ControlluinoII Project Documentation v1.0

Mihalis Tsiakkas

March 29, 2013

Contents

1 Introduction **3**

2 Hardware **4**
2.1 Arduino 4
2.2 XBee 4
2.3 XSens MTi-G 5
2.4 Custom Arduino Shield 5

3 Software **7**
3.1 Arduino 7
3.2 PC Interface 8
3.3 Communications Protocol 9

A Circuit Diagrams & PCB Layouts **13**

B Datasheets **14**

C Example Messages **15**

D Interface Screenshots **17**

1 Introduction

The *Controlluino Project* is an attempt to simplify the development process for UAV control systems, based on the open source Arduino platform. The main components of the project can be listed as follows:

- Hardware
  - Arduino DUE
  - XBee PRO S2B
  - XSens MTi-G INS
  - Custom-built Arduino shield
- Software
  - Arduino program
  - Interface application

The current setup supports up to 6 motors, BLDC or Servo, although in terms of hardware many more can be added. During the development process a trirotor UAV with tilting rotors was kept in mind, thus the motors are configured as 3 BLDC and 3 servos.

Two XBee Pro S2B modules are used for wireless communications between the Arduino board and the host computer.

Attitude and position data are obtained using the XSens MTi-G INS, which incorporates gyroscopes, accelerometers, magnetometers, a barometer and a GPS module to provide all the measurements need for both attitude and position control.

2 Hardware

2.1 Arduino (`http://www.arduino.cc`)

The Controlluino project was developed around the Arduino Due, which runs on an 84MHz 32-bit ARM Cortex-M3 μC from ATmel. Some of the board's features are:

- 4 Serial communications ports (UART), of which one is normally used by the onboard USB socket.
- 15 PWM output channels
- 16 Analog-to-Digital converters (input)
- 2 Digital-to-Analog converters for truly analog output
- SPI bus

The Arduino is programmed in C++, but libraries are provided to simplify the development process. For example, functions are available to communicate through a serial port using a single line of code, which would normally be much more involved. This allows for faster software development and easier to read code.

Depending on the scenario (attitude/position control, full output etc.) sampling times of as low as 20 ms can be achieved using the current setup.

2.2 XBee (`http://www.digi.com/xbee/`)

The XBee is a simple to use wireless communications module. Communications between two devices can be achieved using two modules configured to "talk" to each other. A third party programmer is used to configure the two XBees, through the X-CTU software available from Digi, and to let the one communicate with the host PC during operation.

There are two version of the XBee currently available in the lab. These are the XBee S1 and the much more powerful XBee Pro S2B. While the S1 is very simple to use, with minimal setup required, it can only achieve communication speeds of 57K6BAUD at a range of 100m (30m indoor). On the other hand, the XBee Pro S2B requires a small amount of configuration before being used but can provide communication speeds of up to 115K2BAUD and a range of 1500m (90m indoor).

The two versions are pin-for-pin compatible so they can be changed online, however, they are unable to communicate between them.

2.3 XSens MTi-G (`http://www.xsens.com/en/general/mti-g`)

The MTi-G is a comprehensive INS that includes all instruments required for the navigation of a vehicle. It is able to provide a multitude of measurements including position and velocity both linear and angular. It communicates using the RS-232 serial protocol. The "messages" used for communicating with the sensor are very well documented in the *MT Low-Level Communication Protocol Documentation.* Chapters 4, 5 and 6 of *MTi-G User Manual and Technical Documentation* are a suggested read before using the sensor.

2.4 Custom Arduino Shield

A PCB was designed to allow the Arduino board to communicate with with all the different components of the project.

The first part is a power converter unit which utilises a 12V switching DC-DC converter and a 9V voltage regulator. The 2-stage conversion was required to reduce heat dissipation. The switching converter used has an enable pin; the converter only operated when this pin is pulled to ground. This gives the Arduino the ability to cut power at will. The Arduino and the INS are powered directly from the 9V supply while the XBee is powered from the Arduino's 3.3V output.

Since the sensor uses the RS-232 protocol for communications, a line driver is needed. The MAX3232 was selected to interface the Arduino with the sensor.

An SD card will be used for data storage during operation. SD cards use SPI bus for communications, which at 84MHz should not impact the overall performance of the system. However this is still untested. The total number of bytes stored per sampling interval can vary between control scenarios from as low as 20 to as high as 150. Taking the maximum of 150 bytes and assuming that the communications run at 21MHz the data transfer should in theory be completed within 0.1ms.

A number of indicator LEDs are included on the shield. These greatly facilitate the process of debugging as well as providing an idea of what is currently happening in the Arduino. There are also some breakout pins on the board connecting directly to the Arduino's pins. These provide the following functions: (*a*) Serial Port, (*b*) CAN bus and (*c*) 2 DAC outputs, (*d*) 8 ADC inputs.

The Arduino design is Open Hardware, it is therefore possible at a later stage to design a board that combines the functionality of the Arduino and that of the custom built PCB shield. This would reduce the overall size of the controller.

Figure 1 below shows a high level block diagram representation of the system. It must be noted that not all connections are shown in this diagram. For example, the battery voltage is also connected to the Arduino (through a potential divider) and the XSens sensor is powered from the 9V regulator.

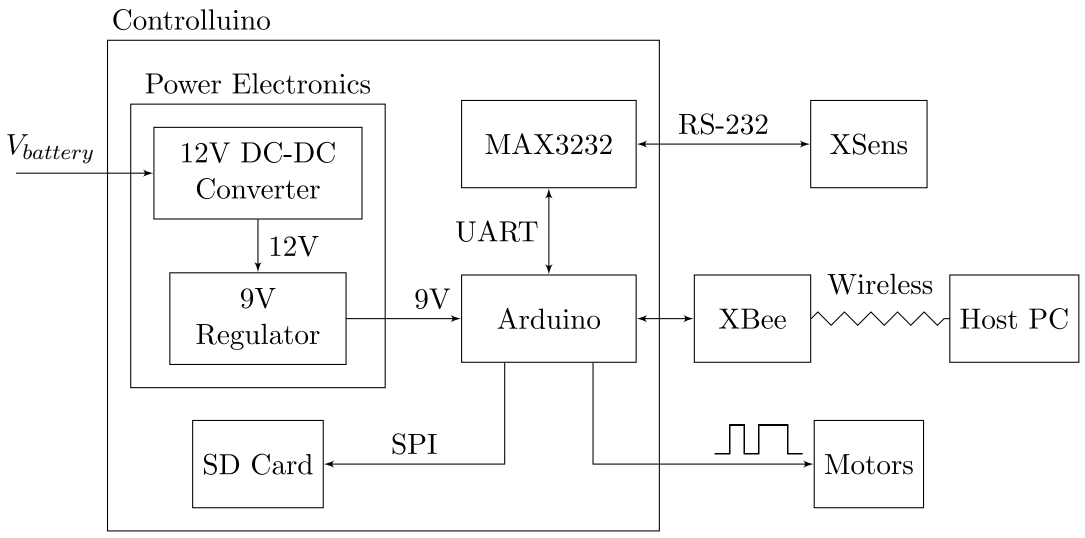


Figure 1: Controlluino Project Block Diagram

3 Software

3.1 Arduino

The Arduino software is split into the following files.

ControlluinoDUE.ino
: Main software structure. Contains the `setup()` and `loop()` functions required by the Arduino, as well as some general purpose functions.

Controlluino.h
: Various constant and macro definitions used throughout the program.

CommsLib.{cpp, h}
: Library for communicating with the host (XBee)

XsensLib.{cpp, h}
: Library for communicating with the sensor

XsensMsgs.h
: Definitions of XSens message headers. Analytical list can be found in the XSens low-level communications protocol documentation.

MotorLib.{cpp, h}
: Library to handle interaction with motors. Contains functions to convert desired speed/angles to PWM pulse width.

Controller.{cpp, h}
: Library containing the controller to be tested.

Functions are commented in depth, therefore maintaining/modifying the program should be easy if anyone wishes to do so. Additionally, a very modular approach was taken in writing the program so libraries could be reused.

Controller Desgin The Controller library must include a function with the prototype

```
void Controller(u, quat, gyro, pos, vel, acc, ref, tfb);
```

where all parameters are pointers to floating point arrays. Table 1 details the usage of each parameter.

Parameter	Type	Length	Description
`u`	`float*`	6	Motor reference points
`quat`	`float*`	4	Quaternion orientation
`gyro`	`float*`	3	Angular rates in body axes
`pos`	`float*`	3	Position vector
`vel`	`float*`	3	Velocity vector in earth axes
`acc`	`float*`	3	Acceleration in body axes
`ref`	`float*`	6	Reference points as received from host PC
`tfb`	`float*`	6	Torques and forces in the body axes

Table 1: Controller function parameters

The user should use the available data to compute a control action and save the desired reference points to each motor in rads or rads/s (depending on the type of each motor) to `u`. It would also be wise to store the desired torque/force values to `tfb` although not doing so would not affect the system's operation.

The library can also contain other supporting functions. For example a prefilter can be programmed as a separate function in this library.

Once the controller is written, the project should be compiled including the new Controller.{cpp, h} files and uploaded to the Arduino. The system is now ready to be deployed.

3.2 PC Interface

A PC program was written to communicate and control the Arduino. Two versions of this were developed, one that provides real time output (plotting up to two sets of data) and one that does not, allowing for faster sampling times. We shall refer to these as *ControlluinoIntefrace* and *ControlluinoSimple*. Only the second program will be documented. Both were written in Java.

NOTE: The RXTX Java library must be installed on the Host PC before the program can be run. For information go to: `http://rxtx.qbang.org`

The ControlluinoSimple program contains 3 tabs. The first is the Control tab which contains all the buttons/functions used during operation. The second is the Settings tab. Here settings such as desired sampling time, transmission interval, reference update interval, XBee COM port and baudrate are available. Finally, the third tab is used to define the Structure of the system. By structure we mean what type of motor is connected to each pin (BLDC or Servo) as well as its parameters. The structures of a trirotor and a quadrotor have been programmed as presets into the program, allowing for quick setup for these two common types of vehicle. Screenshots of these tabs are available in Appendix D. The only data returned from the Arduino is the battery voltage.

3.3 Communications Protocol

Floating Point Numbers Floating point numbers throughout this project are represented by 4 bytes as detailed in the IEEE Standard for Floating-Point Arithmetic (IEEE 754). More specifically, the binary32 format is used commonly referred to as single precision. Conversion from byte arrays to floating point representation and vice-versa can be done either using pointer arithmetic or direct calculation.

Note: The Java library `java.nio.ByteBuffer` provides tools that greatly simplify the above conversion.

Checksum The checksum byte used is defined as in the XSens communication protocol. That is the byte is calculated such that the last byte of the sum of all transmitted bytes (including the check sum) is 0. The required byte can be calculated using the formula

$$cs = 256 - \Sigma\%256,$$

where % is the modulo operator and Σ is the sum of all data bytes.

Transmission Time The transmission time of any message sent can be calculated using the formula

$$T_n = \left(\frac{10}{B} + 10^{-5}\right) n - 10^{-5}\,\text{s},$$

where B is the baud rate and n is the number of bytes. This takes into account both the start and stop bits, as well as the delay between transmissions. The transmission delay between bytes sent from the Arduino is 10μs. The corresponding time for the XSens is not mentioned explicitly in its documentation so it is assumed to be the same.

Messages The messages used in communicating between the Host PC and Arduino are detailed in tables 3 and 2. For more complicated messages, a detailed explanation follows after the two tables.

Function	Header/Message	Notes
Stop	`0xFF,0xFF`	Arduino stops running control loop but does not power off
Power Off	`0xFE,0xFE`	
Start	`0xEE,0xEE`	Start control loop
Sensor	`0xDD,0xDD`	Power up and initialise the sensor
Update Reference	`0xCC,...`	
Setup	`0x22,...`	Power up and initialise the sensor
Motor parameters	`0x33,...`	Update 1 of 6 motor parameters

Table 2: Messages from Host PC to Arduino

Function	Header/Message	Notes
Ready	`0xFF,0xFF`	Arduino completed boot up sequence
Sensor initialised	`0xEF,0xEF`	
Motors intialised	`0xDF,0xDF`	Start control loop
Setup	`0x2F,0x2F`	Setup acknowledgement
Motor parameters updated	`0x3F,0x3F`	Sent when any of the motor parameters are successfully updated
Power Off	`0xCF,0xCF`	Sent before the Arduino powers off
Data	`0xFE,...`	Data packet

Table 3: Messages from Arduino to Host PC

The *"Update Reference"* message for the PC to the Arduino has the following structure `0xCC,DATA,CS`. `DATA` contains the 6 reference points each of which is represented by 4 bytes. An additional checksum byte is aded to the end to confirm that the correct data is received.

The *"Setup"* message is sent to the Arduino before running the experiment. It contains information that define the mode of operation. I has the following structure: `0x22,TXI,SC,SH,OUT1,OUT2,CS`. Each byte represents the following information:

- `TXI` - Transmission interval. This tells the Arduino the how often it should send data back to the host PC. E.g.: If the data should be updated in every sample this should be set to 0, however if it should be updated every 3rd sample it should be set to 2.

- `SC` - Scenario. This affects how the sensor is initialised. There are three options, force control, attitude control and position control. If the selected scenario is force control, then the sensor is not initialised, if it is attitude control then the sensor is initialised to return quaternion and angular rates and finally if position control is selected the sensor returns all data and during initialisation it waits until a GPS signal is received before continuing. `SC` definitions:

  0. Attitude Control
  1. Position Control
  2. Force Control

- `SH` - Desired sampling time. This tells the Arduino the desired sampling time in ms. The Arduino should be able to cope with most sampling times within 10μs.

- `OUT1/2` - Data outputs. Defines what data the Arduino should send back to the Host PC. If the ControlluinoSimple interface is used then these are automatically set to no output. The available outputs are:

  0. Quaternion attitude
  1. Euler angles attitude
  2. Angular rates
  3. Torques
  4. Forces
  5. Servo angles
  6. Motor speeds
  7. Velocity
  8. Position
  9. Acceleration
  10. None

- `CS` - Checksum. Used to confirm that the correct settings were received.

The *"Motor Parameters"* message is sent to update 1 of the 6 configurable motor parameters (applies to all motors) and has the following structure: `0x33,PARAM,DATA,CS`. The `PARAM` byte indicates which parameter will be updated, while the `DATA` field consists of 24 bytes representing either 6 floats or 6 integers depending on the parameter. The exception to this is when the motor types are being updated when `DATA` is only a single byte. `PARAM` definitions:

0. Types - Single byte, right justified. Bits represent the type of each motor with 0 being a BLDC motor and 1 a servo.
1. Offset - The base PWM width in μs that each actuator will respond to. This is the point where BLDC motors are at 0 RPM and servos are at 0 rads. Sent as floats.
2. Gradient - The change change in PWM pulse width in μs required to achieve a unit change in the actuator's output. Sent as floats.
3. Minimum pulse width - The minimum pulse width the actuators respond to. Used as a soft limit. Sent as integers.
4. Maximum pulse width - The maximum pulse width the actuators respond to. Used as a soft limit. Sent as integers.

The *"Data"* message is sent from the Arduino to the Host PC. It has the following structure: `0xFE,LEN,OUT1,OUT2,VB,CS`. `LEN` indicates the number of bytes in the message. `OUT1/2` are the outputs as defined in the *"Setup"* message. Finally, `VB` is the current battery voltage. If both outputs are disabled in the *"Setup"* message then the `LEN` byte is also omitted and the new message structure becomes `0xFE,VB,CS`.

Examples of these messages are available in Appendix C.

A Circuit Diagrams & PCB Layouts

NOT AVAILABLE.

B Datasheets

LM2756 12V Switching DC-DC Converter
`http://www.ti.com/lit/ds/symlink/lm2576.pdf`

LM2940T-9.0 9V Regulator
`http://www.ti.com/lit/ds/symlink/lm2940-n.pdf`

MAX3232 UART to RS-232 transceiver
`http://datasheets.maximintegrated.com/en/ds/MAX3222-MAX3241.pdf`

XSens MTi-G500 INS
Provided with this document

XBee Pro S2B Wireless Communications Module
`http://www.digi.com/pdf/ds_xbeezbmodules.pdf`

XBee S1 Wireless Communications Module
`http://www.digi.com/pdf/ds_xbeemultipointmodules.pdf`

C Example Messages

Update Reference contains 6 floating point numbers along with a header (`0xCC`) and checksum byte, adding up to a total of 26 bytes. In the following example the values sent are {-0.5,-1,0.3,-1.3,0.2,5.7}.

`0xCC`	Header
`0xBF 0x00 0x00 0x00`	Reference 1
`0xBF 0x80 0x00 0x00`	Reference 2
`0x3E 0x99 0x99 0x9A`	Reference 3
`0xBF 0xA6 0x66 0x66`	Reference 4
`0x3E 0x4C 0xCC 0xCD`	Reference 5
`0x40 0xB6 0x66 0x66`	Reference 6
`0xE2`	Checksum

Setup message used to configure Arduino to return Quaternion and Angular Rate outputs can be constructed as follows:

`0x22`	Header
`0x01`	Transmission Interval 1
`0x00`	Attitude Control (0)
`0x14`	Sampling Time 20ms
`0x00`	Output 1 - Quaternion
`0x02`	Output 2 - Angular Rates
`0xE9`	Checksum

Similarly, to disable outputs while keeping the rest of the parameters the same, the message can be constructed as:

`0x22`	Header
`0x01`	Transmission Interval 1
`0x00`	Attitude Control (0)
`0x14`	Sampling Time 20ms
`0x0A`	Output 1 - None
`0x0A`	Output 2 - None
`0xD7`	Checksum

In both of the above cases, the Arduino will send data back to the Host every other sample, which combined with the desired sampling time of 20ms implies that the Host will receive data every 40ms. Furthermore, the sensor will be initialised to give quaternion and angular rate measurements since the scenario is set to Attitude Control.

Data message sent to the Host PC using the first **Setup** above will contain quaternion and angular rate outputs. Let the output values be $\{\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0, 0\}$ and {0.3,0.5,-0.1} respectively and the battery level be 170/255. The resulting message would be:

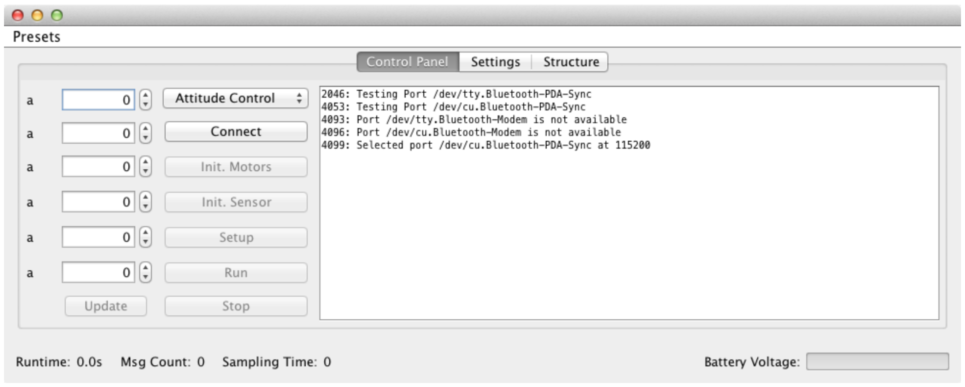
`0xFE`	Header
`0x1C`	Length 28
`0x3F 0x35 0x04 0xF3`	q_0
`0x3F 0x35 0x04 0xF3`	q_1
`0x00 0x00 0x00 0x00`	q_2
`0x00 0x00 0x00 0x00`	q_3
`0x3E 0x99 0x99 0x9A`	Rate about $x_b\,(p)$
`0x3F 0x00 0x00 0x00`	Rate about $y_b\,(q)$
`0xBD 0xCC 0xCC 0xCD`	Rate about $z_b\,(r)$
`0xAA`	Battery Voltage
`0xF9`	Checksum

As for the second **Setup**, the **Data** message would be:

`0xFE`	Header
`0xAA`	Battery Voltage
`0x56`	Checksum

From the above, it is obvious that disabling realtime output to the Host PC would allow for considerably faster sampling times.

D Interface Screenshots

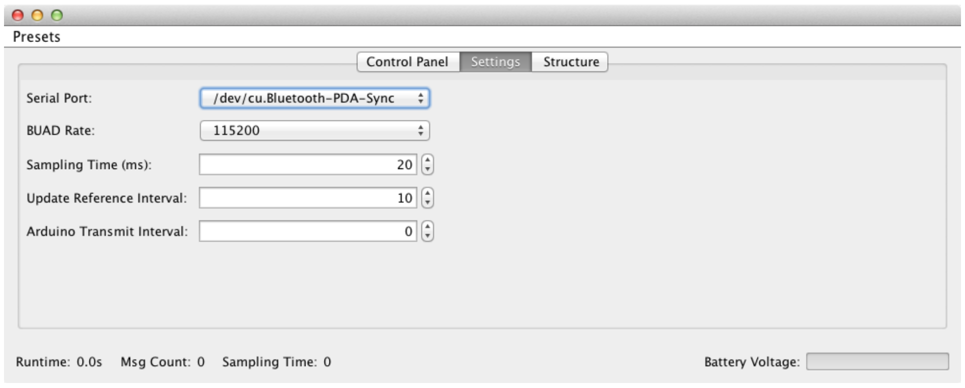


a 0
a 0
a 0
a 0
a 0
a 0

```
2046: Testing Port /dev/tty.Bluetooth-PDA-Sync
4053: Testing Port /dev/cu.Bluetooth-PDA-Sync
4093: Port /dev/tty.Bluetooth-Modem is not available
4096: Port /dev/cu.Bluetooth-Modem is not available
4099: Selected port /dev/cu.Bluetooth-PDA-Sync at 115200
```

Runtime: 0.0s Msg Count: 0 Sampling Time: 0

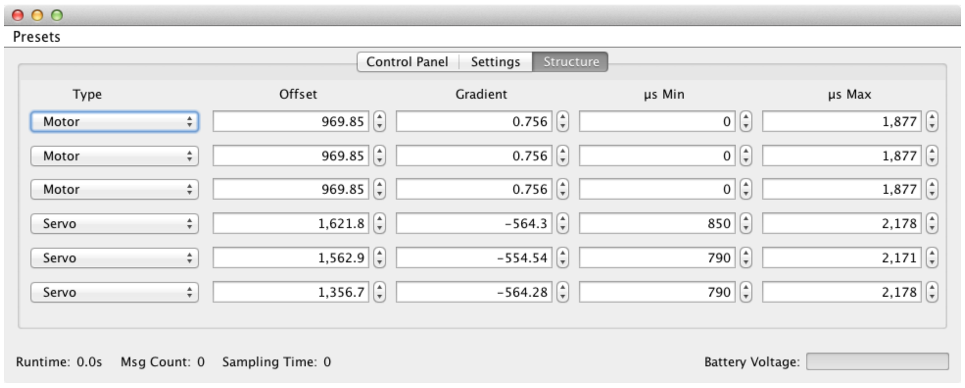
Figure 2: Control Panel Tab



Serial Port:	/dev/cu.Bluetooth-PDA-Sync
BUAD Rate:	115200
Sampling Time (ms):	20
Update Reference Interval:	10
Arduino Transmit Interval:	0

Runtime: 0.0s Msg Count: 0 Sampling Time: 0

Figure 3: Stettings Tab



Presets

Type	Offset	Gradient	µs Min	µs Max
Motor	969.85	0.756	0	1,877
Motor	969.85	0.756	0	1,877
Motor	969.85	0.756	0	1,877
Servo	1,621.8	-564.3	850	2,178
Servo	1,562.9	-554.54	790	2,171
Servo	1,356.7	-564.28	790	2,178

Runtime: 0.0s Msg Count: 0 Sampling Time: 0 Battery Voltage:

Figure 4: Structure Tab