9th Sem.

Controlling Mobile SDK via External Hardware

Advanced Programming in C/C++

By B. Sc. Efthimis Pegas

National Technical University of Athens
21/12/2017

This work is destined for academic purposes only.

Dedicated to all those who strive to make the world a better virtual one.

Special thanks to my advisors and mentors
both known and anonymous.

Authored and published in Germany

Duisburg – Essen University, Germany
2 Forsthausweg
Duisburg, 47057

efthimispegas@gmail.com

TABLE OF CONTENTS

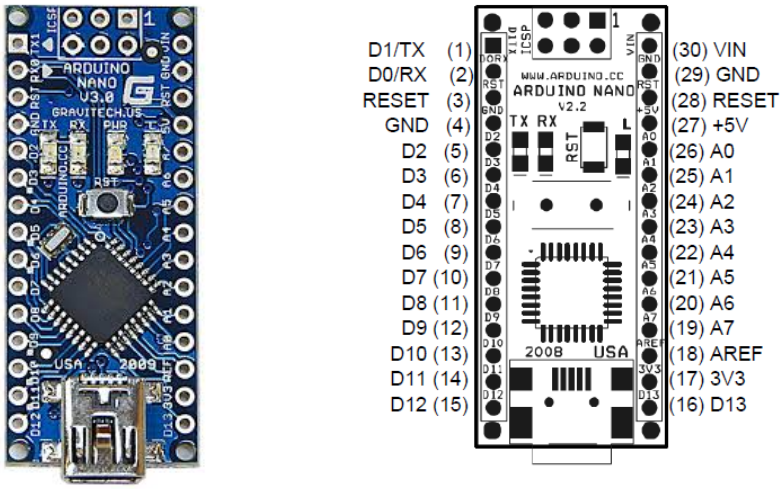
1. Introduction

1.1 A Grasp of the Arduino Board and IDE

Arduino is an open-source platform used for building electronics projects. Arduino consists of both a physical programmable circuit board (often referred to as a microcontroller) and a piece of software, or IDE (Integrated Development Environment) that runs on your computer, used to write and upload computer code to the physical board.

The Arduino platform has become quite popular with people just starting out with electronics, and for good reason. Unlike most previous programmable circuit boards, the Arduino does not need a separate piece of hardware (called a programmer) in order to load new code onto the board – you can simply use a USB cable. Additionally, the Arduino IDE uses a simplified version of C++, making it easier to learn to program. Finally, Arduino provides a standard form factor that breaks out the functions of the micro-controller into a more accessible package.

The Arduino Nano (see Pic. 1) of the Arduino Family is a small, complete, and breadboard-friendly board based on the ATmega328 (Arduino Nano 3.x). It has more or less the same functionality of the Arduino Duemilanove, but in a different package. It lacks only a DC power jack, and works with a Mini-B USB cable instead of a standard one.



Pic. 1: The left image shows the *Arduino Nano* board and the right image shows the pinout of the board

2. Developing in the Arduino IDE

2.1 A Simple Example

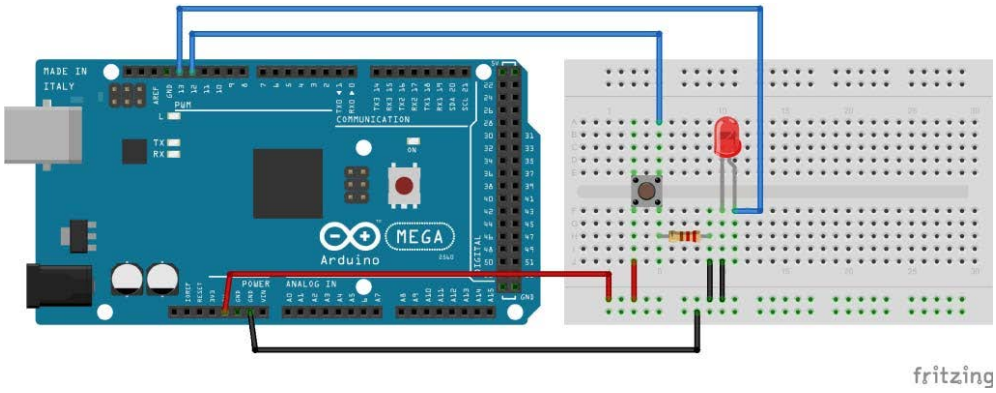
The programming of the board is basically what makes it so commonly used for a vast variety of applications. In a quick demonstration we will try to interact with the board by establishing a serial connection between the computer, the board, a two-state button and an LED (see Pic. 2). We initialize the pin that refers to the LED (pin *13*) and we create the main loop with the intention to make the LED blink for 0.5 sec. The code is the following (see also in *2* folder):

- BlinkLED.ino

```
// initialize pin 13 of the arduino board as the output
int LED = 13;
int i=0;

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  Serial.begin(9600);
  pinMode(LED, OUTPUT);
}
void loop() {
  int t = millis()/1000; // holds the time (in sec) since the last reset

    // print the time from last reset
    if (t >= 10) {
      Serial.print("I was reset ");
      Serial.print(t);
      Serial.print(" seconds ago...");
      Serial.println();
      Serial.println("Stop me anytime now...");
      delay(1000);
    }
```

```
  else{
    for (i; i<10; i++) {
      digitalWrite(LED, HIGH);   // turn the LED on (HIGH is the voltage level)
      delay(500);                // wait for 0.5 seconds
      digitalWrite(LED, LOW);    // turn the LED off by making the voltage LOW
      delay(500);                // wait for 0.5 seconds
      //t++
      Serial.print("I was reset ");
      Serial.print(i);
      Serial.print(" seconds ago...");
      Serial.println();
      Serial.println("Stop me anytime now...");
    }

    //delay(1000);
    //printf("I was reset %d seconds ago...\nStop anytime now...\n", t);
  }
```

We blink the LED 10 times and then the for loop is never executed again. If we want to use the button as an external *RESET* then we only have to connect its pins to the *RST* and the *GND* pins of the board correspondingly.



Pic. 2: A simple circuit.

3. Arduino and Visual Studio 2013 Serial Communication

3.1 Control Arduino via Visual Studio (Send Data Only)

In the Arduino IDE the communication between the Arduino board and the PC is being established implicitly when we upload a sketch to the board. In Visual Studio 2013 we need to make sure that we manually create and include a Serial library in our project. After some research, we managed to open a communication channel, in which we simulate a digital lock. Specifically, we give a string of characters as an input in Visual Studio command prompt and if the input complies with the given restriction we enable the LED to turn on indicating the correct result. Otherwise, the LED blinks 3 times and goes off. The program in Visual Studio C++ is the following (see also in the *3.1* folder):

- arduino.cpp

```
#include <iostream>
#include <stdio.h>
#include <conio.h>
#include "tserial.h"
#include "bot_control.h"

using namespace std;

serial comm; //serial is a class type defined in these files, used for referring to the
communication device

void main() {

    char answer, data[100]; //To store the character to send
    int i = 0;
    do {
            cout << "Enter characters to be sent:\t"; //User prompt
            cin >> data;

            comm.startDevice("COM3", 9600);

            printf("Started\n");
            /* “COM 2” refers to the com port in which the USB to SERIAL port is attached. It is
shown by right clicking on my computer, then going to properties and then device manager
            9600 is the baud-rate in bits per second */
            comm.send_data(&data[0]); //The data is sent through the port
            printf("\n");
            printf("Data sent to device.\nProceeding...\n");

            Sleep(2000); //waits for 2 sec
            printf("Try again? (y/n)\n");
            cin >> answer;
            comm.stopDevice(); //The device is closed down
    } while (answer == 'y');
}
```

- SerialPortComm.ino

```
void setup() {
      pinMode(13, OUTPUT);
      Serial.begin(9600);
    }

    void loop(){
      if(Serial.available()){
        String ch;
        ch = Serial.readString();
        Serial.setTimeout(2000);
        ch.trim();
        if(ch=="ep215"||ch=="EP215"){
          digitalWrite(13, HIGH);
          Serial.println("Unlocked...");
          delay(2000);
          digitalWrite(13, LOW);
        }
        else {
          Serial.println("Wrong password, please try again");
          digitalWrite(13, HIGH);
          delay(500);
          digitalWrite(13, LOW);
          delay(500);
          digitalWrite(13, HIGH);
          delay(500);
          digitalWrite(13, LOW);
          delay(500);
          digitalWrite(13, HIGH);
          delay(500);
          digitalWrite(13, LOW);
        }
    }}
```

3.2 Bidirectional Communication (Send & Receive Data)

The next step was to establish bidirectional communication between the two platforms, in order to send data from Arduino to Visual Studio and then invert the process by receiving data to Arduino from Visual Studio. After connecting the two platforms through a serial port, we exchange messages and data which result to altering the state of the LED. The code is presented below (see also in the *3.2* folder)::

- SendReceive.ino

```
#define BUTTON_PIN 5
#define LED 13
#define DELAY 20 //in ms

boolean was_pressed; //prev state

void setup() {
  Serial.begin(9600); //baud rate
  pinMode(LED, OUTPUT);
  pinMode(BUTTON_PIN, INPUT);
  digitalWrite(BUTTON_PIN, HIGH); //negative logic
}

//function that checks if the button is pressed or not
boolean handle_button() {

  boolean event;
  int pressed = !digitalRead(BUTTON_PIN); //negative logic -> pin low means HIGH

  event = !(pressed && !was_pressed);
  return event;
}
```

```
void loop() {
  //store the event (only when pressed)
  boolean rising_edge = handle_button();
  char start = '0';
  if(rising_edge) {
    //receives the signal that button isn't pressed
      start = '0';
  }
  else {
    //receives the signal that button is being pressed
    start = '1';
    Serial.println(start); //send it to VS2013
  }
   //writing data from VS to arduino
  String input;
  //If any input is detected in arduino
  if(Serial.available() > 0){
    //read the whole string until '\n' delimiter is read
    input = Serial.readStringUntil('\n');
    //If input == "on" then turn on the led and send a reply
    while (start == 1) {
    }
        if (input.equals("on")){
        digitalWrite(LED, HIGH);
        Serial.println("LED is on");
      }
    //If input == "off" then turn off the led and send a reply
    else if (input.equals("off")){
      digitalWrite(LED, LOW);
      Serial.println("LED is off");
    }
```

```
}
delay(DELAY);}
```

- SendReceive.cpp

```
#include <iostream>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "SerialPort.h"

using std::cout;
using std::cin;
using std::endl;
using std::find;

using namespace std;



/*Portname must contain these backslashes, and remember to
replace the following com port*/
char *port_name = "\\\\.\\COM3";

//String for incoming data
char incomingData[MAX_DATA_LENGTH];

//String for outgoing data
char output[MAX_DATA_LENGTH];

SerialPort arduino(port_name);

/*----------------------------------------*/
void send() {

   while (arduino.isConnected()){
          std::string input_string;
          char* c_string;

                cout << "Asking permission to proceed.([Type] on/off): ";

                //Getting input
                getline(cin, input_string);
                printf("\n");

                //Creating a c string
                c_string = new char[input_string.size() + 1];
                //copying the std::string to c string
                std::copy(input_string.begin(), input_string.end(), c_string);
                //Adding the delimiter
                c_string[input_string.size()] = '\n';
                char ans = c_string[1];

                //                  cout << ans << endl;

                //writing string to arduino
```

```
                if (ans == 'n') {
                        //turning led on
                        printf("Permission granted.\n");
                        //Sleep(500);
                        printf("Turning led on now...\n\n");
                        //flag = !flag;
                }
                else if (ans == 'f') {
                        //turning led off
                        printf("Permission granted.\n");
                        //Sleep(500);
                        printf("Turning led off now...\n\n");

                }

        //      bool flag = false;
        //      do {
        //              printf("Do you wish to continue? ([Type] y/n): ");
        //              cin >> answer;
        //              printf("%s\n", answer);
        //              if (answer != 'y' || answer != 'n'){
        //                      printf("Not a valid answer...\n");
        //                      flag = !flag;
        //              }
        //} while (flag);

        //-->here check in arduino what to do according to the order
        arduino.writeSerialPort(c_string, MAX_DATA_LENGTH);
        //Getting reply from arduino
        arduino.readSerialPort(output, MAX_DATA_LENGTH);
        //printing the output
        cout << output;
        //freeing c_string memory
        delete[] c_string;
    }
}
/*---------------------------------------------*/
void receive() {
//this while loop handles the signal sent from arduino -> waits button to be pressed
    while (arduino.isConnected()) {
        int read_result = arduino.readSerialPort(incomingData, MAX_DATA_LENGTH);
        //prints out data
        if (atoi(incomingData) == 1) {
                //start button pressed ->do stuff
                //cout << incomingData << endl; //prints out data from arduino
                printf("Button pressed...\n");
                Sleep(500);
                printf("Commencing procedure...\n");
                Sleep(500);
                break;
        }
    }
}
/*---------------------------------------------*/
int main()
{

    if (arduino.isConnected()) {
        cout << "Connection Established" << endl;
    }
    else
```

```
            cout << "ERROR, check port name" << endl;

    receive();
    //wait a bit between comms
    Sleep(100);

    send();
    //wait a bit
    Sleep(500);
}
```

3.3 Sensing a potentiometer (Transmitting Data Only)

In the next example, we used the Arduino as a resistance sensor and we then transmitted the value to Visual Studio. This example is essential to the project's purpose to manipulate a pendulum's angle according to a potentiometer's value (see also in the *3.3* folder):.

- PotentiometerSensor.ino

```
#define POTENTIOMETER_PIN A7 //potentiometer's pin

#define LED_PIN 13

#define DELAY 20 //ms

int out = 0; //store the sensor's value


void setup() {

  Serial.begin(9600);

  pinMode(LED_PIN, OUTPUT);

  pinMode(POTENTIOMETER_PIN, INPUT);

}


void loop() {

  out = analogRead(POTENTIOMETER_PIN); //read value of potentiometer

//  digitalWrite(LED_PIN, HIGH);

//  delay(out); //delaying (out) ms


//  digitalWrite(LED_PIN, LOW);

//  delay(out); // delaying (out) ms


  Serial.print("DELAY (ms) : ");
```

```
 Serial.println(out);

 delay(500);

}
```

- Potentiometer.cpp

```
#include <iostream>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "SerialPortComm.h"

using std::cout;
using std::cin;
using std::endl;
using std::find;

using namespace std;

/*Portname must contain these backslashes, and remember to
replace the following com port*/
char *port_name = "\\\\.\\COM3";

//String for incoming data
char incomingData[MAX_DATA_LENGTH];

//String for outgoing data
char output[MAX_DATA_LENGTH];

SerialPort arduino(port_name);

int main() {
    //this while loop handles the signal sent from arduino -> waits button to be pressed

    if (arduino.isConnected())
            printf(" Connection established...\n");
    else
            printf("ERROR! Check connection cable.\n");

    while (arduino.isConnected()) {
            int read_result = arduino.readSerialPort(incomingData, MAX_DATA_LENGTH);
            //prints out data
            cout << incomingData << endl; //prints out data from arduino
            //start button pressed ->do stuff

            //hold on for a while
            /*Sleep(500);*/
    }
}
```

3.4 Multithreading and Serial Communication

In hardware development, when you program your hardware to perform one task everything is simple; your code consists of a main loop that performs all the actions. Such an example was presented in *§3.3*, where we implemented a potentiometer sensor. But what happens when you need to perform several tasks between your software and hardware?

Having the main loop go through all the tasks can be time consuming, as we can easily realize, based on the example described in *§3.2*. In that example, we called the same function for transferring and receiving data. That cost us time, as well as memory space.

We noticed that when we received the decision ("turn led on"/"turn led off") from Visual Studio to Arduino (*handshake*), it took some significant amount of time for the Arduino to reply with the result ("LED turned on" or "LED turned off"). Furthermore, we noticed that the launch signal (*start*) was also printed ("1") during the printing of the results to the Visual Studio's screen, which is superfluous, memory – consuming, if not frivolous way of coding.

These problems can be resolved by implementing multithreading developer techniques to our multitasking programs. We implemented the examples in §3.2 and §3.3 with the use of threads[(1)]. The results are increasingly better in terms of speed, memory usage and overall performance. We can now see that, the messages are printed real – time and we avoid printing the additional aces stored in the buffer when we press the launch button (*start = 1*).

The code is the following (see also in the *3.4* folder)::

- DataTransmitter.ino

```
#include <Thread.h>
#include <StaticThreadController.h>


#define POTENTIOMETER_PIN A7 //potentiometer's pin
```

[1] In this case we could implement a server to handle the button as an external interrupt, which is quite similar to implementing thread techniques by reduction to only one thread.

```
#define LED_PIN 13
#define BUTTON_PIN 5
#define DELAY 20 //ms

int out = 0; //store the sensor's value
int pressed;
boolean was_pressed = 0; //prev state
char start = '0';
char quit = '0';
boolean rising_edge;



//instantiate myThread
Thread* myThread = new Thread();



//measurements handler - myThread2
void Callback(){
  if (start == '1'){
    digitalWrite(LED_PIN, HIGH);
    out = analogRead(POTENTIOMETER_PIN); //read value of potentiometer
    Serial.print("DELAY:");
    if (out < 100)
      Serial.print("0");
    if (out < 10)
      Serial.print("0");
    Serial.println(out);
    delay(350);
  }
}
//function that checks if the button is pressed or not
boolean handle_button() {
  boolean event;
  pressed = !digitalRead(BUTTON_PIN); //negative logic -> pin low means pressed so HIGH
```

```
  //initially pressed = 0 ->digitalRead returns 1 (negative logic)
  event = (pressed && !was_pressed);
  return event;
}

StaticThreadController<1> controller(myThread);

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  pinMode(LED_PIN, OUTPUT);
  pinMode(POTENTIOMETER_PIN, INPUT);
  pinMode(BUTTON_PIN, INPUT);
  digitalWrite(BUTTON_PIN, HIGH); //negative logic

 //configure threads
  myThread->onRun(Callback);
  myThread->setInterval(10);//doesn't matter - is executed only once



  while(!rising_edge){
    rising_edge = handle_button();
    if(!rising_edge) {
    //receives the signal that button isn't pressed
      start = '0';
    }
    else {
    //receives the signal that button is being pressed
      start = '1';
      Serial.println(start); //send it to VS2013 to start
    }
  }
}
```

```
void loop() {


  controller.run();

}
```

- DataReceiver.cpp

```
#include <iostream>
#include <stdio.h>
#include <windows.h>
#include <thread>

#include "Serial.h"


using namespace std;

Serial arduino;

//String for incoming data
char incomingData[MAX_DATA_LENGTH];

//String for outgoing data
char* output;

/*-------------------------------------------*/

void parseInput(char * buffer)
{
    string input;
    int measurement;
    input = string(buffer);

    string firstFive = input.substr(0, 5);
    string nextOne = input.substr(5, 1);
    string lastThree = input.substr(6, 3);
    string delayStr = string("DELAY");
    string punctStr = string(":");
    if (!strcmp(firstFive.c_str(), delayStr.c_str()) && !strcmp(nextOne.c_str(),
punctStr.c_str()))
    {
           measurement = atoi(lastThree.c_str());
           cout << "Measurement = " << measurement << endl;
    }
    else // invalid input
    {
           cout << "Invalid input" << endl;
    }
}

/*-------------------------------------------*/

void launchSignal() {
    while (arduino.isConnected()) {
           arduino.read(incomingData, MAX_DATA_LENGTH);
           if (atoi(incomingData) == 1) {
```

```
                printf("Launch...\n");
                output = "on\n";
                break;
            }
    }
}

/*---------------------------------------------*/

int main()
{
    arduino.open();
    if (arduino.isConnected())
            printf("Connection Established\n");
    else
            printf("Error! Check connection cable.\n");

    const int buffLen = 9; // "DELAY:xxx" is 9 characters
    char buffer[buffLen];

    std::thread start(launchSignal);
    start.join(); //pause until start finishes


    for (int iterCount = 0; iterCount < 100; iterCount++) // loop for a while and quit
    {
            arduino.read(buffer, buffLen); // Doesn't return until buffLen bytes are read
            parseInput(buffer);
    }


    return 0;
}
```

In the script above we basically integrated all the essential actions that we need our software – hardware interaction to consist of. This includes launch, data transmission, data reception and data interpretation. All that remains now is the implementation in Mobile SDK platform.

4. Mobile – Arduino Interaction

Our main purpose is to be able to manipulate a structure's properties (i.e. initial position, velocity, acceleration etc.) via external hardware. Thus, the first issue we need to focus on is *merging* Mobile software with 3rd party libraries and source code. Then, we need to tackle the strictly secured Mobile infrastructure, which allows very few, if not any alterations, by creating our own *hybrid* classes. Finally, we use these modifications to create the required environment for our application in 3 steps.

4.1 Integrating C++ source code into Mobile SDK

Here we will discuss the first issue we need to address, including the Serial Communication library. We need prepare a serial port, in order to establish a communication channel between the two platforms, as illustrated previously (in §3.1 - §3.4). This is accomplished by adding all necessary functions (open, read, write, close etc.) to our project. These functions constitute the objects of our new class (*MoSerialPort*) which we will discuss later. This way, we render our software able to receive data and transmit signals from and to the Arduino board, throughout a use-specific interface.

4.2 Creating Hybrid Classes

A **class** in **C++** is a user defined type or data structure declared with keyword *class* that has data and functions (also called methods) as its members whose access is governed by the three access specifiers *private*, *protected* or *public* (by default access to members of a class is *private*). The private members are not accessible outside the class; they can be accessed only through methods of the class. The public members form an interface to the class and are accessible outside the class.

Our class is *MoSerialPort* and its structure is similar to a *MoMap* with a few add-ons. It consists of two parts; the serial comms functions and Mobile's modified motion methods (hence the "hybrid" denomination). The serial comm methods are based on the structures we illustrated in §3. The altered motion methods follow the old structure, but come with a few changes to meet our requirements.

The essential methods are described below:

- The *read*() is the function that is being called whenever there is data that needs to be transmitted from the hardware to the software interface. That is, when we receive the launch signal and when we receive the potentiometer's value. It stores the data in a *buffer* as a series of char types of size *buff_size* which changes respectively to the string's size (1 for the launch signal, 9 for the measurement message)

- The *launchSignal()* is similar to the one created in §3.4 with the only difference that we now choose not to parse the incoming data to the function (instead we are declaring it at the public sector of the class). This function receives the launch signal (as the name states) from the Arduino board, once the board is securely connected and the button is pressed. This is the first instance of hardware – software interaction in our project. The Arduino senses the change in the button's state and sends the signal to a handler (thread implementation - §3.4), which is responsible for the signal transmission to our software. It uses the *read*() function. After the launch signal is received (*buff_size = 1*), the buffer's size changes to meet the needs of the *parseInput()* function (*buff_size = 9*).

- The *parseInput()* has the same form as described in §3. Takes no arguments, and uses the *buffer[buff_size]* to store the message that is being transmitted from the Arduino. It deciphers the string message into a double, which then stores into a MoReal variable. We will use this variable later to convert it into rads and assign it to the angular variable that gives pendulum's initial position. This method is the second and more complicated instance of data transmission. It transforms the received message (string) from the Arduino, which holds the measurement (double).

- The *doMotion()* has 1 argument, the action (a *MoTransmissionSubtask*) which focuses on the system's position (thus its assigned value is *DO_POSITION*). This method's purpose is to initialize the mechanism's state every time the main loop is executed (this depends on the increment – dT of the simulation). This is achieved by calling *read()* and *parseInput()* in the beginning of the function.

In the simulation, this yields that the angle be updated every dT secs with a new value coming from the Arduino with the minimum delay possible.

The class's script is the following (see also in final\ArduinoMobileComm_v5.1):

```
#include <iostream>
#include <cstring>  //std::memset
#include <thread>

using namespace std;

#include "MoSerialPort.h"
//#include "MoMapChain_peiragmeno.h"


void MoSerialPort::init(/*MoAngularVariable &angle_, MoReal &a_, const string& name_*/)
{

	/*angle = &angle_;
	a = &a_;
	setName(name_);*/


	memset(&m_OverlappedRead, 0, sizeof(OVERLAPPED));

}

void MoSerialPort::doMotion(MoAngularVariable &angle_, MoReal &a_, MoTransmissionSubtask action)
{

	// position kinematics
	 if (action & DO_POSITION) { // carry out position-dependent computations
			angle_.q = ((a_*360)/917)*DEG_TO_RAD; //digital to degree to rad conversion
			cout << angle_.q << endl;
			Sleep(70);
	 }

	// Block II: velocity kinematics
	if (action & DO_VELOCITY) { // carry out velocity-dependent computations

    }

	// Block IV: EULER acceleration kinematics

	if (action & DO_ACCELERATION) {

	}

	if (action & DO_EULER) { // EULER ACCELERATION same as velocity transmission but now with
accelerations as inputs

    }

	// Block V: compute CORIOLIS terms
	if (action & COMPUTE_CORIOLIS ) { // compute CORIOLIS terms for later use

	}
	// Block V: add CORIOLIS terms
	if (action & USE_CORIOLIS) { // add CORIOLIS terms to the current acceleration

	}

}
```

```
//instantiate my functions

void MoSerialPort::open()
{
	// reset DCB
	memset(&m_dcb, 0, sizeof(m_dcb));

	// open the serial port
	m_pComPortHandle = CreateFile("\\\\.\\COM3", GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, NULL);

	// set up the overlapped event
	m_OverlappedRead.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

	// set the DCB
	GetCommState(m_pComPortHandle, &m_dcb);
	m_dcb.DCBlength = sizeof(m_dcb);
	m_dcb.BaudRate = 9600;
	m_dcb.Parity = NOPARITY;
	m_dcb.ByteSize = 8;
	SetCommState(m_pComPortHandle, &m_dcb);

	this->connected = true;
	Sleep(ARDUINO_WAIT_TIME);

}

void MoSerialPort::read(char *buffer, int dwBytesToRead)
{
	BOOL bReadStatus;
	DWORD dwBytesRead, dwErrorFlags;
	COMSTAT comStat;

	// Empty the input buffer
	PurgeComm(m_pComPortHandle, PURGE_RXCLEAR);

	// Set comStat.cbInQue to zero (number of bytes in the input buffer)
	ClearCommError(m_pComPortHandle, &dwErrorFlags, &comStat);

	// Get the number of bytes in the input buffer
	dwBytesRead = (DWORD)comStat.cbInQue;

	// Wait until the unput buffer is full
	while (dwBytesRead <= dwBytesToRead)
	{
		ClearCommError(m_pComPortHandle, &dwErrorFlags, &comStat);
		dwBytesRead = (DWORD)comStat.cbInQue;
	}

	// Read the input buffer
	bReadStatus = ReadFile(m_pComPortHandle, buffer, dwBytesToRead + 2, &dwBytesRead,
&m_OverlappedRead);

	// Make sure the unput buffer is empty
	PurgeComm(m_pComPortHandle, PURGE_RXCLEAR);

	buffer[dwBytesToRead] = 0;
}

bool MoSerialPort::isConnected()
```

```
{
	return this->connected;
}

void MoSerialPort::close()
{
	if (m_OverlappedRead.hEvent != NULL)
		CloseHandle(m_OverlappedRead.hEvent);

	if (m_pComPortHandle != NULL)
		CloseHandle(m_pComPortHandle);

	if (this->connected)
		this->connected = false;
}

void MoSerialPort::parseInput(char* buffer, MoReal &a)
{
	string input;
	/*int measurement;*/
	input = string(buffer);

	string firstFive = input.substr(0, 5);
	string nextOne = input.substr(5, 1);
	string lastThree = input.substr(6, 3);
	string delayStr = string("DELAY");
	string punctStr = string(":");
	if (!strcmp(firstFive.c_str(), delayStr.c_str()) && !strcmp(nextOne.c_str(),
punctStr.c_str()))
	{
		a = atoi(lastThree.c_str());
		cout << "Measurement = " << a << endl;
	}
	else // invalid input
	{
		cout << "Invalid input" << endl;
	}
}

void MoSerialPort::launchSignal(char inData[MAX_DATA_LENGTH]) {

	while (isConnected()) {
		read(inData, MAX_DATA_LENGTH);
		if (atoi(inData) == 1) {
			printf("Launch...\n");
		//printf("%d", atoi(inData));
			break;
		}
	}
}


//other necessary functions instatiations
void MoSerialPort::doForce(MoTransmissionSubtask action_)
{
	if (action_ & DO_EXTERNAL)
	{

	}
}
```

```
void MoSerialPort::cleanUpForces()
       {

       }

void MoSerialPort::pushForces()
       {

       }

void MoSerialPort::popForces()
  {
       }

void MoSerialPort::print(ofstream& outFile)
{

        outFile << integratorTime->modelTime << " ";

}
```

5. Single Pendulum Application

After we finish creating our new class, we move onto the main task. Implementation in a real – life application. We create our project, including the essential libraries as described and the new class we generated. The task can be decomposed in three steps:

1) Ensuring that the measurement is correctly parsed into Mobile application and assigned to a Mobile's variable. For this reason, we created a simple application of an 1R mechanism (one degree of freedom and one link) which outputs the measurements to the Visual Studio's prompt after assigning them to a Mobile variable (see also ArduinoMobileComm_v1)

2) Testing that the measurement is parsed correctly into the doMotion(3) method and refreshed every time the loop is executed. The code follows the one above, of a simple 1R mechanism, with the addition of a for loop in order to see how the application responds when time – dependent (see also ArduiniMobileComm_v2).

3) Reapplying those steps in a real – life application, including graphics. We further created a MoScene in order to visualize the simple mechanism. Main purpose of this step is to update the mechanism's inertia properties in correspondence with a new measurement (see also ArduinoMobileComm_v3).

Eventually, we developed the final application of a simple pendulum, where we initialize the angle with the potentiometer every time we open the Mobile graphic editor (see also ArduinoMobileComm_v5.1). We cite the main script here for justification but you can always refer to the eligible folder and have an extended look as well as test the code:

```
/*************************************************************************
*                                                                        *
*  This version (_v5.1) demonstrates how the pendulum's initial state changes *
*  respectively to the potentiometer's value with graphics               *
*                                                                        *
*************************************************************************/

#include <stdio.h>
#include <windows.h>
#include <fstream>

//#include <thread>

// Basic Mobile
#include <Mobile/MoBase.h>
#include <Mobile/MoMapChain.h>
```

```
#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoMassElement.h>

#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoMassElement.h>
#include <Mobile/MoMechanicalSystem.h>
#include <Mobile/MoAdamsIntegrator.h>

#include <Mobile/Inventor/MoScene.h>
#include <Mobile/Inventor/MoWidget.h>

// own classs
#include "MoSerialPort.h"



int main()
{
	//simulation data to output file
	ofstream outFile;
	outFile.open("../data/resultsSimulation.dat");
	outFile.precision(14);
	cout.precision(14);

	//mobile initializations
	MoFrame
		K0("K0"), // world coordinate system (assumed to be at rest)
		K1("K1"),	// frame 1
		K2("K2");	  // frame 2

	MoAngularVariable
		phi("phi");

	MoVector l;

	MoElementaryJoint R(K0, K1, phi, zAxis); //create a revolut joint between frames K0, K1
	MoRigidLink rod(K1, K2, l);
	MoReal a;

	MoSerialPort arduinoMeasurement(phi, a);

	//open serial communication
	arduinoMeasurement.open();
	if (arduinoMeasurement.isConnected())
		printf("Connection Established\n");
	else
		printf("Error! Check connection cable.\n");


	MoMapChain pendulum;

	//geometry and inetria properties
	l = MoVector(0, -1, 0);

	arduinoMeasurement.launchSignal(); // awaits the start signal

	arduinoMeasurement.read();  // Reads the input bytes sent from arduino
	// Doesn't return until buffLen bytes are read
	arduinoMeasurement.parseInput(); // Transforms the input from char->int
                                                // Prints out measurement
```

```
        arduinoMeasurement.doMotion(DO_POSITION);

        pendulum << arduinoMeasurement << R << rod;

        pendulum.doMotion(DO_POSITION);

        MoReal dT = 0.01;

        //visualization
        MoMapChain animationChain;
        animationChain << pendulum;

        MoScene scene(pendulum);

        scene.addAnimationObject(animationChain);
        scene.setAnimationIncrement(dT);

        scene.makeManipulator(R);
        scene.makeShape(R, rod);
        scene.makeShape(K0, 0.4);
        scene.makeShape(K1, 0.4);
        scene.makeShape(K2, 0.4);

        MoMapChain widgetChain;
        widgetChain << pendulum;

        MoWidget widget;
        widget.init(scene, widgetChain, "time");
        widget.addScrollBar(phi.q, 0.0, 360.0, "phi.q", 3);
        widget.addScrollBar(phi.qd, -5.0, 5.0, "phi.qd", 3);
        widget.addSlider(a, 0, 917, "measurement");

        scene.show(); //shows scene
        MoScene::mainLoop();

        return 0;
}
```

It is strongly recommended that you view the code folder for both a better understanding and a a complete image of the the source code. There is the new class included in the include folder, as well as the class .cpp file in the src folder. The parts that are shown here are merely justifying my work and may carry bugs, since there have been several fixes and updates. You can find the final work in the final/ArduinoMobileComm_v5.1 folder.