Serial.begin(9600);
Serial1.begin(38400);
Serial2.begin(19200);
Serial3.begin(4800);
```

The serial port on an Arduino can be set with additional parameters. The config parameter sets data bits, parity, and stop bits. For example:

8 data bits, even parity and 1 stop bit would be - `SERIAL_8E1`

6 data bits, odd parity and 2 stop bit would be - `SERIAL_6O2`

7 data bits, no parity and 1 stop bit would be - `SERIAL_7N1`

Examples

Simple read and write

This example listens for input coming in over the serial connection, then repeats it back out the same connection.

```
byte incomingBytes;

void setup() {
  Serial.begin(9600); // Opens serial port, sets data rate to 9600 bps.
}

void loop() {
  // Send data only when you receive data.
  if (Serial.available() > 0) {
    // Read the incoming bytes.
    incomingBytes = Serial.read();

    // Echo the data.
    Serial.println(incomingBytes);
  }
}
```

Base64 filtering for serial input data

```
String base64="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=";

void setup() {

    Serial.begin(9600); // Turn the serial protocol ON
    Serial.println("Start Typing");
}

void loop() {

    if (Serial.available() > 0) { // Check if data has been sent from the user
        char c = Serial.read();   // Gets one byte/Character from serial buffer
        int result = base64.indexOf(c); // Base64 filtering
        if (result>=0)
            Serial.print(c); // Only print Base64 string
    }
}
```

Command Handling over Serial

```
byte incoming;
String inBuffer;

void setup() {
    Serial.begin(9600); // or whatever baud rate you would like
}
```

```
void loop(){
    // setup as non-blocking code
    if(Serial.available() > 0) {
        incoming = Serial.read();

        if(incoming == '\n') {  // newline, carriage return, both, or custom character

            // handle the incoming command
            handle_command();

            // Clear the string for the next command
            inBuffer = "";
        } else{
            // add the character to the buffer
            inBuffer += incoming;
        }
    }

    // since code is non-blocking, execute something else . . . .
}

void handle_command() {
    // expect something like 'pin 3 high'
    String command = inBuffer.substring(0, inBuffer.indexOf(' '));
    String parameters = inBuffer.substring(inBuffer.indexOf(' ') + 1);

    if(command.equalsIgnoreCase('pin')){
        // parse the rest of the information
        int pin = parameters.substring("0, parameters.indexOf(' ')).toInt();
        String state = parameters.substring(parameters.indexOf(' ') + 1);

        if(state.equalsIgnoreCase('high')){
            digitalWrite(pin, HIGH);
        }else if(state.equalsIgnoreCase('low)){
            digitalWrite(pin, LOW);
        }else{
            Serial.println("did not compute");
        }
    } // add code for more commands
}
```

Serial Communication with Python

If you have an Arduino connected to a computer or a Raspberry Pi, and want to send data from the Arduino to the PC you can do the following:

Arduino:

```
void setup() {
  // Opens serial port, sets data rate to 9600 bps:
  Serial.begin(9600);
}

void loop() {
  // Sends a line over serial:
```

```
  Serial.println("Hello, Python!");
  delay(1000);
}
```

Python:

```
import serial


ser = serial.Serial('/dev/ttyACM0', 9600)  # Start serial communication
while True:
    data = ser.readline()  # Wait for line from Arduino and read it
    print("Received: '{}'".format(data))  # Print the line to the console
```

Read Serial Communication online: https://riptutorial.com/arduino/topic/1674/serial-communication

Chapter 22: Servo

Introduction

A Servo is a an enclosed system containing a motor and some supporting circuitry. The shaft of a servo can be rotated to a fixed angle within an arc using a control signal. If the control signal is maintained, then the servo will maintain its angle. Servos can easily be controlled with the Arduino `Servo.h` library.

Syntax

- #include <Servo.h> // Include the Servo library
- Servo.attach(pin) // Attach to the servo on pin. Returns a Servo object
- Servo.write(degrees) // Degrees to move to (0 - 180)
- Servo.read() // Gets the current rotation of the servo

Examples

Moving the servo back and forth

```
#include <Servo.h>

Servo srv;

void setup() {
  srv.attach(9); // Attach to the servo on pin 9

}
```

To use a servo, you need to call `attach()` function first. It starts generating a PWM signal controlling a servo on a specified pin. On boards other than Arduino Mega, use of Servo library disables analogWrite() (PWM) functionality on pins 9 and 10, whether or not there is a Servo on those pins.

```
void loop() {
  Servo.write(90); // Move the servo to 90 degrees
  delay(1000); // Wait for it to move to it's new position
  Servo.write(0); // Move the servo to 0 degrees
  delay(1000); // Wait for it to move to it's new position
}
```

Note that you are not guaranteed that the servo reached the desired position, nor you can check it from the program.

Read Servo online: https://riptutorial.com/arduino/topic/4920/servo

Chapter 23: SPI Communication

Remarks

Chip select signals

Most slaves have an active low chip select input. So proper code to initialize and use a chip select pin is this:

```
#define CSPIN 1 // or whatever else your CS pin is
// init:
pinMode(CSPIN, OUTPUT);
digitalWrite(CSPIN, 1); // deselect

// use:
digitalWrite(CSPIN, 0); // select
... perform data transfer ...
digitalWrite(CSPIN, 1); // deselect
```

Deselecting a slave is just as important as selecting it, because a slave may drive the MISO line while it is selected. There may be many slaves, but only one may drive MISO. If a slave is not deselected properly, two or more slaves might be driving MISO, which may lead to shorts between their outputs and might damage the devices.

Transactions

Transactions serve two purposes:

- tell the SPI when we want to start and end using it within a particular context
- configure the SPI for a specific chip

The clock line has different idle states in the different SPI modes. Changing the SPI mode while a slave is selected might confuse the slave, so always set the SPI mode before selecting a slave. The SPI mode can be set with an `SPISettings` object passed to `SPI.beginTransaction`:

```
SPI.beginTransaction(SPISettings(1000000, MSBFIRST, SPI_MODE0));
digitalWrite(CSPIN, 0);
... perform data transfer ...
digitalWrite(CSPIN, 1);
SPI.endTransaction();
```

`SPISettings` may also be stored elsewhere:

```
SPISettings mySettings(1000000, MSBFIRST, SPI_MODE0);
SPI.beginTransaction(mySettings);
```

If another part of the code tries to use the SPI between a pair of calls to `beginTransaction()` and `endTransaction()`, an error may be raised - how that is done depends on the implementation.

Also see Arduino Reference: SPISettings

Using the SPI in Interrupt Service Routines

If the SPI has to be used within an ISR, no other transaction may be taking place at the same time. The SPI library provides `usingInterrupt(interrupt_number)` to facilitate this. It works by disabling the given interrupt whenever `beginTransaction()` is called, so the interrupt cannot fire between that pair fo calls to `beginTransaction()` and `endTransaction()`.

Also see Arduino Reference: SPI: usingInterrupt

Examples

Basics: initialize the SPI and a chip select pin, and perform a 1-byte transfer

```
#include <SPI.h>
#define CSPIN 1

void setup() {
  pinMode(CSPIN, OUTPUT); // init chip select pin as an output
  digitalWrite(CSPIN, 1); // most slaves interpret a high level on CS as "deasserted"

  SPI.begin();

  SPI.beginTransaction(SPISettings(1000000, MSBFIRST, SPI_MODE0));
  digitalWrite(CSPIN, 0);

  unsigned char sent = 0x01;
  unsigned char received = SPI.transfer(sent);
  // more data could be transferred here

  digitalWrite(CSPIN, 1);
  SPI.endTransaction();

  SPI.end();
}

void loop() {
  // we don't need loop code in this example.
}
```

This example:

- properly initializes and uses a chip select pin (see remarks)
- properly uses an SPI transaction (see remarks)
- only uses the SPI to transfer one single byte. There is also a method for transferring arrays, which is not used here.

Read SPI Communication online: https://riptutorial.com/arduino/topic/4919/spi-communication

Chapter 24: Time Management

Syntax

- unsigned long millis()
- unsigned long micros()
- void delay(unsigned long milliseconds)
- void delayMicroseconds(unsigned long microseconds)
- See the elapsedMillis header for constructors and operators of that class. In short:
    - elapsedMillis elapsedMillisObject; *creates an object to keep track of time since it was created or since some other explicitly set point in time*
    - elapsedMillisObject = 0; *reset the time tracked by the object to "since now"*
    - unsigned long deltaT = elapsedMillisObject; *lets us look at the tracked time*
    - elapsedMillisObject += and -= *these work as expected*

Remarks

Blocking vs. non-blocking code

For very simple sketches, writing blocking code using `delay()` and `delayMicroseconds()` can be appropriate. When things get more complex, using these functions can have some drawbacks. Some of these are:

- Wasting CPU time: More complex sketches might need the CPU for something else while waiting for an LED blinking period to end.
- unexpected delays: when `delay()` is called in subroutines that are not obviously called, for example in libraries you include.
- missing events that happen during the delay and are not handled by an interrupt handler, for example polled button presses: A button might be pressed for 100 ms, but this might be shadowed by a `delay(500)`.

Implementation details

`millis()` usually relies on a hardware timer that runs at a speed that's much higher than 1 kHz. When `millis()` is called, the implementation returns some value, but you don't know how old that actually is. It's possible that the "current" millisecond just started, or that it will end right after that function call. That means that, when calculating the difference between two results from `millis()`, you can be off by anything between almost zero and almost one millisecond. Use `micros()` if higher precision is needed.

Looking into the source code of `elapsedMillis` reveals that it indeed uses `millis()` internally to compare two points in time, so it suffers from this effect as well. Again, there's the alternative `elapsedMicros` for higher precision, from the same library.

Examples

blocking blinky with delay()

One of the most straight forward way of making an LED blink is: turn it on, wait a bit, turn it off, wait again, and repeat endlessly:

```
// set constants for blinking the built-in LED at 1 Hz
#define OUTPIN LED_BUILTIN
#define PERIOD 500

void setup()
{
  pinMode(OUTPIN, OUTPUT);      // sets the digital pin as output
}

void loop()
{
  digitalWrite(OUTPIN, HIGH);   // sets the pin on
  delayMicroseconds(PERIOD);        // pauses for 500 miliseconds
  digitalWrite(OUTPIN, LOW);    // sets the pin off
  delayMicroseconds(PERIOD);        // pauses for 500 milliseconds

  // doing other time-consuming stuff here will skew the blinking
}
```

However, waiting as done in the example above wastes CPU cycles, because it just sits there in a loop waiting for a certain point in time to go past. That's what the non-blocking ways, using `millis()` or `elapsedMillis`, do better - in the sense that they don't burn as much of the hardware's capabilities.

Non-blocking blinky with the elapsedMillis library (and class)

The elapsedMillis library provides a class with the same name that keeps track of the time that passed since it was created or set to a certain value:

```
#include <elapsedMillis.h>

#define OUTPIN LED_BUILTIN
#define PERIOD 500

elapsedMillis ledTime;

bool ledState = false;

void setup()
{
  // initialize the digital pin as an output.
  pinMode(OUTPIN, OUTPUT);
```

```
}

void loop()
{
    if (ledTime >= PERIOD)
    {
        ledState = !ledState;
        digitalWrite(OUTPIN, ledState);
        ledTime = 0;
    }
    // do other stuff here
}
```

You can see in the example that the `ledTime` object is assigned zero when the LED pin was toggled. This might not be surprising at first glance, but it has an effect if more time-consuming things are happening:

Consider a situation where the comparison between `ledTime` and `PERIOD` is done after 750 milliseconds. Then setting `ledTime` to zero means that all following toggle operations will be 250 ms "late". If, in contrast, `PERIOD` was subtracted from `ledTime`, the LED would see one short period and then continue blinking as if nothing happened.

Non-blocking blinky with millis()

This is very close to an example from the arduino docs:

```
// set constants for blinking the built-in LED at 1 Hz
#define OUTPIN LED_BUILTIN
#define PERIOD 500  // this is in milliseconds

int ledState = LOW;

// millis() returns an unsigned long so we'll use that to keep track of time
unsigned long lastTime = 0;

void setup() {
  // set the digital pin as output:
  pinMode(OUTPIN, OUTPUT);
}

void loop() {
  unsigned long now = millis();
  if (now - lastTime >= PERIOD) // this will be true every PERIOD milliseconds
  {
    lastTime = now;
    if (ledState == LOW)
    {
      ledState = HIGH;
    }
    else
    {
      ledState = LOW;
    }
    digitalWrite(OUTPIN, ledState);
  }

  // now there's lots of time to do other stuff here
```

```
}
```

Using `millis()` in this way - to time operations in a non-blocking way - is something that is needed quite frequently, so consider using the `elapsedMillis` library for this.

Measure how long something took, using elapsedMillis and elapsedMicros

```
#include <elapsedMillis.h>

void setup() {
  Serial.begin(115200);
  elapsedMillis msTimer;
  elapsedMicros usTimer;

  long int dt = 500;
  delay(dt);

  long int us = usTimer;
  long int ms = msTimer;

  Serial.print("delay(");Serial.print(dt);Serial.println(") took");
  Serial.print(us);Serial.println(" us, or");
  Serial.print(ms);Serial.println(" ms");
}

void loop() {
}
```

In this example, an `elapsedMillis` object and an `elapsedMicros` object are used to measure how long something took, by creating them just before the expression we want to time is executed, and getting their values afterwards. They will show slightly different results, but the millisecond result won't be off by more than one millisecond.

More than 1 task without delay()

If you have more than 1 task to execute repeatedly in different intervals, use this example as a starting point:

```
unsigned long intervals[] = {250,2000}; //this defines the interval for each task in
milliseconds
unsigned long last[] = {0,0};           //this records the last executed time for each task

void setup() {
  pinMode(LED_BUILTIN, OUTPUT); //set the built-it led pin as output
  Serial.begin(115200);         //initialize serial
}

void loop() {
  unsigned long now = millis();
  if(now-last[0]>=intervals[0]){ last[0]=now; firstTask(); }
  if(now-last[1]>=intervals[1]){ last[1]=now; secondTask(); }

  //do other things here
}
```

```
void firstTask(){
  //let's toggle the built-in led
  digitalWrite(LED_BUILTIN, digitalRead(LED_BUILTIN)?0:1);
}

void secondTask(){
  //say hello
  Serial.println("hello from secondTask()");
}
```

To add another task to execute every 15 seconds, extend the variables `intervals` and `last`:

```
unsigned long intervals[] = {250,2000,15000};
unsigned long last[] = {0,0,0};
```

Then add an `if` statement to execute the new task. In this example, I named it `thirdTask`.

```
if(now-last[2]>=intervals[2]){ last[2]=now; thirdTask(); }
```

Finally declare the function:

```
void thirdTask(){
  //your code here
}
```

Read Time Management online: https://riptutorial.com/arduino/topic/4852/time-management

Chapter 25: Using Arduino with Atmel Studio 7

Remarks

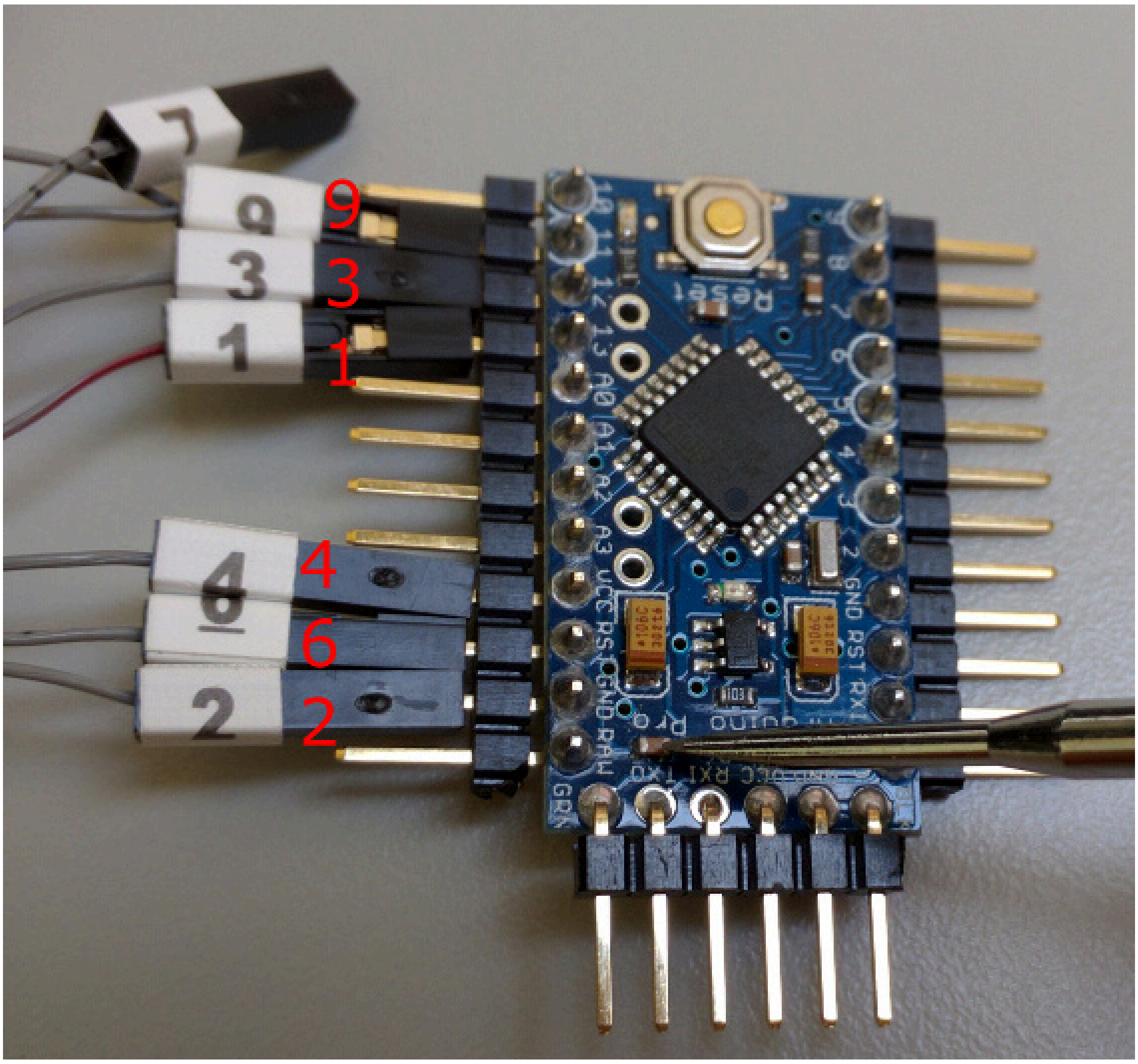
Setup

- Download and install Atmel Studio 7 from here.
- Purchase a debugger. You can get by with a ISP programmer, but if you want debugging capabilities, which is one of the big advantages of using Atmel Studio, you will want a debugger. I recommend the Atmel ICE, as it provides debugging capabilities for AVR based arduinos (like the Uno, pro mini, etc) and the ARM based Arduinos, such as the Zero and Due. If you are on a budget, you can get it without the plastic case and be careful not to shock it.

Connections

- For the Uno, use the 6-pin ICSP cable. Plug one side into the Uno as shown. Plug the other side into the debugger's AVR port.

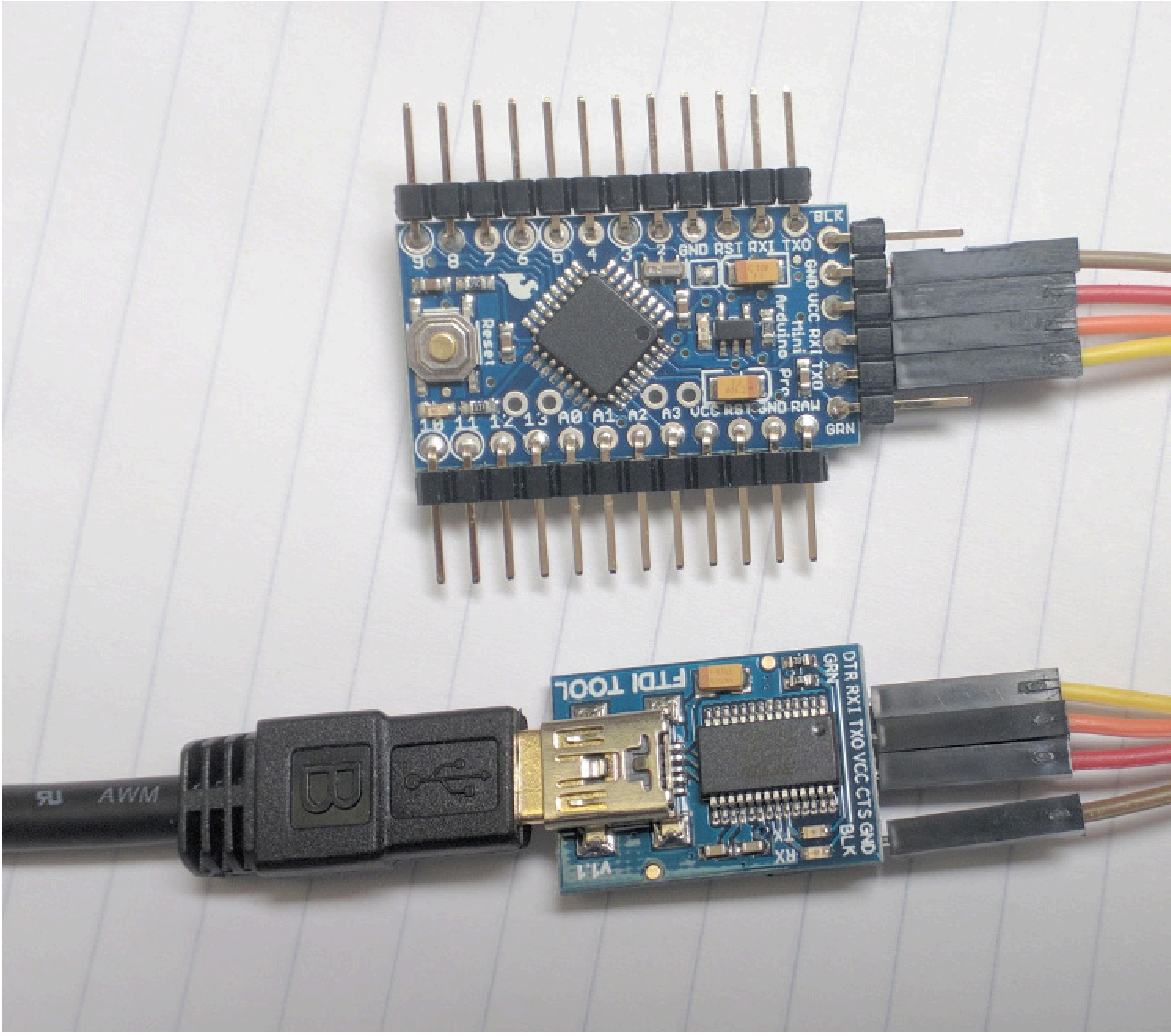
For the Arduino Pro Mini, use the mini squid cable as shown, again connecting the other side the debugger's AVR port.



Debugging considerations

For debugging with the Uno, you will need to cut the Reset-enable trace (you can always solder it back for using with the Arduino IDE):

Using the Pro Mini, if you intend to connect the serial port to your computer using an FTDI board, do not connect the DTR line, as it will interfere with Atmel's Serial Wire Debug (SWD) interface. I simply connect power, ground, Tx and Rx as shown here below. Rx and Tx on Arduino go to Tx and Rx, respectively on FTDI board. Some FTDI boards are labeled differently, so if the serial port doesn't work, swap Rx and Tx.



You will have to provide power separately to the Arduino because the debugger will not power it. This can be done on the Pro Mini through the FTDI board as shown above, or with a USB cable or AC adaptor on the Uno.

Software setup

Plug the Atmel ICE into your computer, start Atmel Studio and you can now import an existing Arduino project.

In Atmel Studio, go to File -> New -> Project and select "Create project from Arduino sketch". Fill out options including board and device dropdown menus.

Go to Project -> yourProjectName Properties, click on Tool, select Atmel ICE under debugger/programmer and debugWire under interface. Go to Debug -> Start debugging and break. You should see a warning and be asked if you want to set the DWEN fuse. Choose OK, unplug the Arduino from power and plug it in again. You can stop debugging by clicking the red square button and start by clicking the green triangle button. To return the Arduino to a state that it can be used in the Arduino IDE, while you're debugging, choose Debug -> disable debugWIRE and close.

Note that any functions you add must include a function prototype as well (loop and setup don't need them). You can see the ones Atmel Studio added at the top of the sketch if there were any functions when you imported your project into Atmel Studio (see sample code for example).

C++11 support is enabled by default in Arduino 1.6.6 and above. This provides more C++ language features and enabling it may increase compatibility with the Arduinio system. To enable C++11 in Atmel Studio 7, right click on your project file, select properties, click on ToolChain on the left, Click on Miscellaneous under AVR/GNU C++ Compiler and put `-std=c++11` in the Other flags field.

To include libraries in your sketch

Copy the .cpp library file into `C:\Users\YourUserName\Documents\Atmel Studio\7.0\YourSolutionName\YourProjectName\ArduinoCore\src\core`, then in Atmel Studio, open the Solution Explorer window right click on the Arduino Core/src/core folder, choose add -> existing item and choose the file you added. Do the same with the .h library file and the YourProjectName/Dependancies folder.

To add the terminal window

You can always have the Android IDE open and use that Serial window (just select the correct serial port), however to add a built in Serial window to Atmel Studio, go to Tools -> Extensions and Updates, click on Available downloads and search for Terminal Window or Terminal for Atmel Studio and install it. Once installed, go to View -> Terminal Window.

Benefits

Programming Arduino with a moder IDE like Atmel Studio 7 gives you numerous advantages over the Arduino IDE, including debugging, autocompletion, jump to definition and declaration, forward/backward navigation, bookmarks and refactoring options to name a few.

You can configure key bindings by going to Tools -> Options -> Environment -> Keyboard. Some that really speed up development are:

- Edit.CommentSelection, Edit.UncommentSelection
- View.NavigateForward, View.NavigateBackward
- Edit.MoveSelectedLinesUp, Edit.MoveSelectedLinesDown
- Edit.GoToDefinition

Examples

Atmel Studio 7 imported sketch example

This is an example of what a simple Arduino sketch looks like after being imported into Atmel Studio. Atmel Studio added the auto generated sections at the top. The rest is identical to the original Arduino code. If you expand the ArduinoCore project that was created and look in the src -> core folder, you will find `main.cpp`, the entry point for the program. There you can see the call to the the Arduino setup function and a never ending for loop that calls the Arduino loop function over and over.

```
/* Begining of Auto generated code by Atmel studio */
#include <Arduino.h>
/* End of auto generated code by Atmel studio */


// Beginning of Auto generated function prototypes by Atmel Studio
void printA();
// End of Auto generated function prototypes by Atmel Studio

void setup() {
  Serial.begin(9600);
}

void loop() {
    printA();
}

void printA() {
    Serial.println("A");
}
```

Read Using Arduino with Atmel Studio 7 online: https://riptutorial.com/arduino/topic/2567/using-arduino-with-atmel-studio-7

Chapter 26: Variables and Data Types

Examples

Create variable

To create a variable:

```
variableType variableName;
```

For example:

```
int a;
```

To create a variable and initialize it:

```
variableType variableName = initialValue;
```

For example:

```
int a = 2;
```

Assign value to a variable

If you have a variable declared before, you can assign some value to it:

For example:

```
int a;  // declared previously
a = 2;
```

Or change the value:

```
int a = 3;  // initalized previously
a = 2;
```

Variable types

- `char` : signed 1-byte character value
- `byte` : unsigned 8-bit integer
- `int` : signed 16-bit (on ATMEGA based boards) or 32-bit (on Arduino Due) integer
- `unsigned int` : unsigned 16-bit (on ATMEGA based boards) or 32-bit (on Arduino Due) integer
- `long` : signed 32-bit integer
- `unsigned long` : unsigned 32-bit integer

- `float` : 4-byte floating point number
- `double` : 4-byte (on ATMEGA based boards) or 8-byte (on Arduino Due) floating point number

Examples:

```
char a = 'A';
char a = 65;

byte b = B10010;

int c = 2;

unsigned int d = 3;

long e = 186000L;

unsigned long f = millis(); // as an example

float g = 1.117;

double h = 1.117;
```

Read Variables and Data Types online: https://riptutorial.com/arduino/topic/2565/variables-and-data-types

Credits

S. No	Chapters	Contributors
1	Getting started with arduino	Abhishek Jain, Christoph, Community, Danny_ds, Doruk, geek1011, gmuraleekrishna, H. Pauwelyn, jleung513, Martin Carney, Mizole Ni, Shef, uruloke, Wolfgang
2	Analog Inputs	Jake Lites, MikeS159, Ouss4, uruloke
3	Arduino IDE	geek1011, Jeremy, jleung513, sohnryang, uruloke
4	Audio Output	Jake Lites, MikeCAT
5	Bluetooth Communication	Girish, Martin Carney
6	Data Storage	Danny_ds, robert
7	Digital Inputs	Martin Carney, uruloke
8	Digital Output	uruloke
9	Functions	datafiddler, Leah, MikeCAT
10	Hardware pins	Jeremy, Martin Carney
11	How Python integrates with Arduino Uno	Danny_ds, Peter Mortensen, Stark Nguyen
12	How to store variables in EEPROM and use them for permanent storage	AZ Vcience, Chris Combs, Danny_ds, Jeremy, Peter Mortensen , RamenChef
13	I2C Communication	Asaf
14	Interrupts	DavidJ, Martin Carney
15	Libraries	Oscar Lundberg
16	Liquid Crystal Library	Morgoth
17	Loops	datafiddler, Martin Carney, MikeCAT

18	MIDI Communication	Rich Maes
19	PWM - Pulse Width Modulation	Danny_ds, Johnny Mopp, JorgeGT, Martin Carney
20	Random Numbers	Danny_ds, Javier Rizzo Aguirre, MikeCAT
21	Serial Communication	blainedwards8, Danny_ds, geek1011, Leah, Martin Carney, MikeS159, Morgoth, Nufail Achath, Peter Mortensen, uruloke
22	Servo	geek1011, mactro, Morgoth
23	SPI Communication	Christoph
24	Time Management	Christoph, Rei
25	Using Arduino with Atmel Studio 7	Danny_ds, Nate
26	Variables and Data Types	Leah, MikeCAT