Digital Human Research Center

Advanced Institutes of Convergence Technology

Tutorial

Arduino Guide using MPU-6050 and nRF24L01

Author:
Daniel titello - Intern

Internship Mentor:
Mathew Schwartz

DHRC Director:
Jaeheung Park

http://dhrc.snu.ac.kr

Suwon, South Korea November 6, 2015

Contents

1 Introduction

This tutorial will explain how to use the Arduino platform Since the creation of a simple arduino sketch until the implementation of more complicate examples using the sensor MPU-6050 and the wireless communication module nRF24L01. All the details to make the sensor and the wireless communication work will be explain step by step in this tutorial, including the library code too.

2 Arduino

2.1 History

Arduino is an open-source platform which allows people easily make project mixing hardware and software. An Arduino board consists of a microcontroller with complementary components that facilitate programming and incorporation into other circuits. Standard connectors are responsible to make so easy the connection between the main board and other auxiliary boards, called Shields.

Arduino is open-source, so everyone has complete access to the schematic and layout boards or a hundred of online examples.

2.2 Software

Although, the Arduino integrated development environment(IDE) was written in Java, the Arduino programs or "Sketchs" are written in C or C++ language. For better working, it is really important keeping update the computer with the last support version of the Arduino IDE.

2.3 How to create a project

The first step is defining the correct Arduino board and serial port.

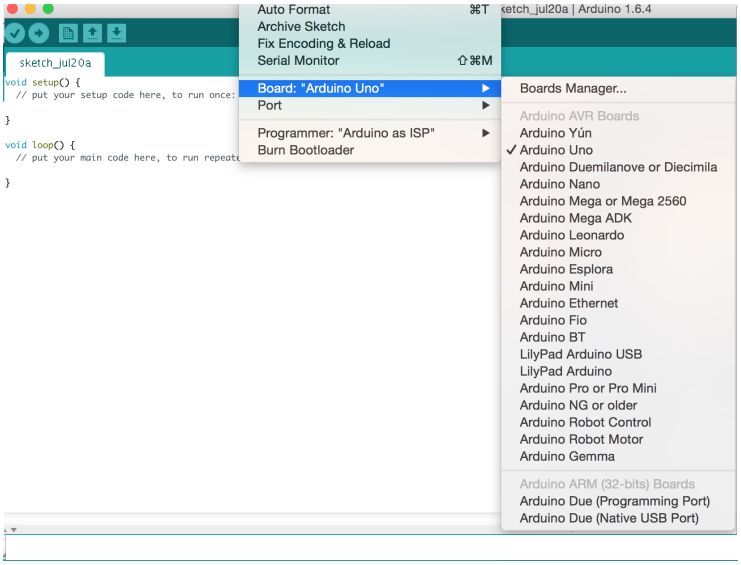


Figure 1: The Arduino board is the most popular board.

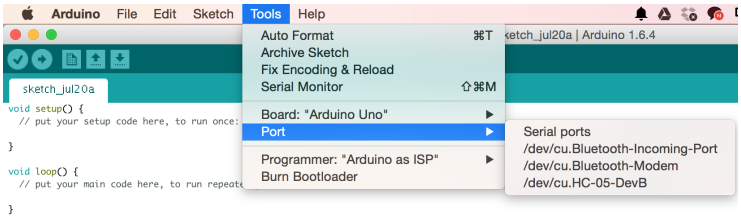


Figure 2: Correct serial port.

The second step is uploading the sample code "Blink" which is an example that will turn on and turn off the LED connect to pin 13 of any Arduino board. If the code is uploaded without error and the LED is blinking, it means the board is working properly and you can start your project.

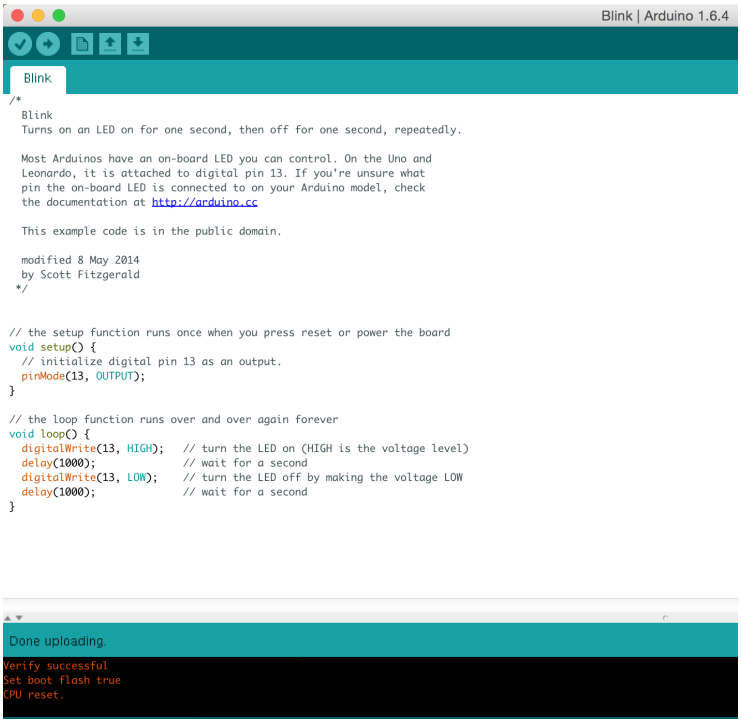


Figure 3: Done uploading

3 Inertial Measurement Unit

3.1 Definition

An inertial measurement unit (IMU) is an electronic device that measures and reports a craft's velocity, orientation, and gravitational forces, using a combination of accelerometers and gyroscopes, sometimes also magnetometers.

3.2 Accelerometer

An accelerometer is an electromechanical device used to measure acceleration forces. Such forces may be static, like the continuous force of gravity or, as is the case with many mobile devices, dynamic to sense movement or vibrations.

Accelerometer allows us to know if objects are moving and since the acceleration of an object is known, we also can determine speed and orientation. For example, 1g is equal to 9.81 m/s2.

3.3 Gyroscope

A gyroscope is a device that uses Earth's gravity to help determine orientation and maintains this level of effectiveness by being able to measure the rate of rotation around a particular axis. Gyroscope are strong used in altitude indicator on typical aircrafts.

4 IMU MPU-6050

MPU-6050 is a sensor that contains MEMS accelerometer and a MEMS gyroscope in one chip. Both accelerometer and gyrosocope contains 3 axis that can captures x,y and z with 16-bits analog to digital conversion hardware for each channel. Mpu-6050 uses I2C for communication which is a multi-master,multi-slave,single-ended,serial computer bus with low speed but very useful because uses only two wires: SCL(clock) and SDA(data) lines.

If we search on Arduino's website about this sensor, there are some examples and we are going to use the library i2cdevlib which comes with two examples: one getting raw values and another one using a Digital Motion Processor(DMP).I am using the example that uses DMP because of the complexity of the project.

The library "Wire.h" has the I2C commands and configuration. For example, the library says that the pins SCL and SDA have to be connect to the pins A5 and A4 respectively when the user uses Arduino Uno and Arduino ethernet. The other includes are part of the library created that was downloaded.

4.1 Schematic

To make the sensor MPU-6050 work, it is necessary:

- Arduino IDE;
- Arduino board;
- MPU-6050;
- Breadboard;
- Two pull-up resistor of 10k ohms;
- Wires;

Although the breadboard and the wires are optional items, the two pull-up resistor are essential. The diagram below shows how to connect the sensor to a Arduino Uno. It is also important connect all the sensor pins with the correct arduino pins. The pull-up resistor will always keep a small amount of current flowing between VCC and the pin, in other words, it will keep a valid logic level if it is not flowing current in the pin.

- Sensor VDD - Arduino 3.3v or 5v
- Sensor GND - Arduino GND
- Sensor INT - Arduino digital pin 2
- Sensor SCL - Arduino SCL dedicated pin = A5
- Sensor SDA - Arduino SDA dedicated pin = A4

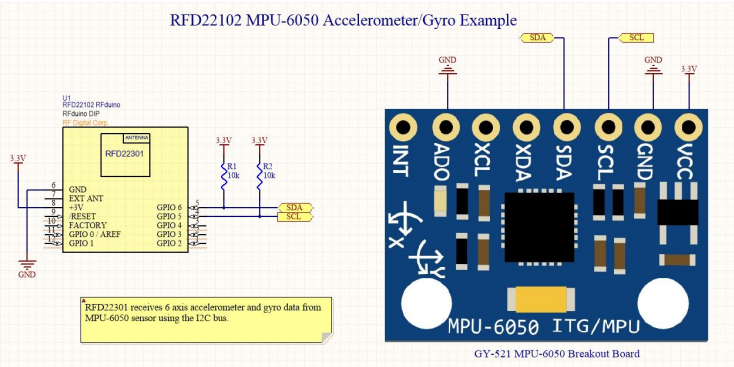


Figure 4: Source: RFduino, Image by Unknown

4.2 Code

The first thing to do is declare the libraries we are going to use. While the libraries i2c and Wire are responsible for the i2c communication, the library i2cdevlib is responsible for setting up all the registers and necessary configuration to make the sensor work.

```
#include <I2Cdev.h>
#include <MPU6050_6Axis_MotionApps20.h>
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
    #include <Wire.h>
#endif
```

The line below is configuring the I2C address. The MPU-6050 has two I2C addresses that makes possible to use two sensor at the same arduino board without an I2C multiplexer. The default address is 0x68 which means that the pin AD0 has to be connect to the ground.

```
MPU6050 mpu;
```

Those control variables are very important to make the MPU works and each one has a different function.

```
bool dmpReady = false;
uint8_t mpuIntStatus;
uint8_t devStatus;
uint16_t packetSize;
uint16_t fifoCount;
uint8_t fifoBuffer[64];
```

The meaning of each variable is:

- dmpReady is true if DMP initialization was successful.
- mpuIntStatus holds the actual interrupt status byte from MPU.
- devStatus returns the status after each device operation (0 = success, !0 = error).
- fifoCount is the DMP packet size(default is 42 bytes).
- fifoBuffer[64] is the FIFO storage buffer.

In the function setup() is important to initialize the library wire and set up the i2c speed. Every device has a maximum speed and this information is found in the datasheet. In our case, the maximum speed is 400KHz. Make sure the processor used is 16MHz, otherwise the speed has to be decreased to 200KHz.

```
  #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
        Wire.begin();
        TWBR = 24;
    #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
        Fastwire::setup(400, true);
    #endif
```

After initialize the i2c, it is time to start the MPU and test if the device is connect to the Arduino. The commands below are responsible for that:

```
 mpu.initialize();
 Serial.println(mpu.testConnection());
```

Note: The library used has a file called "MPU6050.cpp" which initializes the sensor. The following code is located in this file.

```
void MPU6050::initialize()
{
```

```
    setClockSource(MPU6050_CLOCK_PLL_XGYRO);
    setFullScaleGyroRange(MPU6050_GYRO_FS_250);
    setFullScaleAccelRange(MPU6050_ACCEL_FS_2);
    setSleepEnabled(false);
}
bool MPU6050::testConnection()
{
    return getDeviceID() == 0x34;
}
```

The code above is very simple and it is just setting the clock, gyroscope and accelerometer scales and disabling the sleep mode.

If the DMP initializes correctly, the devStatus will receive a status zero and the variable dmpReady will be set in one and this means that the DMP was initialized correctly and the program is ready the send data to the FIFO. Otherwise, it will be printed an error message and the variable dmpReady will continue "false" which makes impossible the executing of the code.

```
if (devStatus == 0)
{
        mpu.setDMPEnabled(true);
        attachInterrupt(0, dmpDataReady, RISING);
        mpuIntStatus = mpu.getIntStatus();
        dmpReady = true;
        packetSize = mpu.dmpGetFIFOPacketSize();
}
    else {
        Serial.print(devStatus);
        Serial.println("error");
    }
```

Now, the code is being executed in the function loop(). The first line is a conditional that tests if the DMP was initialize correctly. The examination is made through the variable dmpReady.

```
if (!dmpReady) return;
```

Until this part, the code was making sure that the device is connected to an Arduino and initializing the DMP. From now, the program will put the data into the FIFO and the FIFO will put the data out through the serial monitor.

The sample rate is specified by the register 25 and it is the speed that the sensor will send data to the FIFO. The sample is generated by dividing the gyroscope output rate by SMPLRTDIV: Sample Rate = Gyroscope Output Rate / (1 + SMPLRTDIV) where Gyroscope Output Rate = 8kHz when the DLPF is disabled (DLPFCFG = 0 or 7), and 1kHz when the DLPF is enabled.

SMPLRTDIV is a 8-bit unsigned value.

As expected, the FIFO has a buffer size and it is impossible knowing how to deal with it. The line below is getting the the number of bytes stored in the FIFO buffer. This number is in turn the number of bytes that can be read from the FIFO buffer and it is directly proportional to the number of samples available given the set of sensor data bound to be stored in the FIFO. The program will return the FIFO buffer size.

```
fifoCount = mpu.getFIFOCount()
```

The following code shows that the command "mpu.getFIFOCount()" comes from the file "MPU6050.cpp" that comes from the library.

```
uint16_t MPU6050::getFIFOCount()
{
    I2Cdev::readBytes(devAddr, MPU6050_RA_FIFO_COUNTH, 2, buffer);
    return (((uint16_t)buffer[0]) << 8) | buffer[1];
}
```

Since the moment the program knows the size of the FIFO, it is possible making a conditional that will delete the oldest data in the FIFO if an overflow happen. If not, the fifobuffer will be read and the data will be available.

The FIFO R-W register is the register is used to read and write data from the FIFO buffer. Data is written to the FIFO in order of register number (from lowest to highest). If all the FIFO enable flags (see below) are enabled and all External Sensor Data registers (Registers 73 to 96) are associated with a Slave device, the contents of registers 59 through 96 will be written in order at the Sample Rate.

The contents of the sensor data registers (Registers 59 to 96) are written into the FIFO buffer when their corresponding FIFO enable flags are set to 1 in FIFO-EN (Register 35). An additional flag for the sensor data registers associated with I2C Slave 3 can be found in I2C-MST-CTRL (Register 36).

If the FIFO buffer has overflowed, the status bit FIFO-OFLOW-INT is automatically set to 1. This bit is located in INT-STATUS (Register 58). When the FIFO buffer has overflowed, the oldest data will be lost and new data will be written to the FIFO.

If the FIFO buffer is empty, reading this register will return the last byte that was previously read from the FIFO until new data is available. The user should check FIFO-COUNT to ensure that the FIFO buffer is not read when empty.

```
if ((mpuIntStatus & 0x10) || fifoCount == 1024)
{
        mpu.resetFIFO();
}
```

```
else if (mpuIntStatus & 0x02)
{
        while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();
        mpu.getFIFOBytes(fifoBuffer, packetSize);
        fifoCount -= packetSize;

        #ifdef OUTPUT_READABLE_YAWPITCHROLL
            mpu.dmpGetQuaternion(&q, fifoBuffer);
            mpu.dmpGetGravity(&gravity, &q);
            mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
              Serial.print("ypr\t");
              Serial.print(ypr[0] * 180/M_PI);
              Serial.print("\t");
              Serial.print(ypr[1] * 180/M_PI);
              Serial.print("\t");
              Serial.println(ypr[2] * 180/M_PI);
        #endif
}
```

Note: To prevent FIFO overflow, do not execute the command delay(). It is also important to make sure that the sample rate has more or less the same speed than the FIFO output.

4.3 Data

More important than get the data from the sensor, it is get a reliable data. The code below is calibrating the sensor with the right offset of each axis: x,y and z. Every sensor has different offsets, so it is essential to find those values using another example that can be MPU6050.raw.ino that comes in the library file too.

```
mpu.setXGyroOffset(220);
mpu.setYGyroOffset(76);
mpu.setZGyroOffset(-85);
mpu.setZAccelOffset(1788);
```

The scale factor of accelerometers is calibrated at the factory and is nominally independent of supply voltage. Before we start using the MEMS we should calibrate using the code above because the raw values change a lot. After we get the offsets from the calibration, we have to write them in the code above. For reliable offsets, the device has to be placed on a flat surface.

```
// MPU-6050 Short Example Sketch
// By Arduino User JohnChi
// August 17, 2014
// Public Domain
```

```
#include<Wire.h>
const int MPU=0x68;
int16_t AcX,AcY,AcZ,Tmp,GyX,GyY,GyZ;
void setup()
{
  Wire.begin();
  Wire.beginTransmission(MPU);
  Wire.write(0x6B);
  Wire.write(0);
  Wire.endTransmission(true);
  Serial.begin(9600);
}
void loop(){
  Wire.beginTransmission(MPU);
  Wire.write(0x3B);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU,14,true);
  AcX=Wire.read()<<8|Wire.read();
  AcY=Wire.read()<<8|Wire.read();
  AcZ=Wire.read()<<8|Wire.read();
  Tmp=Wire.read()<<8|Wire.read();
  GyX=Wire.read()<<8|Wire.read();
  GyY=Wire.read()<<8|Wire.read();
  GyZ=Wire.read()<<8|Wire.read();
  Serial.print("AcX = "); Serial.print(AcX);
  Serial.print(" | AcY = "); Serial.print(AcY);
  Serial.print(" | AcZ = "); Serial.print(AcZ);
  Serial.print(" | Tmp = "); Serial.print(Tmp/340.00+36.53);
  Serial.print(" | GyX = "); Serial.print(GyX);
  Serial.print(" | GyY = "); Serial.print(GyY);
  Serial.print(" | GyZ = "); Serial.println(GyZ);
  delay(333);
}
```

Source: Arduino website by JohnChi.

4.3.1 Three-Axis MEMS Gyroscope

The full-scale range of the gyro sensors can be programmed to 250, 500, 1000, or 2000 degrees per second (dps) and it is 16-bit Analog-to-Digital Converters (ADCs) to sample each axis.

4.3.2 Three-Axis MEMS Accelerometer

The full scale range of the digital output can be adapted to 2g, 4g, 8g, or 16g. When the device is placed on a flat surface, it will measure 0g on the X- and Y-axes and +1g on the Z-axis.

4.3.3 DMP function

According to the website Greek Mom projects, "the MPU6050 IMU contains a DMP (Digital Motion Processor) which combines the accelerometer and gyroscope data together to minimize the effects of errors. The result is computed by the DMP in terms of quaternions and can convert the results to Euler angles and perform other computations with the data as well". The DMP is better than the complementary filter because is able to calculate Pitch, roll and yaw which are known as X,Y and Z axis (Euler angles). These calculations were limited by certain properties of both the accelerometer and gyroscope and one way to avoid the problems is to use an alternate method of representing rotation called quaternions. Quaternions describe rotation in three dimensions by using four scalar values. Three of these scalars define an axis, and the fourth specifies a rotation around that axis.

4.4 Register Map

Although the code above is explained in details, it is good to know a little bit more how the MPU6050 works when the topic is registers. This tutorial has a list with the main registers and the description of each one.

4.4.1 Register 25 - Sample Rate Divider - SMPRTDIV

This register specifies the divider from the gyroscope output rate used to generate the Sample Rate for the MPU-60X0. The sensor register output, FIFO output, DMP sampling and Motion detection are all based on the Sample Rate.

The Sample Rate is generated by dividing the gyroscope output rate by SMPLRTDIV: Sample Rate = Gyroscope Output Rate / (1 + SMPLRTDIV) where Gyroscope Output Rate = 8kHz when the DLPF is disabled (DLPFCFG = 0 or 7), and 1kHz when the DLPF is enabled.

Parameter:

- SMPLRTDIV is a 8-bit unsigned value.

Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
19	25	SMPLRT_DIV[7:0]							

Figure 5: Sample Rate - Source: Datasheet

4.4.2 Register 35 - FIFO Enable

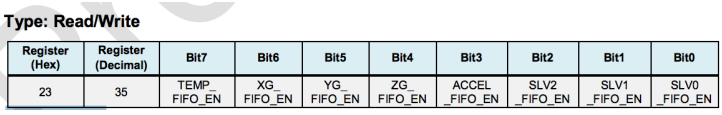
This register determines which sensor measurements are loaded into the FIFO buffer. Data stored inside the sensor data registers (Registers 59 to 96) will be

loaded into the FIFO buffer if a sensor?s respective FIFO-EN bit is set to 1 in this register.

When a sensor FIFO-EN bit is enabled in this register, data from the sensor data registers will be loaded into the FIFO buffer.

Parameters:

- TEMP-FIFO-EN - When set to 1, this bit enables TEMP-OUT-H and TEMP-OUT-L (Registers 65 and 66) to be written into the FIFO buffer.
- XG-FIFO-EN - When set to 1, this bit enables GYRO-XOUT-H and GYRO-XOUT-L (Registers 67 and 68) to be written into the FIFO buffer.
- YG-FIFO-EN - When set to 1, this bit enables GYRO-YOUT-H and GYRO-YOUT-L (Registers 69 and 70) to be written into the FIFO buffer.
- ZG-FIFO-EN - When set to 1, this bit enables GYRO-ZOUT-H and GYRO-ZOUT-L (Registers 71 and 72) to be written into the FIFO buffer.
- ACCEL-FIFO-EN - When set to 1, this bit enables ACCEL-XOUT-H, ACCEL-XOUT-L, ACCEL-YOUT-H, ACCEL-YOUT-L, ACCEL-ZOUT-H, and ACCEL-ZOUT-L (Registers 59 to 64) to be written into the FIFO buffer.
- SLV2-FIFO-EN- When set to 1, this bit enables EXT-SENS-DATAregisters (Registers 73 to96) associated with Slave 2 to be written into the FIFO buffer.
- SLV1- FIFO-EN - When set to 1, this bit enables EXT-SENS-DATAregisters (Registers 73 to 96) associated with Slave 1 to be written into the FIFO buffer.
- SLV0-FIFO-EN- When set to 1, this bit enables EXT-SENS-DATA registers (Registers 73 to 96) associated with Slave 0 to be written into the FIFO buffer.



Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
23	35	TEMP_ FIFO_EN	XG_ FIFO_EN	YG_ FIFO_EN	ZG_ FIFO_EN	ACCEL _FIFO_EN	SLV2 _FIFO_EN	SLV1 _FIFO_EN	SLV0 _FIFO_EN

Figure 6: FIFO enable - Source: Datasheet

4.4.3 Register 56 - Interrupt Enable

This register enables interrupt generation by interrupt sources.

Parameters:

- MOT-EN - When set to 1, this bit enables Motion detection to generate an interrupt.
- FIFO-OFLOW-EN - When set to 1, this bit enables a FIFO buffer overflow to generate an interrupt
- I2C-MST-INT-EN - When set to 1, this bit enables any of the I2C Master interrupt sources to generate an interrupt.
- DATA-RDY-EN - When set to 1, this bit enables the Data Ready interrupt, which occurs each time a write operation to all of the sensor registers has been completed.

Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
38	56		MOT_EN		FIFO _OFLOW _EN	I2C_MST _INT_EN	-	-	DATA _RDY_EN

Figure 7: Interrupt enable - Source: Datasheet

4.4.4 Register 58 - Interrupt Status

This register shows the interrupt status of each interrupt generation source. Each bit will clear after the register is read.

Parameters:

- MOT-INT - This bit automatically sets to 1 when a Motion Detection interrupt has been generated. The bit clears to 0 after the register has been read.
- FIFO-OFLOW-INT - This bit automatically sets to 1 when a FIFO buffer overflow interrupt has been generated. The bit clears to 0 after the register has been read.
- I2C-MST-INT - This bit automatically sets to 1 when an I2 C Master interrupt has been generated. For a list of I2C Master interrupts, please refer to Register 54. The bit clears to 0 after the register has been read.
- DATA-RDY-INT - This bit automatically sets to 1 when a Data Ready interrupt is generated. The bit clears to 0 after the register has been read.

Type: Read Only

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
3A	58	-	MOT_INT	-	FIFO _OFLOW _INT	I2C_MST _INT	-	-	DATA _RDY_INT

Figure 8: Interrupt Status - Source: Datasheet

4.4.5 Registers 59 to 64 - Accelerometer Measurements

These registers store the most recent accelerometer measurements.

Parameters:

- ACCEL-XOUT 16-bit 2?s complement value. Stores the most recent X axis accelerometer measurement.
- ACCEL-YOUT 16-bit 2?s complement value. Stores the most recent Y axis accelerometer measurement.
- ACCEL-ZOUT 16-bit 2?s complement value. Stores the most recent Z axis accelerometer measurement.

Type: Read Only

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
3B	59	ACCEL_XOUT[15:8]							
3C	60	ACCEL_XOUT[7:0]							
3D	61	ACCEL_YOUT[15:8]							
3E	62	ACCEL_YOUT[7:0]							
3F	63	ACCEL_ZOUT[15:8]							
40	64	ACCEL_ZOUT[7:0]							

Figure 9: Accelerometer Measurements - Source: Datasheet

4.4.6 Registers 65 and 66 - Temperature Measurement

These registers store the most recent temperature sensor measurement. The temperature in degrees C for a given register value may be computed as: Temperature in degrees C = (TEMP-OUT Register Value as a signed quantity)/340 + 36.53.

Parameters:

- TEMP-OUT 16-bit signed value. Stores the most recent temperature sensor measurement.

Type: Read Only

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
41	65	TEMP_OUT[15:8]							
42	66	TEMP_OUT[7:0]							

Figure 10: Temperature Measurement - Source: Datasheet

4.4.7 Registers 67 to 72 - Gyroscope Measurements

These registers store the most recent gyroscope measurements.

Parameters:

- GYRO-XOUT 16-bit 2?s complement value. Stores the most recent X axis gyroscope measurement.
- GYRO-YOUT 16-bit 2?s complement value. Stores the most recent Y axis gyroscope measurement.
- GYRO-ZOUT 16-bit 2?s complement value. Stores the most recent Z axis gyroscope measurement.

Type: Read Only

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
43	67	GYRO_XOUT[15:8]							
44	68	GYRO_XOUT[7:0]							
45	69	GYRO_YOUT[15:8]							
46	70	GYRO_YOUT[7:0]							
47	71	GYRO_ZOUT[15:8]							
48	72	GYRO_ZOUT[7:0]							

Figure 11: Gyroscope Measurement - Source: Datasheet

4.4.8 Registers 73 to 96 - External Sensor Data

These registers store data read from external sensors by the Slave 0, 1, 2, and 3 on the auxiliary I2C interface.

External sensor data is written to these registers at the Sample Rate as defined in Register 25. External sensor data registers, along with the gyroscope measurement registers, accelerometer measurement registers, and temperature measurement registers, are composed of two sets of registers: an internal register set and a user-facing read register set.

The data within the external sensors? internal register set is always updated at the Sample Rate (or the reduced access rate) whenever the serial interface is idle. This guarantees that a burst read of sensor registers will read measurements from the same sampling instant. Note that if burst reads are not used, the user is responsible for ensuring a set of single byte reads correspond to a single sampling instant by checking the Data Ready interrupt.

Type: Read Only

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
49	73	EXT_SENS_DATA_00[7:0]							
4A	74	EXT_SENS_DATA_01[7:0]							
4B	75	EXT_SENS_DATA_02[7:0]							
4C	76	EXT_SENS_DATA_03[7:0]							
4D	77	EXT_SENS_DATA_04[7:0]							
4E	78	EXT_SENS_DATA_05[7:0]							
4F	79	EXT_SENS_DATA_06[7:0]							
50	80	EXT_SENS_DATA_07[7:0]							
51	81	EXT_SENS_DATA_08[7:0]							
52	82	EXT_SENS_DATA_09[7:0]							
53	83	EXT_SENS_DATA_10[7:0]							
54	84	EXT_SENS_DATA_11[7:0]							
55	85	EXT_SENS_DATA_12[7:0]							
56	86	EXT_SENS_DATA_13[7:0]							
57	87	EXT_SENS_DATA_14[7:0]							
58	88	EXT_SENS_DATA_15[7:0]							
59	89	EXT_SENS_DATA_16[7:0]							
5A	90	EXT_SENS_DATA_17[7:0]							
5B	91	EXT_SENS_DATA_18[7:0]							
5C	92	EXT_SENS_DATA_19[7:0]							
5D	93	EXT_SENS_DATA_20[7:0]							
5E	94	EXT_SENS_DATA_21[7:0]							
5F	95	EXT_SENS_DATA_22[7:0]							
60	96	EXT_SENS_DATA_23[7:0]							

Figure 12: External Sensor Data - Source: Datasheet

5 nRF24L01

The nRF24L01 is a Radio/Wireless Transceiver module which is able to communicate two or more Arduinos over a distance and it is constantly used for remote sensor, Robot control and monitoring from 50 feet to 2000 feet distances, but this distance can change according to the environment because of walls and materials. There are modules that it is possible to buy and those modules such as, Transmitters power amplifiers and Receivers preamplifiers, permit transmitting in longer distances.

The nRF24L01 supports the high-speed Serial Peripheral Interface(SPI) and it is still low power consumption. Sometimes the low current necessary can cause some power problems, so it is recommended add a 0.1uF or 10uF capacitor between the GND and 3.3V pin to guarantee current to module. The capacitor recommended will serve as a source of energy, in other words, as a battery.

The nRF24L01 operates at 250KHz, 1Mhz and 2Mhz and it is suggested use the lower speed to make sure if the data sent and received are reliable. It is also important to consider those detailed specially because of the different suppliers.

Later on, we are going to talk about Multiceivers which allows 6 Arduinos to talk to a Primary Arduino in an organized manner.

Figure 13: nRF24L01

5.1 Schematic

To make the module nRF24L01 work, it is necessary:

- Arduino IDE;
- Arduino board;
- nRF24L01;
- One 0.1uF or 10uF capacitor;
- Wires;

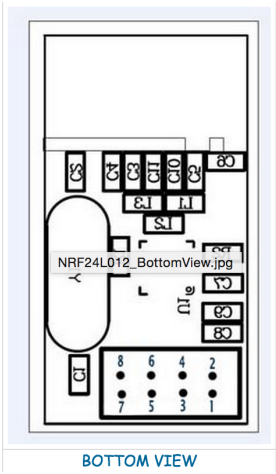


Figure 14: Bottom view. Source: Arduino Info

Signal	RF Module	COLOR	Arduino pin for RF24 Library	Arduino pin for Mirf Library	MEGA2560 pin RF24 Library	Arduino Pin for RH_NRF24 RadioHead Library	MEGA2560 Pin for RH_NRF24 RadioHead Library
GND	1	Brown	GND *	GND	GND *	GND *	GND *
VCC	2	Red	3.3V *	3.3V	3.3V *	3.3V *	3.3V *
CE	3	Orange	9	8	9	8	8
CSN	4	Yellow	10	7	53	10	53
SCK	5	Green	13	13	52	13	52
MOSI	6	Blue	11	11	51	11	51
MISO	7	Violet	12	12	50	12	50
IRQ	8	Gray	2 *		per library	N/C	N/C

Figure 15: Pinout. Source: Arduino Info

The images below will show how to set the modules nRF24L01 up.

The picture above shows that is necessary 8 pins to connect the module to a Arduino. The place of connection of those pins is different from one Arduino to another. The pins "CE" and "CSN" can change by programming.

In our example, we are using the following configuration.

- Module VDD - Arduino 3.3v
- Module GND - Arduino GND
- Module IRQ - not used

- Module CE - Arduino CE dedicated pin = 9
- Module CSN - Arduino CSN dedicated pin = 10
- Module SCK - Arduino SCK dedicated pin = 13
- Module MOSI - Arduino MOSI dedicated pin = 11
- Module MISO - Arduino MISO dedicated pin = 12

Note: The pinout of the transmitter module and receiver module are the same.

5.2 Code

The first thing to take a look in this part is about the library that we are going to use. The Arduino website talks about two libraries: RF24 and Mirf. In this code we will use the first library which has the same configuration above.

Remember to declare the libraries used at the first part of the code right in the top. The following libraries are necessary to communicate with the nRF2401 and with the SPI.

```
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>
```

After the declaration of the libraries, we have the opportunity of modify the pin numbers of CE and CSN which will be used later to initialize the module or radio. According to the datasheet, the pin CE is used to active or standby the mode. The pin CSN is used to tell the nRF24 whether the SPI communication is a command or message to send out as claimed by the same datasheet.

```
#define CE_PIN 9
#define CSN_PIN 10
```

Until here, this part is the same for both modes: the transmitter and receiver. Now, we will explain how to write a transmitter code. This Wireless module send and receive data through pipes, in other words, a pipe is an transceiver address. The lines below has to be programmed in both modules.

```
const uint64_t pipe = 0xE8E8F0F0E1LL;
RF24 radio(CE_PIN, CSN_PIN);
```

The first difference starts in the function setup() which will start the communication with the command "radio.begin()" and open the only pipe for writing because we are programming the transmitter part.

```
radio.begin();
radio.openWritingPipe(pipe);
```

Then, the next function is the loop() which will be very simple too. We just have to write the data we want to send to the receiver. In this example, we are sending an array of three elements.

```
int test[3];
test[0] = 10;
test[1] = 20;
test[2] = 30;
radio.write( test, sizeof(test));
```

Now, we will se how to program the function setup() of the receiver mode.

```
radio.openReadingPipe(1,pipe);
 radio.startListening();;
```

The nRF24L01 supports six pipes for reading, so it is important to define each one we will use. In this example, the pipe 1 is the first. Before reading, we must use the function startListening();

In the function loop(), we will verify if there is a connection, then the program will be ready to receive the 3 elements we are waiting.

```
int text[3];
if ( radio.available() )
 {
 bool done = false;
   while (!done)
   {
     done = radio.read( text, sizeof(text) );
     Serial.print("Number: ");
     Serial.print(text[0]);
      Serial.print("Number: ");
     Serial.print(text[1]);
      Serial.print("Number: ");
     Serial.print(text[2]);
   }
  }
  else
  {
      Serial.println("No radio available");
  }
 }
```

The entire Transmitter code is

```
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

#define CE_PIN 9
#define CSN_PIN 10

const uint64_t pipe = 0xE8E8F0F0E1LL; // Define the transmit pipe

RF24 radio(CE_PIN, CSN_PIN); // Create a Radio
int test[3];
void setup()
{
  radio.begin();
  radio.openWritingPipe(pipe);
}

void loop()
{
  test[0] = 10;
  test[1] = 20;
  test[2] = 30;
  radio.write( test, sizeof(test));
}
```

The entire Receiver code is

```
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

#define CE_PIN 9
#define CSN_PIN 10

const uint64_t pipe = 0xE8E8F0F0E1LL;

RF24 radio(CE_PIN, CSN_PIN);
int text[3];
void setup()
{
 radio.openReadingPipe(1,pipe);
 radio.startListening();;
}

void loop()
```

```
{
   if ( radio.available() )
   {
      bool done = false;
      while (!done)
      {
        done = radio.read( text, sizeof(text) );
        Serial.print("Number: ");
        Serial.print(text[0]);
        Serial.print("Number: ");
        Serial.print(text[1]);
        Serial.print("Number: ");
        Serial.print(text[2]);
      }
   }
   else
   {
      Serial.println("No radio available");
   }
 }
```

5.3 Multiceiver

MultiCeiver is a feature used in RX mode that contains a set of six parallel data pipes with unique address. A data pipe is a logical channel in the physical RF channel. Each data pipe has its own physical address decoding in the NRF24L01+.

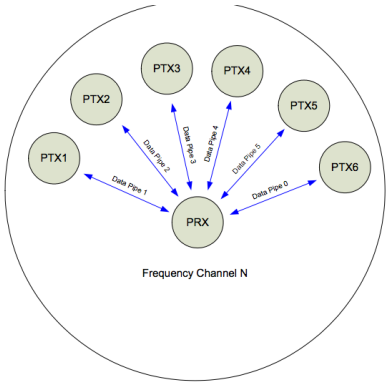


Figure 16: MultiCeiver schematic. Source: Datasheet

Both modes have differents codes from the first and second example one showed. **********Arduino Code for Receiver**********

```
include <SPI.h>
include <nRF24L01.h>
include <RF24.h>

const int pinCE = 9;
const int pinCSN = 10;
RF24 radio(pinCE, pinCSN);
const uint64_t rAddress[] = {0xB00B1E50D2LL, 0xB00B1E50C3LL};
int number = 0;
void setup()
{
  Serial.begin(57600);
  radio.begin();
  radio.openReadingPipe(1,rAddress[0]);
  radio.openReadingPipe(2,rAddress[1]);
  radio.startListening();
}

void loop()
{
    byte pipe = 0;

    while(radio.available(&pipe))
    {
      radio.read( &number, sizeof(number));
      Serial.print("Transmitter number ");
      Serial.println(pipe);
      Serial.print("Number: ");
      Serial.println(number);
       Serial.println();
    }
}
```

Source: ForceTronic by Neil ForceTronic.

**********Arduino Code for Transmitter 1**********

```
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

const int pinCE = 9;
const int pinCSN = 10;

bool done = false;
RF24 radio(pinCE, pinCSN);
const uint64_t wAddress = 0xB00B1E50D2LL;
int number = 10;
void setup()
```

```
{
   Serial.begin(57600);
   radio.begin();
   radio.openWritingPipe(wAddress);
   radio.stopListening();
}


void loop()
{
   if(!done)
   {
        if (!radio.write( &number, sizeof(number) ))
        {
             Serial.println("Sending failed");
          }
        else
        {
            Serial.print("Success sending: ");
            Serial.println(number);
         }
   }
}
```

Source: ForceTronic by Neil ForceTronic.

**********Arduino Code for Transmitter 2**********

```
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

const int pinCE = 9;
const int pinCSN = 10;

bool done = false;
RF24 radio(pinCE, pinCSN);
const uint64_t wAddress = 0xB00B1E50C3LL;
int number = 20;
void setup()
{
   Serial.begin(57600);
   radio.begin();
   radio.openWritingPipe(wAddress);
   radio.stopListening();
}


void loop()
```

```
{
        if(!radio.write( &number, sizeof(number) ))
        {
             Serial.println("Sending failed");
          }
        else
        {
            Serial.print("Success sending ");
            Serial.println(number);
         }

}
```

Source: ForceTronic by Neil ForceTronic.

It is possible to use more four transmitter together. The only thing to care is the addresses and the data that the new transmitter would send. All addresses must be different and the RX mode has to receive the same data that the TX mode are sending. If we want more than two transmitter we can use the same code, but instead of the same address we have to modify for the following addresses below.

```
\item const uint64_t wAddress = 0xB00B1E50B4LL;
\item const uint64_t wAddress = 0xB00B1E50A5LL;
 \item const uint64_t wAddress = 0xB00B1E5096LL;
 \item const uint64_t wAddress = 0xB00B1E5087LL;
```

The data sent should change according to the necessity.

6 Integration

The aim of this tutorial is learn about two devices: MPU-6050 and nRF24L01. After understanding that, we chose to integrate both, getting the data from the sensor and sending through the Wireless module. We had some problems specially because the sensor gets information using the DMP that only works without delay, but the library used has a small delay in one of the functions. After a lot of tests, we know that the devices are working well separately, through they don't work reliably together.

The ready - functions from the RF24 library were analysed too since we thought the problem could be there. Some of the functions inside the library have delay. Those delays are responsible for sending and receiving data, which can overflow the MPU - 6050 FIFO because the data output is becoming slower than the data input.

One of the possible solutions was decrease the non - crucial delays to see the response. Sadly, after some minutes the program crashed again. We also tried to run two loops that we could initialize the MPU-6050 and nRF24L01 after

the counter achieves some number. The answer was good, however, the data got was not reliable because every time the sensor was initiazed, the sensor was calibrating for a few seconds.

7 References

[1] Arduino Official.

[2] MPU-6050 Datasheet.

[3] Gyroscopes and Accelerometers.

[4] DMP data.

[5] Nrf24L01-2.4GHz-Arduino Official.

[6] Nrf24L01-2.4GHz Datasheet.

[7] Nrf24L01-2.4GHz-HowToWork.

[8] GithubRF24.

[9] Multiceiver.