1 Arduino Data Types

'byte' needs one byte of memory.
'int' needs two bytes of memory.
Number 'type's.

- *boolean* (8 bit) - simple logical true/false, Arduino does not use single bits for bool
- *byte* (8 bit) - unsigned number from 0 to 255
- *char* (8 bit) - signed number from -128 to 127. The compiler will attempt to interpret this data type as a character in some circumstances, which may yield unexpected results
- *unsigned char* (8 bit) - same as 'byte'; if this is what you're after, you should use 'byte' instead, for reasons of clarity word (16 bit) - unsigned number from 0 to 65535
- *unsigned int* (16 bit)- the same as 'word'. Use 'word' instead for clarity and brevity
- *int* (16 bit) - signed number from -32768 to 32767. This is most commonly what you see used for general purpose variables in Arduino example code provided with the IDE
- *unsigned long* (32 bit) - unsigned number from 0 to 4,294,967,295. The most common usage of this is to store the result of the *millis()* function, which returns the number of milliseconds the current code has been running
- *long* (32 bit) - signed number from -2,147,483,648 to 2,147,483,647
- *float* (32 bit) - signed number from -3.4028235E38 to 3.4028235E38. Floating point on the Arduino is not native; the compiler has to jump through hoops to make it work. If you can avoid it, you should. We'll touch on this later. Sparkfun.

You select the 'type' best suited for your variables.
ex:

- your variable does not change and it defines a pin on the Arduino. const byte limitSwitchPin = 34;
- since an analog variable can be 0 to 1023, a byte will not do, you can select 'int'. int temperature;
- if your variable needs to be within -64 to +64 a 'char' will do nicely. *char joystick;*

- if your variable is used for ASCII then you need type 'char', *char myText[] = "Raspberry Pie Smells";*
- if your variable enables some code then boolean can be used. boolean enableFlag = false;
- *millis()* returns the time in ms since rebooting, *unsigned long currentTime = millis();* etc.

2 Volatile

volatile is a keyword known as a variable qualifier, it is usually used before the datatype of a variable, to modify the way in which the compiler and subsequent program treat the variable.

Declaring a variable volatile is a directive to the compiler. The compiler is software which translates your C/C++ code into the machine code, which are the real instructions for the Atmega chip in the Arduino.

Specifically, it directs the compiler to load the variable from RAM and not from a storage register, which is a temporary memory location where program variables are stored and manipulated. Under certain conditions, the value for a variable stored in registers can be inaccurate.

A variable should be declared volatile whenever its value can be changed by something beyond the control of the code section in which it appears, such as a concurrently executing thread. In the Arduino, the only place that this is likely to occur is in sections of code associated with interrupts, called an interrupt service routine.

3 Brown Out Reset

Brown Out Reset is an important function to increase the reliability of a microcontroller after start-up. Normally used to solve problems with the power supply. A "brown out" of a microcontroller is a partial and temporary reduction in the power supply voltage below the level required for reliable operation. Many microcontrollers have a protection circuit which detects when the supply voltage goes below this level and puts the device into a reset state to ensure proper startup when power returns. This action is called a "Brown Out Reset" or BOR. A similar feature is called Low Voltage Detect (LVD) which is more complex and adds detection of multiple voltage levels and can produce an interrupt before a reset is triggered.

4 Watchdog System Reset

A watchdog is an independent timer that monitors the progress of the main controller running Data ONTAP. Its function is to serve as an automatic server restart in the event the system encounters an unrecoverable system error.

5 If you see DDR, PORT or PIN its about Port Manipulation

There are three registers for Port Manipulation.

- DDR: to tell whether pin is an input or output, can r/w.

  Eg., DDRB = 0b00000101 means D8 and D10 are declared as OUTPUTs and rest all inputs. Moreover, Atmega pins defauls to inputs, so one does not need to explicitly declare the inputs.

- PORT: it registers whether the pin is a HIGH or LOW. Find more in section 6, can r/w.

  Eg., PORTD = B01100000 to set digital pins 5 and 6 as HIGH. Before using it you first need to declare those pins as OUTPUT: DDRD = B01100000. If you observe, 0bx and Bx can be used interchangeably.

- PIN: to read the state of the pins which are defined. Used in ISR. can r only.

  Eg., PINB reads the state of D8-D13. Lets suppose if D8 is HIGH, PINB returns B00000001.

  if (PINB & B00000001) means if D8 is HIGH.

6 PORTs

PORT: it registers whether the pin is a HIGH or LOW
PORTB : (PCINT0 – PCINT7): D8-D13; The two high bits D14 & D15 map to crystal pins and are not usable.
PORTC : (PCINT8 - PCINT13): A0-A5; Analog pins
PORTD : ((PCINT16 – PCINT23): D0-D7; Digital pins

7 Interrupts

https://dronebotworkshop.com/interrupts/

7.1 PCICR

Refer *https://thewanderingengineer.com/2014/08/11/arduino-pin-change-interrupts/*
https://www.teachmemicro.com/arduino-interrupt-tutorial/

8 Flash Memory String

What is F below?

```
1 Serial.println(F("This is a constant string."));
```

F() is a macro. This helps save the SRAM(Static RAM) memory by moving the constant string from SRAM to FLASH memory. Usually, FLASH memory is much more available than SRAM memory. Let's see two examples.

Without F()

```
1
2  void setup() {
3    Serial.begin(9600);
4
5    Serial.println("ArduinoGetStarted.com");
6  }
7
8  void loop() {
9
10 }
11 .
12 .
13 .
14 /*
15 Compile Log:
16 Sketch uses 1496 bytes (4%) of program storage space. Maximum is
       32256 bytes.
17 Global variables use 210 bytes (10%) of dynamic memory, leaving
       1838 bytes for local variables. Maximum is 2048 bytes.*/
```

With F()

```
1
2  void setup() {
3    Serial.begin(9600);
4
5    Serial.println(F("ArduinoGetStarted.com"));
6  }
7
8  void loop() {
9  }
10 .
11 .
12 .
13 /*
14 Compile Log:
15 Sketch uses 1506 bytes (4%) of program storage space. Maximum is
       32256 bytes.
16 Global variables use 188 bytes (9%) of dynamic memory, leaving 1860
        bytes for local variables. Maximum is 2048 bytes.*/
```

9 TWI - Two Wire Interface and I2C

AVR (ATmega32) contains some in-built registers for TWI communication which not only reduce the level of complexity but also make the whole communication process smooth. TWBR: TW Bit Rate Register This register is used in master mode to set the division factor for bit rate generator (SCL clock frequency). The bit rate generator unit controls the time period of SCL. The SCL clock

frequency is decided by the Bit Rate Register (TWBR) and the prescaler bits of TWSR register.

9.1 Setting I2C Clock Speed

There are two ways:

- TWBR: TW Bit Register In this method, by setting TWBR to any value between 0 to 255, we can get the clock speed. The *F-CPU* define only tells the compiler what CPU speed to use for various delays and functions. We can set the desired frequency as

```
frequency = (2*TWBR + 16)/F_CPU;
/*Here F_CPU frequency is 16MHz*/
// to obtain a frequency of 400kHz, we will need TWBR of
TWBR = ( (F_CPU/frequency) - 16)/2;
// which is equal to 12
```

- Using Wire.h

```
Wire.SetClock{frequency}
```

Refer Engineers Garage: AVR ATmega32 TWI Registers
Refer *https://www.exploreembedded.com/wiki/Basics of I2C with AVR*
https://arduino.stackexchange.com/questions/29583/difference-between-wire-setclock-method-and-twbr-method-for-changing-i2c-frequ

10 Wire

This library allows you to communicate with I2C/TWI devices.

The MCU brings START condition on the bus, checks the presence of the slave on the bus by sending its 7-bit address through the execution of this code: Wire.beginTransmission(slaveAddress)

Wire.write(0x25); instruction stores/queues the data byte (0x25) in a buffer no in slave.

At the Master side, the Wire.endTransmission(); tells the MCU to get the data byte from the buffer and send it to slave using I2C bus/protocol and then brings STOP condition on the bus.

Finally, the sequence of codes are:

```
Wire.beginTransmission(slaveAddress);
Wire.write(0x25);
Wire.endTransmission();
```

10.1 Wire.requestFrom()

This function is used by the controller device to request bytes from a peripheral device. The bytes may then be retrieved with the *available*() and *read*() functions. As of Arduino 1.0.1, *requestFrom*() accepts a boolean argument changing its behavior for compatibility with certain I2C devices. If true, *requestFrom*() sends a stop message after the request, releasing the I2C bus. If false, *requestFrom*() sends a restart message after the request. The bus will not be released, which prevents another master device from requesting between messages. This allows one master device to send multiple requests while in control. The default value is true.

Syntax

Wire.requestFrom(*address*, *quantity*)

Wire.requestFrom(*address*, *quantity*, *stop*)

Parameters

address: the 7-bit slave address of the device to request bytes from.

quantity: the number of bytes to request.

stop: true or false. true will send a stop message after the request, releasing the bus. False will continually send a restart after the request, keeping the connection active.

Returns byte : the number of bytes returned from the peripheral device.

11 Breaking the setup code

You need to run this code after assembling the hardware. The goal of this code is to note down the necessary data for the variables into EEPROM for the compilation of other two programs. It includes:

- Check if the I2C clock speed is set to 400kHz
- Check whether the receiver signals are valid i.e., they lie within the range of [900,2100]
- Store the center positions of all the sticks
- Noting that there are 4 ways we can move the stick in the transmitter. In this check we start with noting down the stick for throttle control and its direction
- Noting down the roll stick and its direction (which way positive and which way negative)
- "" pitch sticks
- "" yaw sticks
- Note the max and min of all the channels
- Check for the gyro options (In our case: MP06050)

- Now setup the gyro axis of rotation
- If all done, store the information in the EEPROM

Important:

- D8 >> Channel 1 (C1) >> roll
- D9 >> C2 >> pitch
- D10 >> C3 >> throttle
- D11 >> C4 >> yaw

Assigning the global variables

```
#include <Wire.h> // so we can communicate with the gyro
#include <EEPROM.h> // so we can store information into the EEPROM

// Declaring Global Variables
// byte is used to declare pins
// byte is a keyword that is used to tell the compiler to reserve 1-memory location
// for 1 byte size when the number has a range 0 - 255 or 0x00 - 0xFF.
// it is similar to "unsigned char" in C++; or uint8_t
byte last_channel_1, last_channel_2, last_channel_3, last_channel_4;
byte lowByte, highByte, type, gyroaddress, error, clockspeed_ok;
byte channel_1_assign, channel_2_assign, channel_3_assign, channel_4_assign;
byte roll_axis, pitch_axis, yaw_axis;
byte receiver_check_byte, gyro_check_byte;

// gets stored in RAM
volatile int receiver_input_channel_1, receiver_input_channel_2, receiver_input_channel_3, receiver_input_channel_4;

int center_channel_1, center_channel_2, center_channel_3, center_channel_4;
int high_channel_1, high_channel_2, high_channel_3, high_channel_4;
int low_channel_1, low_channel_2, low_channel_3, low_channel_4;
int address, cal_int;

// time is stored in unsigned long
unsigned long timer, timer_1, timer_2, timer_3, timer_4, current_time;
float gyro_pitch, gyro_roll, gyro_yaw;
float gyro_roll_cal, gyro_pitch_cal, gyro_yaw_cal;
```

```
void wait_for_receiver(){
  byte zero = 0;
  timer = millis() + 10000;
  while(timer > millis() && zero < 15){ // until 10 seconds && all the inputs are checked.
    if (receiver_input_channel_1 < 2100 && receiver_input_channel_1 > 900)zero |= 0b00000001;
```

```
    // now zero is 0b00000001
    if (receiver_input_channel_2 < 2100 && receiver_input_channel_2 > 900)zero |= 0b00000010;
    // zero = zero | 0b00000010, so now zero is 0b00000011
    if (receiver_input_channel_3 < 2100 && receiver_input_channel_3 > 900)zero |= 0b00000100;
    // zero = zero | 0b00000100, so now zero is 0b00000111
    if (receiver_input_channel_4 < 2100 && receiver_input_channel_4 > 900)zero |= 0b00001000;
    // zero = zero | 0b00001000, so now zero is 0b00001111 which is 15. HAHA!
    // This means, until 10 seconds AND until all the receiver inputs: throttle, yaw,
    // roll and pitch are checked whether they are valid or not (lie within 900 to 2100)
    // this while loop runs.
    delay(500);
    Serial.print(F("."));
  }
  if (zero == 0){
    error = 1;
    Serial.println(F("."));
    Serial.println(F("No valid receiver signals found!!! (ERROR 1)"));
  }
  else Serial.println(F("RECEIVERS OK"));
}
```

12 Little about Gyro

Brief Summary:

- The IMU_{gyro} is less sensitive to vibrations: Gyro brings short term advantage - The IMU_{gyro} drifts a little over time - Vibrations make the IMU_{accel} useless - Only the average of IMU_{accel} is usable. The accel brings long term advantage.

Final Question: How do we create a usable value with the accelerometer that can be used to compensate the gyro's drift? This is where complementary filter comes into picture.

13 EEPROM Bytes store

```
    EEPROM.write(0, center_channel_1 & 0b11111111);
    EEPROM.write(1, center_channel_1 >> 8);
    EEPROM.write(2, center_channel_2 & 0b11111111);
    EEPROM.write(3, center_channel_2 >> 8);
    EEPROM.write(4, center_channel_3 & 0b11111111);
    EEPROM.write(5, center_channel_3 >> 8);
    EEPROM.write(6, center_channel_4 & 0b11111111);
    EEPROM.write(7, center_channel_4 >> 8);
    EEPROM.write(8, high_channel_1 & 0b11111111);
    EEPROM.write(9, high_channel_1 >> 8);
```

```
11      EEPROM.write(10, high_channel_2 & 0b11111111);
12      EEPROM.write(11, high_channel_2 >> 8);
13      EEPROM.write(12, high_channel_3 & 0b11111111);
14      EEPROM.write(13, high_channel_3 >> 8);
15      EEPROM.write(14, high_channel_4 & 0b11111111);
16      EEPROM.write(15, high_channel_4 >> 8);
17      EEPROM.write(16, low_channel_1 & 0b11111111);
18      EEPROM.write(17, low_channel_1 >> 8);
19      EEPROM.write(18, low_channel_2 & 0b11111111);
20      EEPROM.write(19, low_channel_2 >> 8);
21      EEPROM.write(20, low_channel_3 & 0b11111111);
22      EEPROM.write(21, low_channel_3 >> 8);
23      EEPROM.write(22, low_channel_4 & 0b11111111);
24      EEPROM.write(23, low_channel_4 >> 8);
25      EEPROM.write(24, channel_1_assign);
26      EEPROM.write(25, channel_2_assign);
27      EEPROM.write(26, channel_3_assign);
28      EEPROM.write(27, channel_4_assign);
29      EEPROM.write(28, roll_axis);
30      EEPROM.write(29, pitch_axis);
31      EEPROM.write(30, yaw_axis);
32      EEPROM.write(31, type);
33      EEPROM.write(32, gyro_address);
```