UDG_PololuMaestro User guide

Omar Alejandro Rodríguez Rosas

February 7, 2015

Contents

Chapter 1

Introduction

This library provides a hardware abstraction class to easily control the Pololu Maestro board. The current implementation supports servo outputs only, other features of the board sucha as general purpose PWM or digital/analog I/O are not yet available.
This guide assumes some degree of familiarity with the board itself an its features. It is strongly suggested that you read the Pololu Maestro's user guide before continuing any further with this document.

Chapter 2

Setup

2.1 Preparing your environment

1. Follow the instructions on the user guide to install Maestro Control Center, available for Linux and Windows. If using windows you might need to install a driver for the board.
2. Connect the board to your computer. If you're using ubuntu it is possible that an error flag on the Maestro activates (you'll see a red LED turn on). This behavior is expected and you can turn it off at the error tab on Maestro Control Center.
3. Open Maestro Control Center an make sure that the cannels you intend to use are enabled on the Status tab (see figure 2.1).
4. For safety reasons, it is important to verify that the min and max pulse widths allowed by the board match with those actually supported by your servos. These limits are usually provided by the vendor. You can verify this on the Channel settings tab at the Maestro Control Center (see figure 2.2).
5. Connect the servos to the channel of your choice (see the Maestro's user guide for the layout of your specific board). Remember the channel numbers

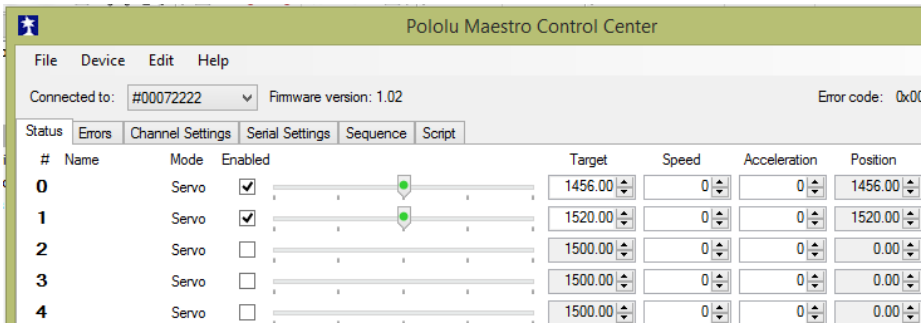

Figure 2.1: Status tab of the Maestro Control Center

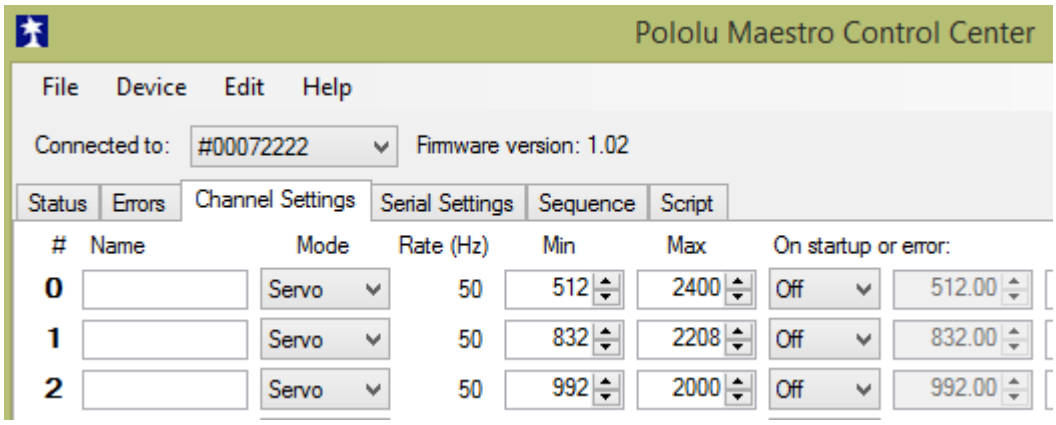

Figure 2.2: Channel settings tab of the Maestro Control Center

since they need to be explicitly declared in the code. Don forget to plug the DC supply for your servos (see the Maestro's user guide).

2.2 Building with UDG_PololuMaestro

This guide uses Microsoft Visual Studio Express for Desktop 2012 and Windows 8.1 or g++ 4.7 and Ubuntu 13.10. Other compiler/OS/Distribution combnations might work as well but haven't been tested.

2.2.1 Windows

1. Copy the UDG_PololuMaestro release folder to a suitable location.
2. Open your Visual Studio solution.
3. As an optional step, you might want to add UDG_PololuMaestro include (<Release_Folder >\include) and library (<Release_Folder>\include\bin\<Architecture>) directories to your default search folders. To do so, for the include folder go to Project\<Name>Properties\Configuration Properties\C/C++\General and enter the appropriate path in Additional Include Directories. Similarly, for the binary folder edit the Additional library directories property in Project\<Name>Properties\Configuratio Properties\Linker\General.
4. Your source files should use #include <UDG_PololuMaestro.h>or #include "UDG_PololuMaestro.h"[1]
5. Make sure you're linking with UDG_PololuMaestro.lib. To do so add UDG_PololuMaestro.lib to Additional Dependencies in Project\<Name>Properties\Configuration Prop-

[1]Remember that, when using quotes, If the header file is not on your current directory the full path should be included i.e. #include "c:\foo\bar\UDG_PololuMaestro.h".

erties\Linker\Input or add #pragma comment(lib,"UDG_PololuMaestro.lib") to your code. [2]
6. In order for UDG_PololuMaestro to work, serial settings of the board must be USB Dual Port. You can verify this at the serial settings tab on Maestro Control Center.

2.2.2 Linux (Ubuntu)

1. Copy the UDG_PololuMaestro release folder to a suitable location.
2. As an optional step you might want to place the header and library files into your default search directories. Typically you'll need to copy <Release_Folder>/include/UDG_PololuMaestro.h to /usr/include/ and <Release_Folder>/bin/<Architecture>/libUDG_PololuMaestro.a to /usr/lib.[3]
3. Your source files should use #include <UDG_PololuMaestro.h>or #include "UDG_PololuMaestro.h"[4]
4. To build your application use -std=c++11 [5] and link with -lUDG_PololuMaestro.a [6]

[2]Use the whole path if you skipped step 3.
[3]These default libraries might vary depending on your compiler settings.
[4]Remember that, when using quotes, If the header file is not on your current directory the full path should be included or you can also use the -I option (see g++ help).
[5]-std=c++0x might work as well if you're using an older g++ version
[6]Or simply UDG_PololuMaestro.a if you skipped step 2. If the file is not on your current folder you can use its whole path or the -L option (see g++ help).

Chapter 3

The UDG_PololuMaestro API

3.1 PololuMaestro class

Provides a software abstraction for the PololuMaestro board

3.1.1 public members

3.1.1.1 Error code definitions

ERROR_MAESTRO_SERIAL_SIGNAL 0x0001
Serial Signal Error (bit 0): A hardware-level error that occurs when a bytes stop bit is not detected at the expected place. This can occur if you are communicating at a baud rate that differs from the Maestros baud rate.

ERROR_MAESTRO_SERIAL_OVERRUN 0x0002
Serial Overrun Error (bit 1): A hardware-level error that occurs when the UARTs internal buffer fills up. This should not occur during normal operation.

ERROR_MAESTRO_SERIAL_RX_BUFFER_FULL 0x0004
Serial RX buffer full (bit 2): A firmware-level error that occurs when the firmwares buffer for bytes received on the RX line is full and a byte from RX has been lost as a result. This error should not occur during normal operation.

ERROR_MAESTRO_SERIAL_CRC_ERROR 0x0008
Serial CRC error (bit 3): This error occurs when the Maestro is running in CRC-enabled mode and the cyclic redundancy check (CRC) byte at the end of the command packet does not match what the Maestro has computed as that packets CRC (Section 5.d). In such a case, the Maestro ignores the command

packet and generates a CRC error.

ERROR_MAESTRO_SERIAL_PROTOCOL 0x0010
Serial protocol error (bit 4): This error occurs when the Maestro receives an incorrectly formatted or nonsensical command packet. For example, if the command byte does not match a known command or an unfinished command packet is interrupted by another command packet, this error occurs.

ERROR_MAESTRO_SERIAL_TIMEOUT 0x0020
Serial timeout error (bit 5): When the serial timeout is enabled, this error occurs whenever the timeout period has elapsed without the Maestro receiving any valid serial commands. This timeout error can be used to make the servos return to their home positions in the event that serial communication between the Maestro and its controller is disrupted.

ERROR_MAESTRO_SCRIPT_STACK 0x0040
Script stack error (bit 6): This error occurs when a bug in the user script has caused the stack to overflow or underflow. Any script command that modifies the stack has the potential to cause this error. The stack depth is 32 on the Micro Maestro and 126 on the Mini Maestros.

ERROR_MAESTRO_SCRIPT_CALL_STACK 0x0080
Script call stack error (bit 7): This error occurs when a bug in the user script has caused the call stack to overflow or underflow. An overflow can occur if there are too many levels of nested subroutines, or a subroutine calls itself too many times. The call stack depth is 10 on the Micro Maestro and 126 on the Mini Maestros. An underflow can occur when there is a return without a corresponding subroutine call. An underflow will occur if you run a subroutine using the Restart Script at Subroutine serial command and the subroutine terminates with a return command rather than a quit command or an infinite loop.

ERROR_MAESTRO_SCRIPT_PROGRAM_COUNTER 0x0100 Script program counter error (bit 8): This error occurs when a bug in the user script has caused the program counter (the address of the next instruction to be executed) to go out of bounds. This can happen if your program is not terminated by a quit, return, or infinite loop.

3.1.1.2 PololuMaestro(HANDLE fd)

This constructor method initializes the PololuMaestro using the provided Serial Port descriptor. By default, the number of channels is set to 18 (assumes Pololu Maestro 18 board). For a different number of channels, use PololuMaestro(HANDLE, int). **HANDLE fd:** A file descriptor for a Serial Port (COM ports in windows, /dev/tty in linux). You can get a file descriptor using the following function:

Windows

```
HANDLE OpenSerialPort(LPWSTR name)
{
        HANDLE portDescriptor;
        DCB deviceControlBlock;
        portDescriptor = CreateFile(name,
                                    GENERIC_READ|
                                        GENERIC_WRITE,
                                    0,0,OPEN_EXISTING,0,0);
        if(portDescriptor == INVALID_HANDLE_VALUE)
        {
                printf("Failed opening %ls\n", name);
                return INVALID_HANDLE_VALUE;
        }

        FillMemory(&deviceControlBlock, sizeof(DCB),0);
        deviceControlBlock.DCBlength = sizeof(DCB);
        if(BuildCommDCB(L"57600,n,8,1",&
            deviceControlBlock))
        {
                printf("Successfully configured DCB!\n");

        }
        else
        {
                printf("Something went wrong while
                    building the DCB\n");
                return INVALID_HANDLE_VALUE;
        }

        if (!SetCommState(portDescriptor,&
            deviceControlBlock))
        {
                printf("Error configuring port\n");
                return INVALID_HANDLE_VALUE;
        }
        else
        {
                printf("Serial port ready\n");
        }

        return portDescriptor;
}
```

Linux

```
int OpenSerialPort(const char* device)
{
        int fd = open(device, O_RDWR | O_NOCTTY);
        struct termios options;
        tcgetattr(fd, &options);
        options.c_lflag &= ~(ECHO | ECHONL | ICANON |
            ISIG | IEXTEN);
        options.c_oflag &= ~(ONLCR | OCRNL);
        tcsetattr(fd, TCSANOW, &options);
        return fd;
}
```

3.1.1.3 PololuMaestro(HANDLE fd,int nchannels)

This constructor method initializes the PololuMaestro using the provided Serial Port descriptor.The number of channels is set to *nchannels*. You should set *nchannels* accordingly to your board (18 for Maestro 18, 24 for Maestro 24, etc.).
HANDLE fd: A file descriptor for a Serial Port (COM ports in windows, /dev/tty in linux (See 3.1.1.2 for a code sample).
int nchannels: The number of channels on your board.

3.1.1.4 PololuMaestro()

Destructor method of the class. Cleans all of the resources requested by the class.

3.1.1.5 bool SetAngle(int channel, float angle)

Approximates the position specified by *angle* for the motor connected to *channel* based on the constraints specified by the ServoMotor class instance at channels[*channel*] (see 3.2). The appropriate pulse to achieve *angle* is calculated using $\frac{a+r}{2r}(p_{max} - p_{min}) + p_{min}$ where a is the angle specified by *angle*, r is the movement range of the servo, p_{max} is the max pulse supported by the servo and p_{min} is the servo's minimum pulse. If $|angle|$ is greater than $|r|$, a is capped to $\pm r$.
int channel: The channel where the desired motor is connected.
float angle: Signed angle in degrees relative to the neutral position.

Return value: true if the operation suceeds, false if it doesn't.

3.1.1.6 float GetAngle(int channel)

Gets the approximate position in degrees of the motor connected at *channel*, relative to its neutral position. This angle is approximated using $\frac{p-p_{min}}{p_{max}-p_{min}}2r - r$ where p is the current pulse at *channel*, r is the movement range of the servo, p_{max} is the max pulse supported by the servo and p_{min} is the servo's minimum pulse.

Return value: The approximate motor position in degrees relative to the neutral position.

3.1.1.7 bool SetTarget(int channel,unsigned short targetUs)

Sets *targetUs* as the pulse at channel *channe*. If *targetUS* is outside the range of the motor range (specified at the associated ServoMotor class instnace) the value is capped to the appropriate max or min value.
int channel: the number of the channel to set.
unsigned short targetUs: target pulse in μs.

Return value: true if the operation succeeds, false if it doesn't.

3.1.1.8 bool SetSpeed(int channel,unsigned short speed)

Limits the speed at which a servo connected to *channel* changes its output value. The speed limit is given in units of (0.25 μs)/(10 ms) by *speed*. A speed of 0 makes the speed unlimited, so that setting the target will immediately affect the position. Note that the actual speed at which your servo moves is also limited by the design of the servo itself, the supply voltage, and mechanical loads; this parameter will not help your servo go faster than what it is physically capable of. At the minimum speed setting of 1, the servo output takes 40 seconds to move from 1 to 2 ms.

int channel: the number of the channel to set.
unsigned short speed: the speed limit in units of (0.25 μs)/(10 ms)

Return value: true if the operation succeeds, false if it doesn't.

3.1.1.9 bool SetAcceleration(int channel,unsigned short acceleration)

Limits the acceleration of the servo connected to *channel*. The acceleration limit is a value from 0 to 255 in units of (0.25 μs)/(10 ms)/(80 ms) specified by *acceleration*. A value of 0 corresponds to no acceleration limit. An acceleration limit causes the speed of a servo to slowly ramp up until it reaches the maximum speed, then to ramp down again as position approaches target, resulting in a relatively smooth motion from one point to another.
int channel: the number of the channel to set.
unsigned short acceleration: the acceleration limit in units of (0.25 μs)/(10

ms)

Return value: true if the operation succeeds, false if it doesn't.

3.1.1.10 unsigned short GetPosition(int channel)

Gets to current pulse at *channel*.
int channel: the channelto get the pulse with from.

Return value: The current pulse width of the selected channel.

3.1.1.11 bool AssignChannel(int channel, ServoMotor motor)

Assigns the ServoMotor instance *motor* to channel *channel*. The information provided by this instance will be used to determine important security and operational constraints.
int channel: the number of the channel to be assigned.
ServoMotor motor: An abstraction of the motor being used

Return value: true if the operation succeeds, false if it doesn't.

3.1.1.12 bool IsMoving()

Indicates if there is at least one channel that hasn't reached its target position. This command works properly only with speed or acceleration constrained channels.

Return value: true if there is at least one channel that hasn't reached it target pulse, false if it doesn't.

3.1.2 short GetErrors()

Sets a short (2 bytes) integer in which each bit corresponds to a particular error detectable by the Maestro. See 3.1.1.1 for the specific codes and their description.
Return value: A short integer containing the error the applicable codes.

3.1.3 short GoHome()

Sets al outputs to their default home values.

3.1.4 Private members

3.1.4.1 HANDLE deviceDescriptor

A file descriptor associated to the device serial communication port. HANDLE type is defined as int in linux.

3.1.5 ServoMotor* channels

An abstraction of the Maestro's channels. Memory for this array is allocated at runtime and it's freed in the destructor method.

3.1.5.1 int numberOfChannels

The number of channels on the board. Thi is set at the constructor method and must match the number of physically available channels on the board.

3.1.5.2 bool WriteToDevice(unsigned char* buffer, int nBytes)

Writes *nBytes* bytes from *buffer* to the associated *deviceDescriptor*. Returns true if the operation succeeds, false if it doesn't.

3.1.5.3 bool ReadFromDevice(unsigned char* buffer, int nBytes)

Reads *nBytes* bytes from the associated *deviceDescriptor* into *buffer*. Returns true if the operation succeeds, false if it doesn't.

3.2 ServoMotor class

3.2.1 int minPulse

The servo's minimum pulse width in μs.

3.2.2 int maxPulse

The servo's max supported pulse width in μs.

3.2.3 int neutralPulse

The servo's max supported pulse width in μs.

3.2.4 int degreeRange

The amount of degrees the servo can move from it neutral position its max or minimum (i.e. for 180 servo degreeRange = 90).

3.2.5 int AngleToPulse(float angle)

Converts *angle* to its equivalent pulse width according to the class parameters.
float angle: The (signed) angle to convert.

Return value: The equivalent pulse width for *angle*.

3.2.6 void SetParameters(int _minPulse, int _maxPulse, int _neutralPulse, int _degreeRange)

Sets values for the servo's parameters.
int _minPulse The value of minPulse.
int _maxPulse The value of maxPulse.
int _neutralPulse The value of neutralPulse.
int _degreeRange The value of degreeRange.

Chapter 4

Code Samples

The code samples work for the Microsoft c++ compiler (cl, using Visual Studio Express 2012 for desktop) and g++ 3.8. Other compilers might work as well but haven't been tested yet. Windows examples assume an unicode build.

4.1 Opening a serial port descriptor

For both Linux and Windows implementations, you need to provide a file descriptor associated with a serial port. The following samples demonstrate how to code a function to do so.
textbfNOTE: The name of the serial port associated with your board varies from system to system. You'll notice that two virtual communication ports with contiguous numbering will register themselves when connecting the board. You need to use the one with the higher number. Typically this will be COMx for Windows and ttyACMx for linux.

4.1.1 Windows

```
HANDLE OpenSerialPort(LPWSTR name)
{
        HANDLE portDescriptor;
        DCB deviceControlBlock;
        portDescriptor = CreateFile(name,
                                    GENERIC_READ|
                                        GENERIC_WRITE,
                                    0,0,OPEN_EXISTING,0,0);
        if(portDescriptor == INVALID_HANDLE_VALUE)
        {
```

```
                printf("Failed opening %ls\n", name);
                return INVALID_HANDLE_VALUE;
        }

        FillMemory(&deviceControlBlock, sizeof(DCB),0);
        deviceControlBlock.DCBlength = sizeof(DCB);
        if(BuildCommDCB(L"57600,n,8,1",&
            deviceControlBlock))
        {
                printf("Successfully configured DCB!\n");

        }
        else
        {
                printf("Something went wrong while
                    building the DCB\n");
                return INVALID_HANDLE_VALUE;
        }

        if (!SetCommState(portDescriptor,&
            deviceControlBlock))
        {
                printf("Error configuring port\n");
                return INVALID_HANDLE_VALUE;
        }
        else
        {
                printf("Serial port ready\n");
        }

        return portDescriptor;
}
```

4.1.2 Linux

NOTE: In the linux implementation, HANDLE is defined as int.

```
int OpenSerialPort(const char* device)
{
        int fd = open(device, O_RDWR | O_NOCTTY);
        struct termios options;
  tcgetattr(fd, &options);
  options.c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG |
      IEXTEN);
  options.c_oflag &= ~(ONLCR | OCRNL);
  tcsetattr(fd, TCSANOW, &options);
```

```
  return fd;
}
```

4.2 Typical applications

This section presents typical use cases and some basic operations.

4.2.1 Initialize Servo and PololuMaestro objects

Once a serial port is open, you need to initialize a ServoMotor object (matching the features of your physical servo). Then you'll assign that servo to a maestro channel.

```
#include "UDG_PololuMaestro.h"
//OpenSerialPort() definition
//{...}
int main()
{
fd = OpenSerialPort(PortName);
ServoMotor motor;
motor.SetParameters(992,//min pulse
                    2000,//max pulse
                    1500,//neutral pulse
                    90//degree range
                    );
PololuMaestro maestro(fd,
                    18//number of channels on the board
                    );
maestro.AssignChannel(0,motor); //assign to channel 0
return 0;
}
```

4.2.2 Moving the servos: blocking and non-blocking

To change the position of your servos, you can use PololuMaestro::SetTarget() and PololuMaestro::SetAngle(). Remember that both functions will return immediatly even if the position hasn't been physically reached. To avoid this you can set a speed or acceleration limit (or both) and use PololuMaestro::isMoving() function to wait until all motors reach their position. This sample demonstrates both possibilities.

```
#include "UDG_PololuMaestro.h"
//OpenSerialPort() definition
```

```
//{...}
int main()
{
fd = OpenSerialPort(PortName);
ServoMotor motor;
motor.SetParameters(992,2000,1500,90);
PololuMaestro maestro(fd,18);
maestro.AssignChannel(0,motor);
//Go to neutral position
maestro.SetTarget(0,motor.neutralPulse);
//function returns immediately, you might want
//to include a Sleep/Delay routine

//set a speed limit of 20
maestro.SetSpeed(0,20);
//Set an angle as target and wait until is reached
float angle = 45;
maestro.SetAngle(0,angle);
while (maestro.IsMoving());

return 0;
}
```